UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Collaboration de solveurs du problème de tournées de véhicules

Mémoire présenté par
Louis BAUDOUX

en vue de l'obtention du grade de Master
Ingénieur civil en informatique

Promoteur(s)
Yves DEVILLE

Lecteur(s)
Pierre SCHAUS, Michael SAINT-GUILLAIN

Année académique 2017-2018

Remerciements

Tout d'abord, je voudrais remercier mon promoteur Yves Deville, pour sa supervision, ses conseils et la qualité de son suivi tout au long de l'année.

Je voudrais aussi remercier Michael Saint-Guillain, pour sa disponibilité, ses conseils et l'aide qu'il m'a apporté pour mieux comprendre et améliorer mes algorithmes. Je le remercie également pour les idées qu'il a apportées pour m'aiguiller lors de la réalisation de ce mémoire.

Finalement, je voudrais remercier mes parents, mon frère et ma sœur pour leur relecture, leurs conseils et l'intérêt qu'ils ont porté à ce mémoire.

Table des matières

Chapitre 1

Introduction

Le problème de tournées de véhicules est un problème qui a été défini en 1959 par George Dantzig et John Ramser [18]. Le problème consiste à optimiser les itinéraires d'un ensemble de véhicules afin de desservir un ensemble de clients. C'est un problème qui est depuis très étudié, d'une part parce que ses applications sont nombreuses et d'autre part par sa grande difficulté.

L'apparition de ce problème dans des situations de la vie réelle est très fréquente, c'est pourquoi de nombreuses variantes du problème ont été créées et étudiées afin de modéliser le plus fidèlement possible ces situations [36]. On notera par exemple des variantes dans lesquelles les véhicules transportent des biens à livrer et ont une capacité maximale autorisée. Dans d'autres cas, les biens devront être d'abord récupérés avant de pouvoir être livrés. Bien souvent, les situations réelles sont modélisées par une combinaison de ces variantes, ce qui donne lieu à un très grand nombre de problèmes légèrement différents pour lesquels des algorithmes spécifiques ont été développés.

Il existe beaucoup d'approches différentes pour résoudre ce problème et ses très nombreuses variantes. Certaines de ces approches se basent sur la modification de solutions déjà existantes, comme dans le cas de la recherche locale [50], [24], [33], [26]. D'autres se basent sur des comportements animaliers, tels que celui d'une colonie de fourmis [32], [9], [7], [27], [23]. Et enfin, d'autres utilisent une représentation mathématique du problème pour le résoudre [38], [42], [35].

L'objectif de ce mémoire est d'analyser dans quelle mesure ces différentes méthodes peuvent collaborer entre elles afin de trouver la solution d'un problème de tournées de véhicules plus rapidement. L'accent sera donc mis sur la façon dont les performances peuvent être influencées par cette collaboration plutôt que sur les performances du modèle en lui-même. Un framework de collaboration sera proposé. Celui-ci inclura un serveur gérant la connexion entre les solveurs, deux implémentations de solveurs différents adaptés pour la collaboration et un programme client permettant de gérer quelle instance du problème doit être résolu. Des expériences seront ensuite menées en utilisant ces solveurs, avec et sans collaboration, afin de mesurer l'impact sur leurs performances.

Ce mémoire est organisé comme suit :

Le chapitre 2 fera une description détaillée du problème de tournées de véhicules, de ses variantes et des problèmes qui lui sont liés.

Le chapitre 3 fera un état de l'art des différentes méthodes qui ont été développées pour résoudre ce problème.

Le chapitre 4 décrira le solveur par optimisation par colonie de fourmis qui a été développé pour ce travail. Cette méthode basée sur le comportement collectif des fourmis y sera expliquée et celle-ci sera utilisée lors de la collaboration entre solveurs.

Le chapitre 5 décrira le solveur par optimisation linéaire en nombres entiers, il s'agit également d'une méthode de résolution de ce problème et sera aussi utilisée lors de la collaboration entre solveurs.

Le chapitre 6 expliquera les différentes approches possibles pour faire collaborer des solveurs entre eux et détaillera l'approche utilisée dans ce travail.

Le chapitre 7 sera consacré aux expériences, on y montrera les performances des solveurs et l'impact de la collaboration entre solveurs.

