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

Clément Legrand-Lixon

25 juillet 2018

Résumé

De nombreux algorithmes font régulièrement leur apparition pour tenter de résoudre des problèmes d'optimisation combinatoire, comme le *Capacitated Vehicle Routing Problem*, ou d'améliorer des solutions existantes. Dans la plupart des problèmes, il est possible d'extraire de la connaissance de petites instances résolues, afin de guider 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*, souvent plus performante que l'algorithme initial.

Mots-clés : *Optimisation combinatoire*, *Capacitated Vehicle Routing Problem*, *Extraction de connaissances*, *Intégration de connaissances*, *Learning Heuristic*

Référence : Stage effectué du 22 Mai 2018 au 27 Juillet 2018 au laboratoire CRIStAL à Lille sous la direction de Laetitia Jourdan, Univ Lille, CNRS - UMR 9189, F-59600 Lille, France.

Introduction

Les problèmes d'optimisation combinatoire sont généralement des problèmes difficiles à résoudre [9], 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 combinatoire. 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, correspondant à la capacité du véhicule de transport. 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...), comme le montre [10] dans le cas du CVRP, 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 [3], 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.

La section 1 commence par présenter le problème étudié, et introduit les notations et opérateurs utilisés dans la suite. L'objectif fixé et la méthode mise en place pour y parvenir y sont également présentés. La section 2 explique comment obtenir une solution initiale au problème de bonne qualité, puis la section 3 propose un algorithme d'optimisation utilisé pour la résolution du problème. La section 4 décrit comment extraire de la connaissance des instances étudiées. Enfin, la section 5 explique comment intégrer de la connaissance à un algorithme d'optimisation et présente une nouvelle *learning heuristic* pour résoudre le problème.

1 Présentation des notions et notations

Cette section introduit dans un premier temps le problème étudié, en rappelant au préalable ce qu'est un problème d'optimisation combinatoire. Les différentes notations utilisées dans la suite du papier sont ensuite décrites, enfin l'objectif de cet article est présenté.

1.1 Description du problème

De nombreux problèmes sont dits d'optimisation combinatoire, 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 combinatoire, puis présente le problème étudié : *CVRP*

1.1.1 Optimisation combinatoire

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

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

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

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 combinatoire, où sont donnés n points de coordonnées (x_i, y_i), représentant 1 dépôt et $n-1$ clients.

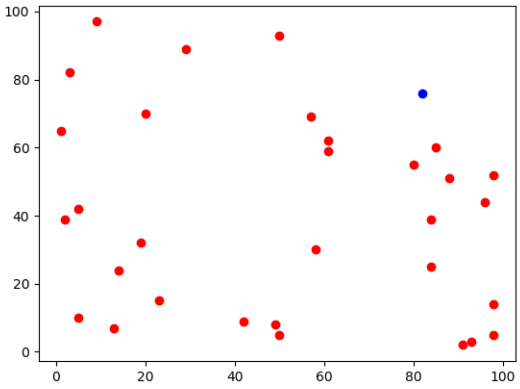

FIGURE 1 – Représentation de l'instance A-n32-k05 de la littérature (31 clients et 1 dépôt).

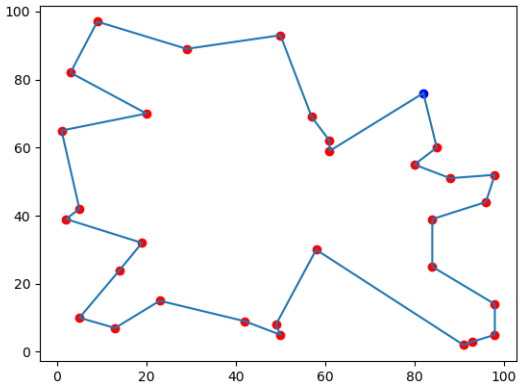

FIGURE 2 – Représentation d'une solution de l'instance A-n32-k05.

On dispose également d'une flotte de k véhicules. L'objectif est de minimiser la longueur du réseau (i.e. l'ensemble des tournées effectuées par les véhicules, une *tournée* correspondant à l'ensemble des clients desservis par un véhicule). En reprenant les notations de [1], on définit $x_{i,j}^v$ 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. Ainsi une solution Sol du problème est la donnée d'une matrice, telle que $Sol[i][j] = c_{i,j}x_{i,j}^v$. On cherche donc à déterminer :

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

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 : $\forall i > 0 \sum_{j=1}^{n} \sum_{v=1}^{k} x_{i,j}^v = 1$;
— Chaque tournée doit partir et s'arrêter au dépôt : $\forall v \sum_{j=1}^{n} x_{0,j}^v = 1$ et $\sum_{j=1}^{n} x_{j,0}^v = 1$.

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 : $\forall v \sum_{i=0}^{n} \sum_{j=0}^{n} x_{i,j}^v d_j < C$.

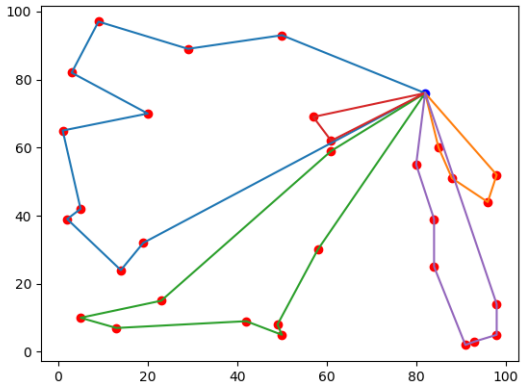

FIGURE 3 – Représentation d'une solution de l'instance A-n32-k05, où les demandes des clients sont prises en compte. Elle comporte 5 tournées.

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.

Les instances du CVRP étudiées par la suite, sont disponibles sur [11]

1.1.4 Méthodes de résolution de CVRP

L'article [10] présente les différents types d'algorithme et méthodes qui existent pour résoudre le CVRP (et plus généralement un problème d'optimisation combinatoire). Il distingue principalement deux types de méthodes : les méthodes exactes et les méthodes approchées.

Les méthodes exactes, ou encore méthodes *complètes*, permettent de trouver la solution optimale au problème. Ces méthodes sont basées sur des explorations exhaustives de l'ensemble des solutions réalisables. Toutefois ces méthodes basiques restent inappropriées aux problèmes d'optimisation combinatoire. Il existe néanmoins des algorithmes qui permettent de restreindre l'ensemble à explorer en éliminant des sous-ensembles de mauvaises solutions à l'aide de techniques, dites d'élagage.

Contrairement aux méthodes exactes, les méthodes approchées sont dites *incomplètes*, car elles permettent de trouver de bonnes solutions, mais ne garantissent pas l'optimalité de celles-ci. Cette méthode regroupe notamment les heuristiques et méta-heuristiques.

Une heuristique est un moyen de guider les choix que doit faire un algorithme pour réduire sa complexité. Une heuristique est spécifique à un problème et ne peut donc pas être généralisée.

Une méta-heuristique peut être considérée comme une heuristique "puissante et évoluée", dans la mesure où elle est généralisable à plusieurs problèmes d'optimisation. On classe souvent les méta-heuristiques en fonction du nombre de solutions qu'elles manipulent. Celles à solution unique (recherche tabou, descente du gradient), et celles à population de solutions (algorithmes génétiques et colonies de fourmis).