Chapitre 2

Problème de tournées de véhicules

2.1 Description et notations

Le problème de tournées de véhicules consiste à établir les itinéraires pour un ensemble de véhicules afin de visiter un ensemble de clients. Chaque véhicule démarre d'un même endroit et doit y retourner après sa tournée, on nomme généralement cet endroit le dépôt [18]. On réfère souvent à ce problème par son sigle anglais, VRP (Vehicle Routing Problem).

Une configuration valide des routes de chaque véhicule doit respecter les conditions suivantes :

— tous les clients sont desservis
— chaque véhicule part du dépôt, visite un ou plusieurs clients et retourne au dépôt
— chaque client est desservi par un et un seul véhicule
— un véhicule ne visite jamais deux fois le même client.

Les applications réelles de ce problème dans l'industrie sont nombreuses, voici une liste non-exhaustive de quelques-unes de ces applications :

— créations d'itinéraire pour des véhicules de livraison
— distribution de presse ou de publicité
— maintenances et réparations d'équipements (chauffage, électricité, informatique, etc)
— intervention de contrôle ou d'expertise chez les particuliers.

Introduisons maintenant quelques termes et notations utiles pour décrire le problème [38].

Les clients, le dépôt et les routes forment un **graphe** non-dirigé $G = (V, E)$ où V est l'ensemble des nœuds et E est l'ensemble des arêtes, ces notions ainsi que d'autres notations utiles sont expliquées ci-dessous.

Un **nœud** est soit un client, soit le dépôt. On utilise ce terme lorsqu'on ne veut pas faire la distinction entre un client et le dépôt. Par exemple, on définit plus loin le coût des arêtes, celui-ci est défini de la même manière, qu'il s'agisse du coût de l'arête entre deux clients ou entre un client et le dépôt, on utilisera donc la notion de nœud dans ce cas. On associe à chaque nœud un indice, si il y n clients dans le problème, on utilisera l'indice 0 pour le dépôt et les indices 1 à n pour les clients.

Une **arête** est la section de route reliant deux nœuds, on note (i, j) une arête entre les nœuds d'indice i et j. On peut alors définir E comme étant : $E = \{(i, j) | i, j \in V, i > j\}$

Le **coût** d'une arête est le poids associé à cette arête, plus le poids est élevé, plus il est coûteux d'emprunter cette section de route. On note c_e ou c_{ij} le coût de l'arête $e = (i, j)$. Le coût peut par exemple être défini comme la longueur du plus court chemin ou le temps minimum nécessaire pour aller d'un nœud à l'autre. De manière plus générale, le coût peut représenter n'importe quelle quantité tant qu'elle est définie entre chaque paire de nœuds du problème. Parfois, il se peut qu'il soit impossible d'atteindre un nœud depuis un autre, pour modéliser cela, on peut affecter un coût infini à l'arête correspondante. On considère dans ce travail que les coûts sont symétriques, c'est-à-dire que $c_{ij} = c_{ji}, \forall i, j \in V$. Ceci signifie donc que le sens dans lequel circule le véhicule n'a pas d'importance. On fait également la supposition que le coût des arêtes respecte

l'inégalité triangulaire : $c_{ij} \leq c_{ik} + c_{kj}, \forall i, j, k \in V$.

On note K le nombre de véhicules à utiliser.

Pour un ensemble de nœuds $S \subseteq V$, on note $\delta(S)$ l'ensemble des arêtes $e \in E$ qui ont une seule extrémité dans S. Si on considère un seul nœud $i \in V$ plutôt qu'un ensemble de nœuds, on note $\delta(i)$ plutôt que $\delta(\{i\})$.

On appelle **instance** une réalisation concrète d'un problème. Dans le cadre du VRP, ceci se traduit par l'ensemble des données nécessaires pour caractériser complètement le problème :

— le coût des arêtes entre chaque paire de nœuds. On donne parfois la position géographique des nœuds et on définit alors le coût d'une arête par la distance entre ses extrémités. On utilise souvent la distance euclidienne

— K, le nombre de véhicules à utiliser

— des données spécifiques à la (aux) variante(s) du VRP considérée(s) (on verra dans la section 2.3 les variantes les plus courantes).

On appelle **solution** une configuration valide des routes des véhicules. Parmi l'ensemble des solutions possibles, seule une solution (parfois plusieurs, mais de coût total égal) est la solution dite "optimale", celle qui minimise le coût total. On notera donc que quand on parle de solution, on ne désigne pas nécessairement la solution optimale du problème.