De plus les méthodes approchées doivent trouver un équilibre entre deux tendances opposées : l' *intensification* (ou *exploitation*) et la *diversification* (ou *exploration*). **L'intensification** de la recherche signifie que celle-ci se concentre autour des meilleures solutions rencontrées, considérées prometteuses. Alors que **la diversification** incite davantage la recherche à explorer des nouvelles zones de l'espace de solutions.

Ainsi les méta-heuristiques à solution unique ont plus tendance à l'exploitation du voisinage de la solution en question, et les approches à base de population de solutions ont plutôt tendance à l'exploration.

1.2 Parcours et exploration des voisinages

Il existe plusieurs moyens pour procéder à une diversification de la solution courante [6]. En effet, il est possible qu'en modifiant légèrement la solution courante, on obtienne une autre solution réalisable, appelée *voisin* de la solution courante. On appelle alors *voisinage* de la solution courante, l'ensemble des voisins de cette solution. Une fonction qui attribue à une solution, l'un de ses voisins, est appelée *opérateur de voisinage*.

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 un voisinage :

— 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 trois 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 un certain voisinage qui sera choisi 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 [4], amélioré par [2]. Ainsi, nous commençons par décrire l'algorithme utilisé, puis détaillons le problème du choix des paramètres pour son exécution.

2.1 Description de l'algorithme Clarke & Wright

L'algorithme Clarke & Wright (CW) est un algorithme glouton. L'initialisation *Init* la plus courante pour cet algorithme est d'attribuer un véhicule à chaque clients (de cette manière la contrainte sur le nombre de véhicules disponibles n'est pas respectée). Ensuite les tournées sont fusionnées en fonction des *savings* (économies en français) calculées. L'algorithme 1 présente le fonctionnement de l'algorithme CW.

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. Le paramètre λ a été introduit par Gaskell [5] et Yellow [12], et est appelé *route shape parameter*. Le paramètre μ prend en compte l'asymétrie entre les clients i et j, en tenant compte de leur distance respective au dépôt. Il a été introduit par Paessens [8]. Enfin, le paramètre ν a été ajouté, en s'inspirant d'une méthode de résolution du *bin packing problem* (BPP), développée par Martello et Toth [7], qui consiste à s'intéresser en priorité aux éléments les plus gros et à les placer en premier.

Un exemple d'exécution de l'algorithme avec $(\lambda, \mu, \nu) = (1, 1, 1)$, sur l'instance A-n37-k06, est 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. Pour cela il existe une heuristique, appelée *Lin-Kernighan*, que nous décrirons en section 3.1.4.

Algorithm 1: CLARKE-WRIGHT calcule une solution initiale

Input: Une solution initiale $Init$, 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

```
Calculer les savings de toutes les arêtes de I
Sol ← Init
while max_{i,j} s(i,j) > 0 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
```

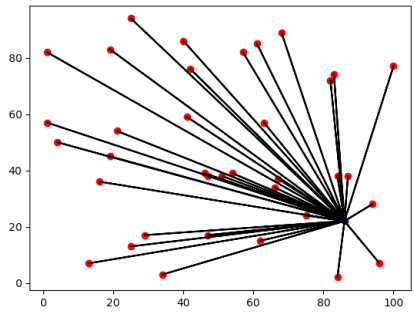

FIGURE 4 – Initialisation.

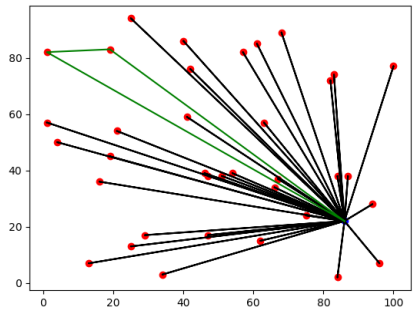

FIGURE 5 – 1^{ere} fusion.

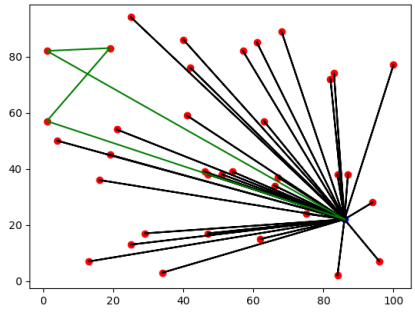

FIGURE 6 – 2^{eme} fusion.

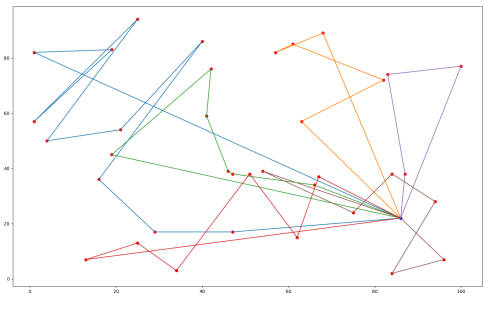

FIGURE 7 – Solution obtenue, $cost = 1297$.

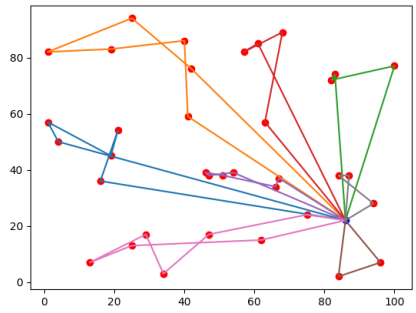

FIGURE 8 – $(\lambda, \mu, \nu) = (1.9, 0.1, 1.5)$, $cost = 1106$.

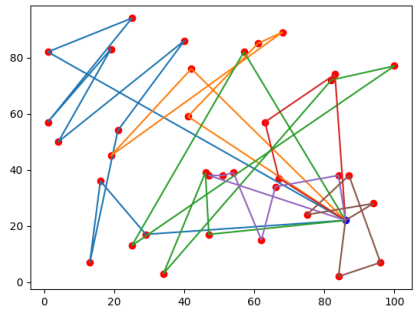

FIGURE 9 – $(\lambda, \mu, \nu) = (0.1, 0.1, 0.1)$, $cost = 1569$.

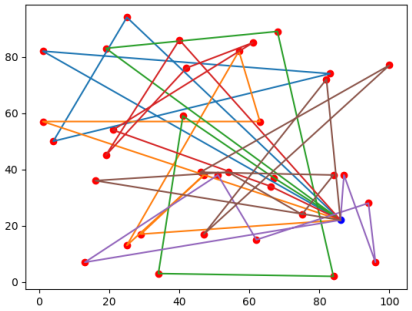

FIGURE 10 – $(\lambda, \mu, \nu) = (0.0, 1.0, 1.5)$, $cost = 2191$.

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 [2] 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 une précision inférieure au dixième lorsqu'on choisit les valeurs des paramètres.

Les figures 8 à 10 présentent différents résultats obtenus pour différents triplets (λ, μ, ν). 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.

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 [3], dont nous détaillons d'abord le fonctionnement, avant de décrire davantage les mécanismes mis en jeu lors de son exécution.

3.1 Heuristique de Arnold & Sörensen

L'heuristique proposée par Arnold et Sörensen [3], 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 $BI - O$ (cf section 1.2), 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).

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

```
1  Sol ← CW(I, D, λ, μ, ν)
2  N ← Size(D)
3  nextSol ← Sol
4  while La dernière amélioration date de moins de 3 min do
5      worstEdge ← Calcul de la pire arête
6      nextSol ← EC_{BI−O}(worstEdge, I, D)
7      nextSol ← LK_{BI−O}(nextSol)
8      nextSol ← CE_{BI−O}(worstEdge, I, D)
9      nextSol ← LK_{BI−O}(nextSol)
10     if cost(Sol) > cost(nextSol) then
11         Sol ← nextSol
12     if Pas d'améliorations depuis N/10 itérations then
13         Appliquer les opérateurs sur toutes les arêtes de la solution
14     if Pas d'améliorations depuis 20N itérations then
15         Changer de fonction de pénalisation en prenant un autre triplet (γ_w, γ_c, γ_d)
16     if Pas d'améliorations depuis 100N itérations then
17         Réinitialiser les pénalités des arêtes
18 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 [3] :

— 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 [3] $\beta = 0.1$. $p(i, j)$ correspond au nombre de fois où l'arête (i, j) a été pénalisé ;

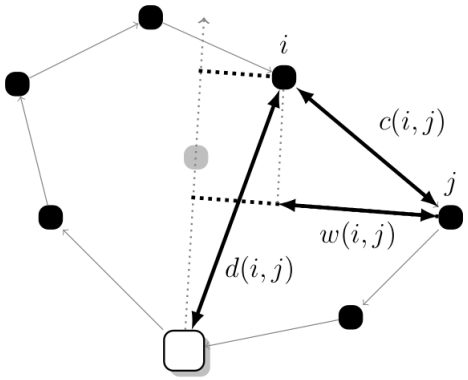

FIGURE 11 – Illustration des caractéristiques d'une arête.

— 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 :

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

Les paramètres $\gamma_w, \gamma_c, \gamma_d$, 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 $\gamma_w = \gamma_c = 0$, puisqu'il fournit $b(i,j) = 0$.

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

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

Les opérateurs de voisinage, présentés ci-après, vont orienter leurs recherches autour de cette pire arête.

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.

Supposons que l'on veuille supprimer une arête (c_1^-, c_1) d'une tournée r_1. On commence par chercher le plus proche voisin c_2 de c_1 appartenant à une tournée r_2 dans laquelle il est possible d'ajouter c_1. On insère alors c_1 après c_2 sur r_2. Ainsi la portion $[..., c_2, c_2^+, ...]$ de r_2 se transforme en $[..., c_2, c_1, c_2^+, ...]$. On cherche ensuite une arête à supprimer sur r_2, pour éjecter un client sur une tournée r_3, et ainsi de suite jusqu'à atteindre la tournée r_l. Le fonctionnement de cet opérateur est présenté sur la figure 12.

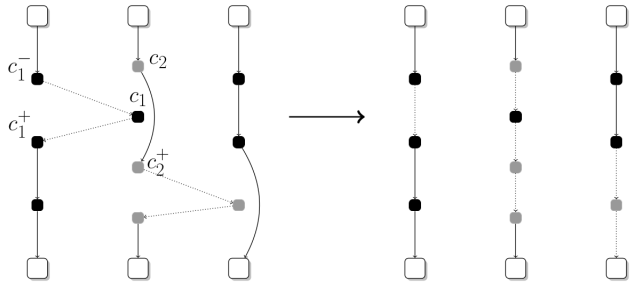

FIGURE 12 – Exemple de fonctionnement de l'opérateur ejection-chain.

Dans l'article [3] $l = 3$. 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.

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 $possibleSol \leftarrow sol$
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 3, il est possible d'utiliser les méthodes de la section 1.2 pour explorer les voisinages.

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.

Supposons que l'on veuille supprimer une arête (c_1, c_2) appartenant à une tournée r_1. On cherche le plus proche voisin c_4 de c_1 appartenant à une tournée r_2. On échange alors c_1 avec le prédécesseur c_3 de c_4. Il suffit ensuite de considérer deux autres clients entre les deux tournées, pour échanger deux séquences de clients. 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 $O(n^2)$.

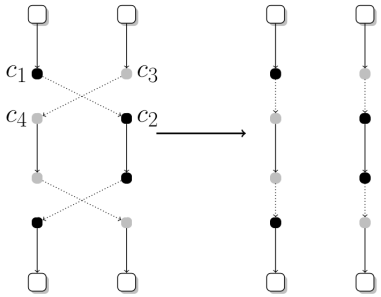

FIGURE 13 – Exemple de fonctionnement de l'opérateur cross-exchange.

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*

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.

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. L'algorithme 5 décrit le fonctionnement de l'opérateur lorsque $k = 2$, valeur choisie dans [3].

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

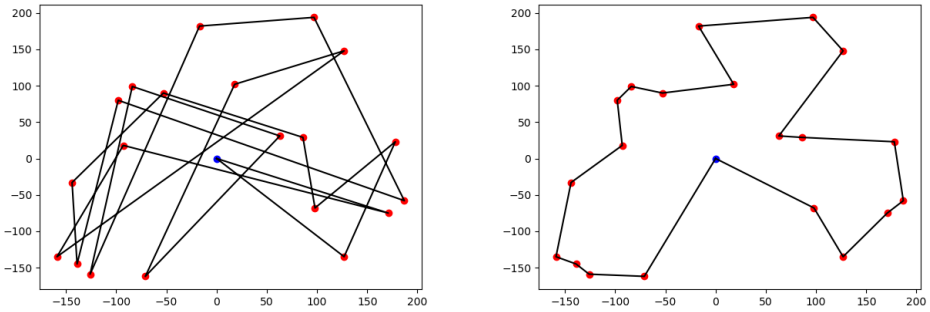

FIGURE 14 – Exemple de fonctionnement de l'opérateur LK.

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

```
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.

3.2 Algorithme d'optimisation utilisé

L'algorithme d'optimisation que nous utilisons est un peu différent de l'heuristique A&S précédente.

Nous changeons la condition d'arrêt, en prenant *limitTime* secondes (au lieu de 180). Cela permet d'adapter la condition d'arrêt pour chaque instance. Pour que l'exploration du voisinage des solutions soit plus efficace, nous décidons de passer les opérateurs *EC* et *CE* en mode $FI - RD$ (i.e. on prend le premier voisin qui améliore la solution courante, en parcourant le voisinage de manière aléatoire). Enfin, nous ajoutons une condition de *Restart* , s'il n'y a pas eu d'améliorations depuis *restartTime* secondes. Cela consiste à conserver $a\%$ des arêtes de la meilleure solution obtenue jusqu'à présent. On relie ensuite les arêtes qui ont un client en commun pour former des tournées. On applique alors l'algorithme CW sur les tournées obtenues (en conservant les mêmes paramètres qu'au début de l'algorithme).