On appelle **solveur** le programme informatique permettant de résoudre automatiquement une instance d'un problème, on verra dans ce travail deux solveurs de VRP différents, présentés aux chapitres 4 et 5.

Voyons un exemple de solution d'une instance du VRP :

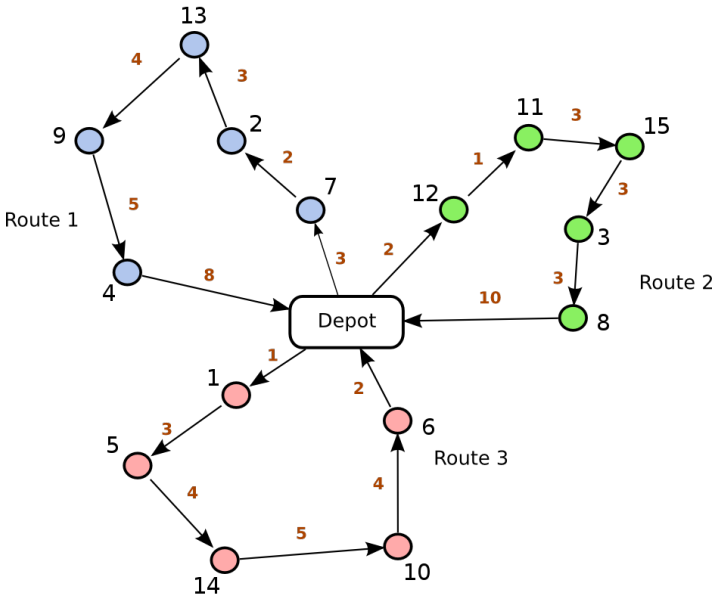


FIGURE 2.1 – Exemple de solution du problème de tournées de véhicules. Source : [49].

Dans l'exemple ci-dessus, chaque couleur représente un véhicule, on a donc ici une solution à trois véhicules et le but est de minimiser le coût total des trajets. Le coût est indiqué ici en rouge sur chacune des arêtes reliant les nœuds.

On peut représenter cette solution par :

$[7, 2, 13, 9, 4]$

$[12, 11, 15, 3, 8]$

$[1, 5, 14, 10, 6]$

Chaque ligne représente une des routes, la première ligne indique par exemple le trajet de la route 1 en indiquant les clients desservis et l'ordre dans lequel ils sont visités.

Le coût associé à une solution est simplement la somme des coûts des arêtes qui la composent, on a ici un coût de 25 pour la route 1, 22 pour la route 2 et 19 pour la route 3. Le coût total est donc de 66.

2.2 Définition mathématique

Dans cette section, on va décrire mathématiquement le problème par un ensemble de variables et de contraintes sur ces variables. Cette définition est importante car elle permet de décrire formellement les conditions à remplir pour qu'une solution soit valide.

On peut modéliser mathématiquement le VRP sous forme d'une formulation de flots à deux index [38]. Dans cette formulation, on utilise une variable x_e pour chaque arête $e \in E$ qui indique combien de fois elle fait partie de la solution. 0 indique que l'arête n'est empruntée par aucun véhicule, 1 signifie qu'un véhicule emprunte cette arête une fois et 2 indique que le véhicule fait un aller-retour, ceci n'est possible que si le véhicule part du dépôt, dessert un client et retourne directement au dépôt. Seules les variables x_e connectant un client et le dépôt peuvent donc prendre cette valeur de 2.

En reprenant l'exemple de la figure 2.1, les variables représentant cette solution sont (elles sont notées ici x_{ij} plutôt que x_e pour voir facilement de quelle arête il s'agit) :

$x_{0,7} = x_{7,2} = x_{2,13} = x_{13,9} = x_{9,4} = x_{4,0} = 1 \rightarrow$ Route 1

$x_{0,1} = x_{1,5} = x_{5,14} = x_{14,10} = x_{10,6} = x_{6,0} = 1 \rightarrow$ Route 2

$x_{0,12} = x_{12,11} = x_{11,15} = x_{15,3} = x_{3,8} = x_{8,0} = 1 \rightarrow$ Route 3

Toutes les autres variables associées aux arêtes qui ne sont pas représentées sur la figure valent 0 car aucun véhicule n'emprunte ces arêtes. Il n'y a ici aucune variable valant 2, car il n'y a pas de véhicule faisant un aller-retour pour un client.