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: L'ensemble des valeurs prises par *Sol* au cours de l'algorithme

$allSol \leftarrow \{\}$
$Sol \leftarrow CW(I, D, \lambda, \mu, \nu)$
$allSol \leftarrow allSol \cup \{(cost(Sol), Sol)\}$
$N \leftarrow length(D)$
$nextSol \leftarrow Sol$
while *La dernière amélioration date de moins de limitTime secondes* **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**
    $Sol \leftarrow nextSol$
    $allSol \leftarrow allSol \cup \{(cost(Sol), Sol)\}$
  if *Pas d'améliorations depuis restartTime secondes* **then**
    fixer $a\%$ des arêtes de *Sol*
    Construire une solution initiale *Init* avec les arêtes fixées
    $nextSol \leftarrow CW(Init, I, D, \lambda, \mu, \nu)$
  if *Pas d'améliorations depuis resetTime secondes* **then**
    Changer de fonction de pénalisation en prenant un autre triplet $(\gamma_w, \gamma_c, \gamma_d)$
    Réinitialiser les pénalités des arêtes
Trier *allSol*
return *allSol*

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

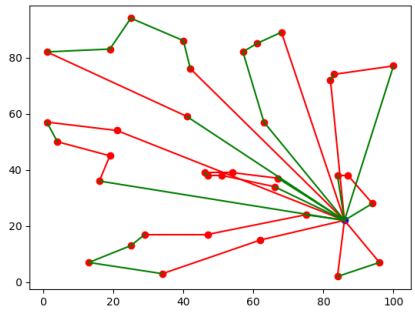

FIGURE 15 – $CW(1.9, 0.1, 1.5) + LK$, $cost = 1041$, 19 arêtes optimales.

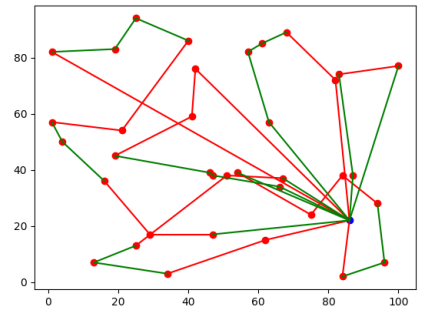

FIGURE 16 – $CW(0.1, 0.1, 0.1) + LK$, $cost = 1170$, 19 arêtes optimales.

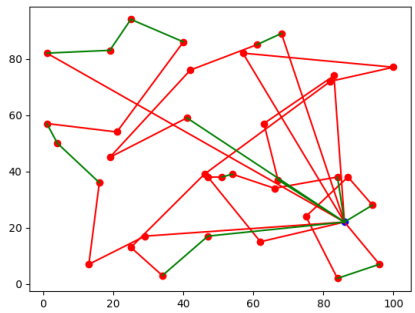

FIGURE 17 – $CW(0.0, 1.0, 1.5) + LK$, $cost = 1600$, 11 arêtes optimales.

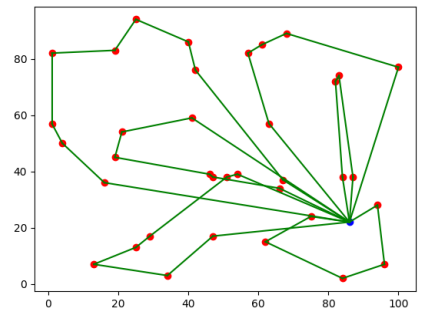

FIGURE 18 – Solution optimale, 42 arêtes.

4 Extraction de la connaissance

Cette section s'intéresse à l'extraction de connaissance à partir de solutions initiales générées. Nous expliquons dans un premier temps quelle va être la connaissance extraite des solutions initiales, puis nous décrivons 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.

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) (i.e. 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 (i.e. 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

4.3.1 Choix de l'échantillon et de la base

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 figures 19 à 24, présentent les trois instances, ainsi que les solutions optimales associées.

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.

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 ;

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. Les résultats obtenus ne nous permettent toutefois pas de choisir le meilleur critère pour l'apprentissage. Nous décidons alors de raffiner nos résultats.

4.3.2 Étude des critères Rang et Seuil

Afin de comparer ces deux critères, nous allons étudier la répartition des coûts des solutions initiales obtenues pour différentes valeurs de Rang et de Seuil. Pour chacune des instances ci-dessus, nous générons un échantillon de taille 50, et utilisons la base $Qual_{10}$. Nous conservons aussi les 10 triplets (λ, μ, ν) qui donnent les meilleures solutions dans l'échantillon. Pour chaque critère, on apprend à partir de $Qual_{10}$, puis on applique CW avec les arêtes conser-

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	10	6	0.14	10	10	6	0.14	10	10	7	0.16
	20	20	13	0.31	20	20	13	0.32	20	20	13	0.31
	18	18	12	0.28	18	18	13	0.3	18	18	12	0.28
100	10	10	9	0.21	10	10	9	0.21	10	10	10	0.24
	20	20	16	0.38	20	20	16	0.38	20	20	15	0.36
	18	18	13	0.3	18	18	13	0.3	18	18	12	0.29
500	10	10	9	0.21	10	10	10	0.24	10	10	9	0.21
	20	20	16	0.38	20	20	16	0.38	20	20	15	0.36
	18	18	13	0.3	18	18	13	0.3	18	18	12	0.28
Complet	10	10	8	0.19	10	10	9	0.21	10	10	7	0.17
	20	20	14	0.33	20	20	14	0.33	20	20	14	0.33
	18	18	12	0.29	18	18	12	0.29	18	18	12	0.29

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	10	6	0.08	10	10	7	0.1	10	10	7	0.1
	20	20	14	0.2	20	20	15	0.21	20	20	14	0.19
	33	33	23	0.32	33	33	26	0.36	33	33	24	0.33
100	10	10	6	0.08	10	10	7	0.1	10	10	7	0.1
	20	20	16	0.22	20	20	16	0.22	20	20	14	0.19
	33	33	26	0.36	33	33	26	0.36	33	33	25	0.34
500	10	10	7	0.1	10	10	7	0.1	10	10	6	0.08
	20	20	17	0.23	20	20	15	0.21	20	20	13	0.18
	33	33	27	0.37	33	33	26	0.36	33	33	25	0.34
Complet	10	10	7	0.1	10	10	7	0.1	10	10	6	0.08
	20	20	17	0.23	20	20	17	0.23	20	20	13	0.18
	33	33	27	0.37	33	33	27	0.37	33	33	25	0.34

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	10	8	0.08	10	10	8	0.08	10	10	8	0.08
	20	20	18	0.17	20	20	17	0.16	20	20	18	0.17
	50	50	43	0.41	50	50	44	0.43	50	50	44	0.43
100	10	10	8	0.08	10	10	8	0.08	10	10	8	0.08
	20	20	18	0.17	20	20	18	0.17	20	20	18	0.17
	50	50	46	0.44	50	50	46	0.44	50	50	46	0.44
500	10	10	8	0.08	10	10	8	0.08	10	10	8	0.08
	20	20	18	0.17	20	20	18	0.17	20	20	18	0.17
	50	50	46	0.44	50	50	46	0.44	50	50	46	0.44
Complet	10	10	8	0.08	10	10	8	0.08	10	10	8	0.08
	20	20	18	0.17	20	20	18	0.17	20	20	18	0.17
	50	50	46	0.44	50	50	46	0.44	50	50	46	0.44

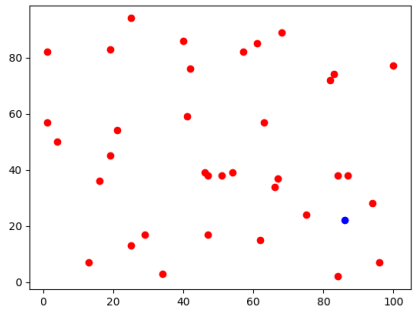

FIGURE 19 – Instance *A-n37-k06*.