Le modèle est défini par ces équations :

$$\min \sum_{e \in E} c_e x_e \tag{2.1}$$

tel que

$$\sum_{e \in \delta(i)} x_e = 2 \qquad \forall i \in V \setminus \{0\}, \tag{2.2}$$

$$\sum_{e \in \delta(0)} x_e = 2K, \tag{2.3}$$

$$\sum_{e \in S} x_e \leq |S| - 1 \qquad \forall S \subseteq E \setminus \delta(0), S \neq \emptyset \tag{2.4}$$

$$x_e \in \{0, 1\} \qquad \forall e \notin \delta(0) \tag{2.5}$$

$$x_e \in \{0, 1, 2\} \qquad \forall e \in \delta(0) \tag{2.6}$$

L'équation 2.1 représente la minimisation du coût total des routes.

La contrainte 2.2 implique que pour chaque client, un véhicule arrive et sort de ce nœud.

La contrainte 2.3 implique que pour le dépôt, les K véhicules utilisés sortent et retournent au dépôt.

La contrainte 2.4 empêche qu'un véhicule effectue une tournée ne passant pas par le dépôt. En effet, si pour tout sous-ensemble S d'arêtes non-incidentes au dépôt, on autorise seulement $|S| - 1$ d'entre elles à être utilisées, il est impossible de former un cycle ne contenant pas le dépôt. On notera que cette formulation donne lieu en réalité à un nombre exponentiel de contraintes,

puisqu'il y en a une pour chaque sous-ensemble $S \subseteq E \setminus \delta(0)$ et qu'il existe 2^n sous-ensembles d'un ensemble de taille n.

La contrainte 2.5 signifie que les arêtes reliant un client à un autre client ne peuvent être empruntées qu'une seule fois maximum.

La contrainte 2.6 signifie que les arêtes reliant un client au dépôt peuvent être empruntées zéro, une, ou deux fois (dans le cas d'un aller-retour).

Il existe d'autres formulations pour ce problème étant parfois plus adaptées pour résoudre certaines variantes du VRP. Seule cette formulation-ci est montrée ici car c'est sur celle-ci que se basera le solveur présenté au chapitre 5 (voir section 1.3 de [38] pour une description détaillée des autres formulations).

2.3 Variantes

Ce problème possède de nombreuses variantes en ajoutant par exemple des contraintes sur les véhicules ou sur les clients. On peut également combiner ces contraintes entre elles car elles sont généralement non-exclusives.

Ces variantes ont généralement été introduites pour modéliser des contraintes présentes dans les applications réelles du VRP.

Pour plus de détails sur les techniques spécifiques aux variantes présentées ici, la référence [28] fait un état de l'art et une classification de la littérature sur le VRP.

2.3.1 Contrainte de capacité sur les véhicules

Cette variante est la plus courante (environ 90% de la recherche actuelle considère cette contrainte [28]), c'est cette variante qui sera étudiée tout au long de ce travail.

Elle consiste à imposer une capacité maximale aux véhicules. On associe alors à chaque client une demande d_i et chaque véhicule doit faire en sorte que la somme des demandes des clients qu'il dessert n'excède pas sa capacité. Formellement, si C est la capacité maximale des véhicules, $R \subseteq V \setminus \{0\}$ est l'ensemble des clients desservi sur la route d'un véhicule, on doit avoir :

$$\sum_{i \in R} d_i \leq C$$

Dans cette variante, on peut ne pas préciser à l'avance le nombre de véhicules à utiliser, puisque celui-ci peut être déterminé à partir de la capacité maximale des véhicules et des demandes de chaque client. Il s'agit cependant d'un problème difficile (problème de *bin packing*) nécessitant lui aussi des algorithmes dédiés. Il est expliqué plus en détails dans la sous-section 2.4.2.

Dans la littérature, on réfère à cette variante par CVRP (pour Capacitated Vehicle Routing Problem).

On peut tenir compte de cette contrainte dans le modèle mathématique présenté à la sous-section 2.2 en ajoutant l'équation suivante [38] :

$$\sum_{e \in \delta(S)} x_e \geq 2r(S) \qquad \forall S \subseteq V \setminus \{0\}, S \neq \emptyset \tag{2.7}$$

$r(S)$ est le nombre minimum de véhicule nécessaire pour desservir les clients dans S. Par exemple, si S contient trois clients ayant une demande de 40, 50 et 80 et que la capacité des véhicules vaut 100, $r(S)$ vaut 2 car il faut au minimum deux véhicules pour servir ces clients sans faire excéder la capacité d'un des véhicules.