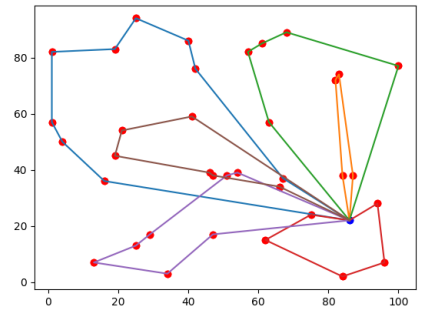

FIGURE 20 – Solution optimale *A-n37-k06*, 42 arêtes.

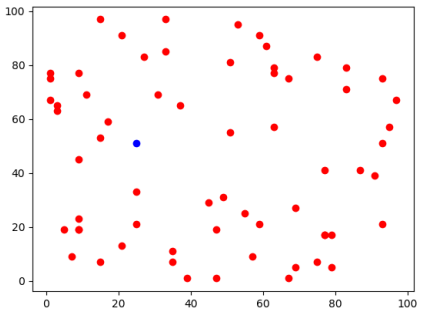

FIGURE 21 – Instance *A-n65-k09*.

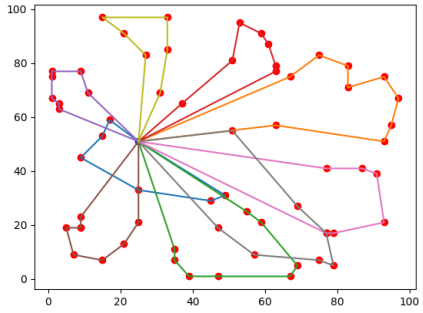

FIGURE 22 – Solution optimale *A-n65-k09*, 73 arêtes.

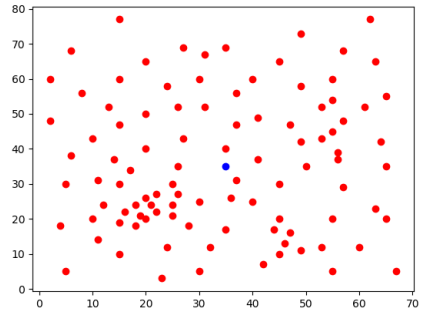

FIGURE 23 – Instance *P-n101-k04*.

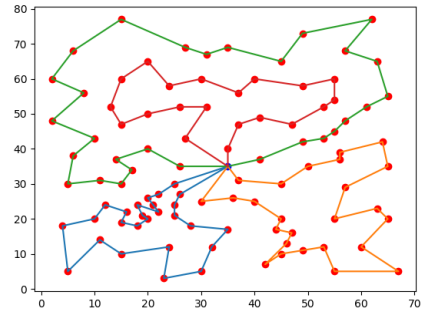

FIGURE 24 – Solution optimale *P-n101-k04*, 104 arêtes.

vées et les triplets choisis. De cette manière, on dispose de 100 solutions initiales calculées pour chaque critère.

Les résultats obtenus sont présentés sur les figures 25 à 30. Quelle que soit l'instance, on remarque les apprentissages avec le critère Rang donnent de meilleurs résultats que les apprentissages avec le critère Seuil. Nous pouvons remarquer qu'avec $seuil = S_{lb}/2$, nous obtenons les meilleurs résultats avec le critère Seuil. En ce qui concerne le critère Rang, plus le rang augmente et plus les solutions obtenues sont de meilleure qualité. Cependant, un grand nombre d'arêtes contraint davantage les solutions initiales. On préférera donc un rang compris entre

$n/2$ et $4n/5$ selon le nombre d'arêtes que l'on souhaite conserver.

5 Intégration de la connaissance

Cette section s'intéresse à l'intégration de connaissance dans l'algorithme d'optimisation retenu, i.e. 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 32, où les arêtes retenues sont vertes. 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 tournées, 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 associé à la solution de coût le plus faible présente dans l'échantillon, i.e. 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.

Cette heuristique commence par effectuer un apprentissage, comme vu en section 4. Nous obtenons de cette manière le triplet $(\lambda^*, \mu^*, \nu^*)$, utilisé ensuite, ainsi qu'une solution initiale *Init*. Ensuite, pendant *NbIterations*, on récupère les solutions renvoyées par H_c, puis on apprend à partir de ces solutions. Le critère employé est le Rang, et sa valeur varie entre deux bornes au cours de l'exécution. Cela permet de relâcher la contrainte sur le nombre d'arêtes conservées. Finalement l'algorithme renvoie la meilleure solution trouvée.

5.3 Résultats

5.3.1 Comparaison avec des ensembles d'instances résolues

La *learning heuristic* précédente a été appliquée aux ensembles d'instance A, B et P (disponibles sur le site [11]. Pour ces instances toutes les distances sont arrondies à l'entier le plus proche.

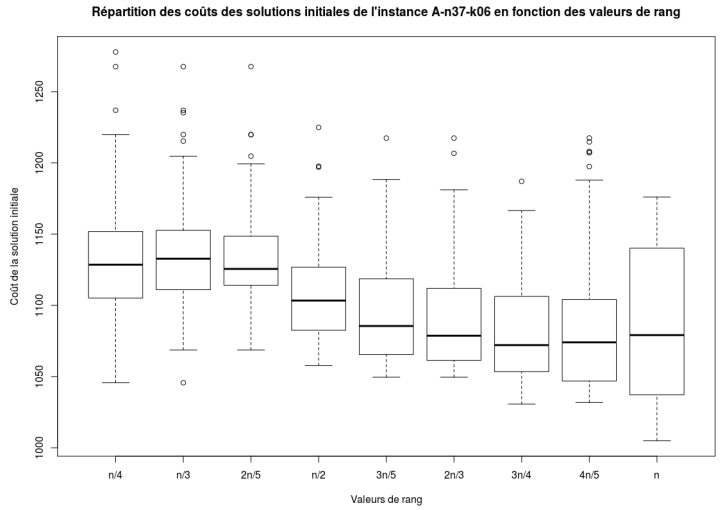

FIGURE 25 – Influence du Rang lors de l'apprentissage sur l'instance A-n37-k06.

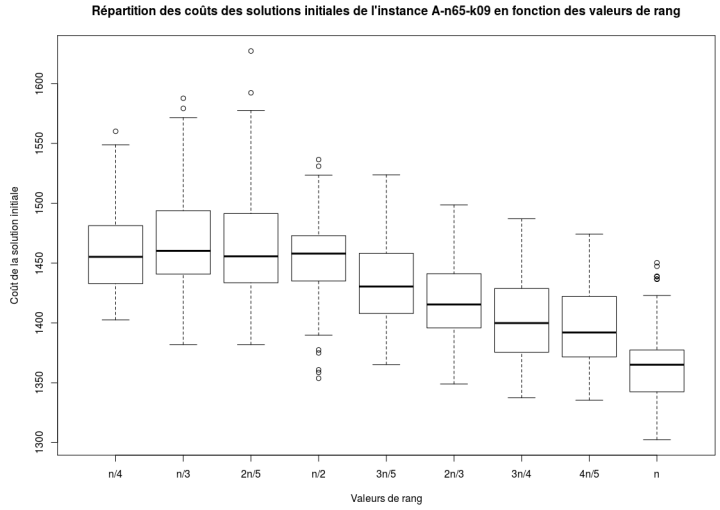

FIGURE 26 – Influence du Rang lors de l'apprentissage sur l'instance A-n65-k09.