L'effet de cette contrainte est double. Elle impose d'une part que les véhicules respectent leur contrainte de capacité car elle assure que pour chaque sous-ensemble de clients, il y ait au moins le nombre minimum requis de véhicules pour les desservir. D'autre part, elle impose que les véhicules passent par le dépôt lors de leur tournée car si un cycle ne contenant pas le dépôt

est formé, la contrainte associée à ce sous-ensemble de clients sera violée puisque le nombre d'arcs entrants, donné par $\sum_{e \in \delta(S)} x_e$, vaudra alors 0. L'addition de cette contrainte signifie donc que la contrainte 2.4 éliminant les sous-tours ne passant pas par le dépôt peut être retirée du modèle.

Pour la même raison que la contrainte 2.4 (voir l'explication à la section précédente), on a en réalité un nombre exponentiel de contraintes.

2.3.2 Contrainte de fenêtre de temps

Dans cette variante, chaque client doit être desservi dans une plage horaire définie. Les véhicules peuvent éventuellement arriver en avance, mais devront alors patienter. Si un véhicule arrive en retard, la contrainte est violée et la solution est soit considérée comme invalide soit pénalisée par un coût supplémentaire, par exemple. On associe parfois un temps de service à chaque client correspondant au temps nécessaire pour servir ce client.

Cette variante est également très courante car elle modélise les cas concrets où les clients doivent être servis lorsqu'ils sont disponibles (livraison de bien à domicile, maintenance, etc).

Dans la littérature, on réfère à cette variante par VRPTW (pour Vehicle Routing Problem with Time Windows).

Parmi les méthodes utilisées pour résoudre le VRPTW, on trouve :

— l'approche *branch-and-price* [34]
— la recherche locale [22]
— les algorithmes génétiques [30].

2.3.3 Véhicules hétérogènes

Dans cette variante, on considère que les véhicules à disposition n'ont pas tous les mêmes caractéristiques, il peut notamment y avoir :

— des capacités différentes (si considéré avec CVRP)
— des coûts fixes et/ou variables différents
— un nombre limité de véhicules de chaque type disponible
— des contraintes de temps spécifique au véhicule (si considéré avec VRPTW).

Dans la littérature, on réfère à cette variante par HFVRP ou HVRP (pour Heterogeneous (Fleet) Vehicle Routing Problem).

Dans [2], une recherche locale avec voisinage variable est utilisée pour résoudre approximativement le problème. Dans [14], une recherche locale tabou est utilisée, la contrainte de fenêtre de temps est considérée en plus des véhicules hétérogènes.

2.3.4 Dépôts multiples

Dans cette variante, plusieurs dépôts peuvent servir de station de départ et d'arrivée des véhicules. On distingue deux sous-variantes : celle où les véhicules partent et arrivent d'un même dépôt et celle où ils ont la possibilité d'arriver à un dépôt différent de celui de départ. Cette variante est également beaucoup étudiée car, en pratique, une même société aura souvent plusieurs sites d'où peuvent partir les véhicules de livraison ou maintenance.

Dans la littérature, on réfère à cette variante par MDVRP (pour Multiple Depot Vehicle Routing Problem).

Dans [3], deux approches sont proposées. Dans la première, la résolution se fait en deux temps, d'abord l'assignation des véhicules à un dépôt, puis la création des itinéraires. Dans la seconde, ces deux étapes sont effectuées en même temps. La création des itinéraires est faite par une recherche locale tabou.

Dans [4], une méthode de résolution exacte basée sur la programmation entière est utilisée.

Une méthode hybride utilisant l'optimisation par colonie de fourmis et la recherche locale est proposée dans [12] pour résoudre le MDVRP avec contrainte de fenêtre de temps.

2.3.5 Collecte et livraison

Dans cette variante, chaque client est soit un point de collecte soit un point de livraison. Elle est nécessairement associée avec une contrainte de capacité sur les véhicules, telle que présentée dans la première variante, autrement elle reviendrait à un VRP classique. Il existe deux sous-variantes :

— les biens collectés sont ceux qui doivent être livrés
— les biens collectés sont différents de ceux qui doivent être livrés, il n'y a donc pas d'échange entre les clients.