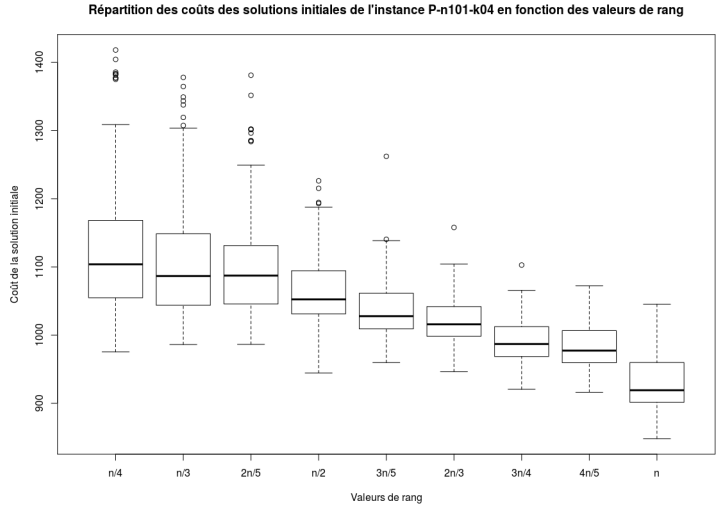

FIGURE 27 – Influence du Rang lors de l'apprentissage sur l'instance P-n101-k04.

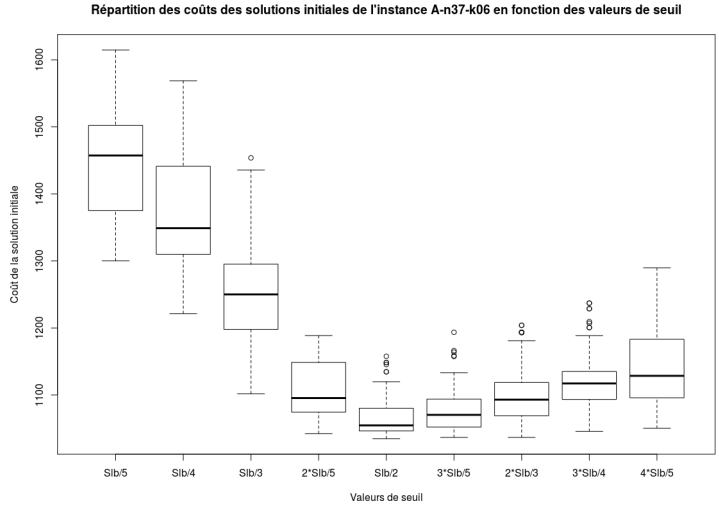

FIGURE 28 – Influence du Seuil lors de l'apprentissage sur l'instance A-n37-k06.

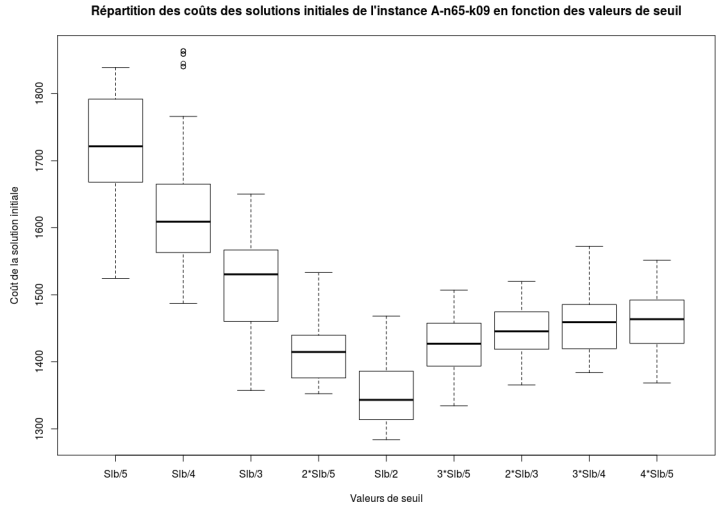

FIGURE 29 – Influence du Seuil lors de l'apprentissage sur l'instance A-n65-k09.

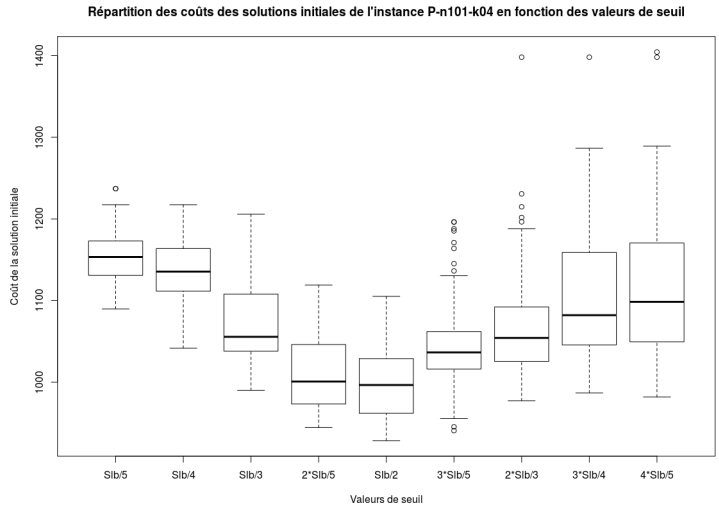

FIGURE 30 – Influence du Seuil lors de l'apprentissage sur l'instance P-n101-k04.

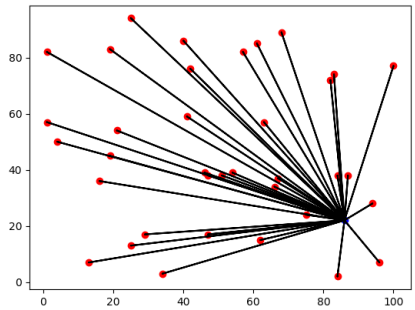

FIGURE 31 – Initialisation habituelle de CW.

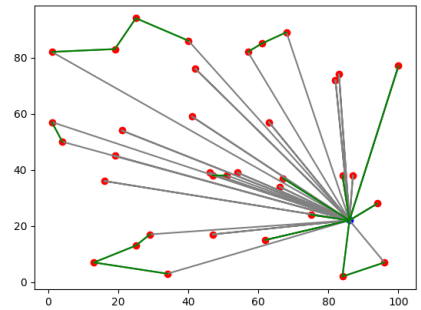

FIGURE 32 – Initialisation de CW après apprentissage : *Init*.

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

$(\lambda^*, \mu^*, \nu^*), Init \leftarrow Apprentissage()$
$cpt \leftarrow 0$
$allSolutions \leftarrow \{\}$
for $i \leftarrow 1$ ***to*** $NbIterations$ **do**
  $BaseSolution \leftarrow H_c(Init, I, D, \lambda^*, \mu^*, \nu^*)$
  $allSolutions \leftarrow allSolutions \cup BaseSolution$
  $crit \leftarrow upBound - cpt/10$
  $cpt \leftarrow cpt + 1$
  if $crit = lowBound$ **then**
    $cpt \leftarrow 0$
  Déterminer un nouvel *Init* en apprenant à partir de *BaseSolution* et du critère $rang = n * crit$
Trier *allSolutions*
return $allSolutions[0]$

Pour effectuer la phase d'apprentissage (décrite dans la section 4), nous choisissons de prendre un échantillon de taille 50, nous utilisons la base $Qual_{10}$, et le critère $rang = 0.7$. Nous choisissons ensuite les valeurs suivantes dans les algorithmes utilisés : $NbIterations = 25$, $limitTime = n/4$, $restartTime = n/40$, $resetTime = n/100$.

Les résultats obtenus sont disponibles sur les tableaux 4 à 6, la colonne *Best-Known* rappelle le coût de la meilleure solution connue pour l'instance considérée. Les résultats fournis sont ceux obtenus lors de l'exécution de l'algorithme considéré. *Best* renvoie la meilleure solution sur les 25% obtenues, $Mean_{10}$ renvoie la moyenne des 10 meilleures solutions, *Gap* correspond au pourcentage de différence entre la *Best* et la *Best-Known*, *Time* donne le temps d'exécution de l'algorithme.

La plupart des solutions obtenus avec l'algorithme LearnHeuristic sont très proches des solutions optimales, comme le montrent les résultats moyens présentés dans le tableau 7. Il est possible qu'en changeant la méthode d'apprentissage, ou le temps disponible dans H_c, on obtienne de meilleurs résultats. Néanmoins les paramètres choisis, permettent d'avoir un

TABLE 4 – Résultats pour l'ensemble A

Instance	Best-Known	LearnHeuristic			
		Best	$Mean_{10}$	Gap (%)	Time (s)
A-n32-k05	784	784	785	0	351
A-n33-k05	661	661	661	0	381
A-n33-k06	742	742	742	0	412
A-n34-k05	778	779	779	0.13	331
A-n36-k05	799	799	799	0	464
A-n37-k05	669	670	671	0.15	421
A-n37-k06	949	949	949	0	465
A-n38-k05	730	730	730	0	361
A-n39-k05	822	822	822	0	405
A-n39-k06	831	831	831	0	382
A-n44-k06	937	937	939	0	500
A-n45-k06	944	950	958	0.63	487
A-n45-k07	1146	1149	1149	0.26	500
A-n46-k07	914	914	916	0	498
A-n48-k07	1073	1073	1073	0	508
A-n53-k07	1010	1014	1017	0.39	515
A-n54-k07	1167	1167	1169	0	521
A-n55-k09	1073	1073	1073	0	506
A-n60-k09	1354	1354	1356	0	547
A-n61-k09	1034	1035	1042	0.09	562
A-n62-k08	1288	1308	1310	1.53	624
A-n63-k09	1616	1627	1636	0.68	641
A-n63-k10	1314	1320	1320	0.45	662
A-n64-k09	1401	1416	1419	1.06	684
A-n65-k09	1174	1176	1180	0.17	704
A-n69-k09	1159	1164	1167	0.43	768
A-n80-k10	1763	1767	1773	0.23	843

TABLE 5 – Résultats pour l'ensemble B

Instance	Best-Known	LearnHeuristic			
		Best	$Mean_{10}$	Gap (%)	Time (s)
B-n31-k05	672	672	672	0	319
B-n34-k05	788	788	788	0	371
B-n35-k05	955	955	957	0	388
B-n38-k06	805	806	806	0.12	413
B-n39-k05	549	549	550	0	436
B-n41-k06	829	831	831	0.24	491
B-n43-k06	742	742	744	0	534
B-n44-k07	909	910	910	0.11	463
B-n45-k05	751	751	751	0	461
B-n50-k07	741	741	741	0	604
B-n50-k08	1312	1320	1327	0.61	587
B-n52-k07	747	747	748	0	562
B-n56-k07	707	710	711	0.42	473
B-n57-k09	1598	1599	1601	0.06	523
B-n63-k10	1496	1533	1537	2.41	766
B-n64-k09	861	865	868	0.46	787
B-n66-k09	1316	1321	1324	0.38	770
B-n67-k10	1032	1040	1041	0.77	784
B-n68-k09	1272	1289	1290	1.32	666
B-n78-k10	1221	1231	1238	0.57	881

TABLE 6 – Résultats pour l'ensemble P

Instance	Best-Known	LearnHeuristic			
		Best	$Mean_{10}$	Gap (%)	Time (s)
P-n016-k08	450	450	450	0	132
P-n019-k02	212	212	215	0	163
P-n020-k02	216	216	216	0	203
P-n021-k02	211	211	211	0	168
P-n022-k02	216	216	216	0	227
P-n023-k08	529	529	530	0	205
P-n040-k05	458	458	458	0	383
P-n045-k05	510	510	510	0	580
P-n050-k07	554	554	555	0	596
P-n050-k08	631	631	633	0	485
P-n050-k10	696	699	700	0.43	584
P-n051-k10	741	741	741	0	479
P-n055-k07	568	568	572	0	641
P-n055-k10	694	694	696	0	632
P-n060-k10	744	750	751	0.8	513
P-n060-k15	968	975	978	0.72	596
P-n065-k10	792	792	792	0	668
P-n070-k10	827	839	840	1.43	1120
P-n076-k04	593	596	600	0.50	863
P-n076-k05	627	630	634	0.48	691
P-n101-k04	681	686	687	0.73	1114

algorithme à la fois rapide et efficace.

TABLE 7 – Synthèse des résultats pour les ensembles A, B et P

Instances	Size	Gap (%)	Time (s)
Set A	27	0.23	14043
Set B	20	0.37	11279
Set P	21	0.24	6141

5.3.2 Résultats obtenus sur les problèmes plus difficiles

Dans cette section sont présentés les résultats obtenus sur l'ensemble d'instances *Golden* [11]. Ces instances ont la particularité d'avoir un grand nombre de clients à desservir (de 200 à 483). Peu de solutions exactes ont été trouvées pour ces instances, de ce fait il est toujours possible d'améliorer les solutions existantes.

TABLE 8 – Résultats pour l'ensemble *Golden*

Instance	Nb customers	Best-Known	Optimal	LearnHeuristic			
				Best	Mean_{10}	Gap (%)	Time (s)
Golden-01	240	5623.47	no	5499	5531.28	- 2.25	
Golden-02	320	8404.61	no	8441.46	8494.86	0.44	
Golden-03	400	11036.2	no	11140.33	11230.34	0.93	
Golden-04	480	13590.0	no	13729.76	13741.65	1.02	
Golden-05	200	6460.98	no	6495.13	6506.04	0.53	
Golden-06	280	8412.9	no	8556.70	8592.90	1.68	
Golden-07	360	10102.7	no	10340.80	10361.58	2.30	
Golden-08	440	11635.3	no	11899.97	11956.68	2.22	
Golden-09	255	579.71	no	606.18	610 03	4.37	
Golden-10	323	735.66	no	772.69	773.91	4.79	
Golden-11	399	912.03	no	963.38	966.40	5.33	
Golden-12	483	1101.5	no	1160.72	1177.08	5.10	
Golden-13	252	857.19	yes	886.53	895.55	3.31	
Golden-14	320	1080.55	yes	1147.72	1148.12	5.85	
Golden-15	396	1337.87	no	1430.69	1431.89	6.49	
Golden-16	480	1611.56	no	1749.97	1752.71	7.90	
Golden-17	240	707.76	yes	751.02	751.62	5.76	
Golden-18	300	995.13	yes	1071.55	1072.70	7.13	
Golden-19	360	1365.6	yes	1470.84	1472.22	7.14	
Golden-20	420	1817.59	yes	1982.82	1986.61	8.51	

Pour exécuter l'algorithme nous reprenons en partie les valeurs précédentes, mais nous changeons le nombre d'arêtes conservées lors du restart : on garde à présent 95% des arêtes. Pour le relâchement de la contrainte, on prend $upBound = 0.9$ et $lowBound = 0.7$. Les instances étant plus grandes, il faut conserver plus d'arêtes pour que l'apprentissage soit plus efficace.

Les résultats obtenus sont présentés dans le tableau 8. Nous parvenons à obtenir une nouvelle meilleure solution pour l'instance *Golden-01*, toutefois c'est le seul résultat remarquable. Il est possible que pour les autres instances les solutions actuelles soient déjà très bonnes. Un autre choix de paramètres pourrait aussi donner de meilleurs résultats. La figure 33 montre la nouvelle meilleure solution obtenue pour l'instance *Golden-01*.

5.4 Limites et perspectives

Nous avons vu que l'algorithme proposé était efficace pour résoudre les petites instances (inférieures à 100 clients). Il a aussi permis de trouver une nouvelle solution à l'instance *Golden-01*. Néanmoins les autres résultats obtenus ne sont pas aussi bons. Le temps des exécutions pour les plus grandes instances est long, ce qui ne permet pas de tester de nombreuses valeurs de paramètres. De plus le premier apprentissage influence fortement les résultats obtenus ensuite. Il va notamment donner les paramètres $(\lambda^*, \mu^*, \nu^*)$, utilisées pendant toute la durée de l'exécution. Une amélioration possible pourrait donc être de pré-calculer le meilleur $(\lambda^*, \mu^*, \nu^*)$, pour le choisir lors de l'exécution, mais cela impose de refaire ce calcul pour chaque instance.

L'un des avantages de l'algorithme proposé est que l'on peut remplacer l'algorithme d'optimisation H_c par celui que l'on veut. Ainsi, nous pourrions privilégier un algorithme efficace sur certains types d'instances afin d'obtenir de meilleurs résultats. De plus, l'algorithme A&S peut facilement être adapté à d'autres extensions du *VRP* [3]. Ce qui rend notre algorithme d'optimisation particulièrement intéressant.

Conclusion

A travers cet article, nous nous sommes intéressés dans un premier temps à un algorithme permettant d'obtenir une solution initiale de bonne qualité. Puis nous avons proposé un algorithme d'optimisation, à partir de l'heuristique proposé par Arnold et Sörensen [3]. Nous avons ensuite cherché à extraire de la connaissance des solutions initiales obtenues, pour pouvoir finalement l'intégrer dans l'algorithme d'optimisation retenu. L'algorithme résultant de ce travail, est la *Learning Heuristic* décrite dans l'algorithme 7.

Cette heuristique a donné de très bons résultats sur les petits ensembles d'instances A, B et P. Les résultats obtenus ensuite sur l'ensemble *Golden* étaient moins bons, bien que nous soyons parvenus à trouver une nouvelle meilleure solution à l'instance *Golden-01*.

Bien que l'heuristique puisse être utilisée avec n'importe quel algorithme d'optimisation, celui retenu a l'avantage d'être facilement adaptable à d'autres problèmes de tournées de véhicules (comme le problème multi-dépôts [3]). L'un des principaux inconvénients de cette heuristique reste toutefois son temps d'exécution.

Pour améliorer notre *Learning Heuristic*, il faudrait apprendre différemment ou de manière plus efficace, en s'aidant notamment de la meilleure solution courante obtenue pendant l'exécution.

Remerciements

Je tiens à remercier l'équipe ORKAD, du centre CRIStAL de Lille, qui m'a aidé à réaliser ce projet. Je remercie plus particulièrement Laetitia Jourdan (laetitia.jourdan@univ-lille.fr), Marie-Éléonore Kessaci (me.kessaci@univ-lille.fr), membres de l'équipe ORKAD, ainsi que Diego Cattaruzza (diego.cattaruzza@ec-lille.fr) membre de l'équipe INOCS, qui m'ont encadré pendant la durée de ce projet.

Références

[1] Le probleme de tournees de vehicules : etude et resolution approchee. [Rapportde recherche] RR-2197, INRIA., 1994.

[2] 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.

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

[4] G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 1964.

[5] T. J. GASKELL. Bases for vehicle fleet scheduling. *Operational Research Quarterly*, 18 :281–295, September 1967.

[6] Marie-Eleonore Marmion. *Recherche locale et optimisation combinatoire : de l'analyse structurelle d'un problème à la conception d'algorithmes efficaces*. PhD thesis, Université des Sciences et Technologie de Lille - Lille I, 2011, Mars 2012.

[7] Silvano Martello and Paolo Toth. *Knapsack problems : algorithms and computer implementations*. John Wiley and Sons, New York, USA, 1990.

[8] H. Paessens. The savings algorithm for the vehicle routing problem. *European Journal of Operational Research*, 34 :336–344, March 1988.

[9] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization : algorithms and complexity*. Dover Publications, Upper Saddle River, NJ, USA, 1982.

[10] François Legras Sahbi Ben Ismail and Gilles Coppin. Synthèse du problème de routage de véhicules. `https://portail.telecom-bretagne.eu/publi/public/fic_download.jsp?id=5745`, 2011.

[11] Ivan Xavier. Cvrplib. `http://vrp.atd-lab.inf.puc-rio.br/index.php/en/`, 2014.

[12] P. Yellow. A computational modification to the savings method of vehicle scheduling. *Operational Research Quarterly*, 21 :281–283, 1970.

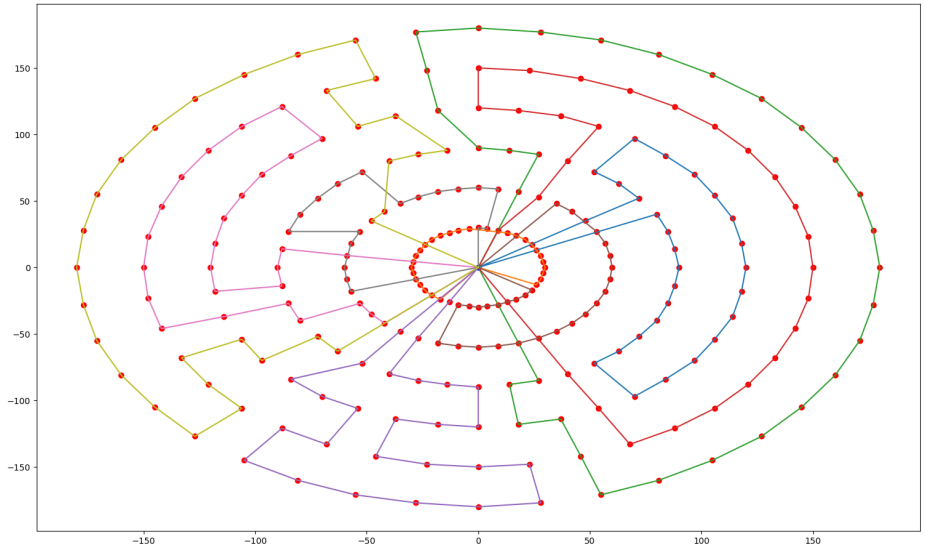

FIGURE 33 – Nouvelle *Best-Known* pour l'instance *Golden-01*.