Dans la littérature, on réfère à cette variante par VRPPD (pour Vehicle Routing Problem with Pick-up and Delivery).

Une approche hybride basée sur un algorithme génétique et une heuristique constructive est proposée dans [47] pour résoudre le VRPPD avec contrainte de fenêtre de temps.

Une recherche locale à voisinage variable a été proposée dans [41] pour le VRPPD avec demandes divisées (la demande d'un client est traitée en plusieurs trajets).

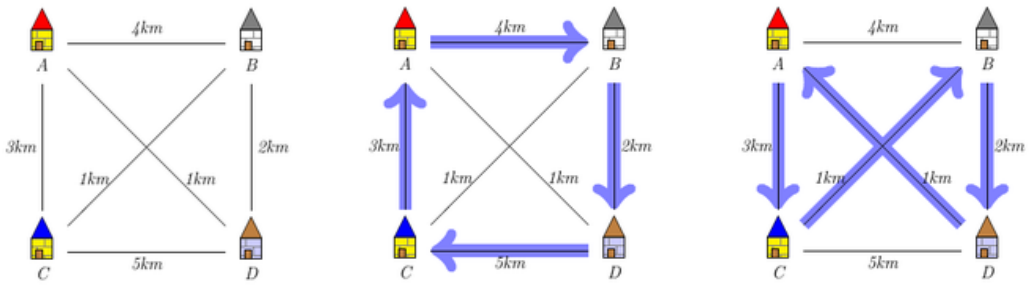
2.4 Problèmes liés au CVRP

Le problème du voyageur de commerce et le problème de *bin packing* peuvent être vus comme deux cas particuliers du CVRP (voir sous-section 2.3.1 pour un rappel de la définition). Nous allons voir ici plus en détails en quoi ils consistent.

2.4.1 Problème du voyageur de commerce

Le VRP est en fait une extension du problème du voyageur de commerce, lui aussi très étudié (appelé TSP en anglais, Traveling Salesman Problem). Dans le problème du voyageur de commerce, le but est de trouver le cycle de longueur minimale visitant un ensemble de villes. Le VRP est fortement lié à ce problème car, après avoir affecté à un véhicule les clients qu'il doit desservir, il faut trouver le cycle de longueur minimale passant par chacun de ces clients et le dépôt. On peut également remarquer que si il n'y a qu'un seul véhicule à disposition, cela revient à résoudre le TSP puisqu'il devra desservir à lui seul tout les clients.

Voyons un exemple d'instance du TSP :



(a) Définition de l'instance du TSP

(b) Solution possible ayant un coût total de 14 km

(c) Meilleure solution possible ayant un coût total de 7 km

FIGURE 2.2 – Exemple d'une instance du problème de voyageur de commerce. Source : [40].

Dans cet exemple, le but est donc de trouver le cycle parcourant la plus courte distance et visitant les villes A, B, C et D. Pour transformer ce problème en VRP, il suffit de prendre une des quatre villes comme dépôt et de considérer les villes restantes comme des clients que le véhicule doit desservir.

On peut formuler le problème mathématiquement de manière très similaire au VRP. On utilise les mêmes notations que pour le VRP. On associe à chaque arête $e \in E$ une variable

binaire x_e qui indique si l'arête appartient à la solution, elle vaut 1 si elle est empruntée et 0 sinon.

Le modèle est défini par ces équations :

$$\min \sum_{e \in E} c_e x_e \tag{2.8}$$

tel que

$$\sum_{e \in \delta(i)} x_e = 2 \qquad \forall i \in V, \tag{2.9}$$

$$\sum_{e \in S} x_e \leq |S| - 1 \qquad \forall S \subset E, S \neq \emptyset \tag{2.10}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E \tag{2.11}$$

L'équation 2.8 signifie qu'on cherche à minimiser le coût total du cycle.

La contrainte 2.9 implique que pour chaque ville, deux arêtes y soient connectées, une pour entrer et une pour sortir.

La contrainte 2.10 élimine les sous-tours. En effet, sans cette contrainte, on pourrait avoir une solution visitant toutes les villes et qui soit constituée de deux cycles, hors il doit n'y en avoir qu'un.

La contrainte 2.11 impose que les variables soient binaires.

Comme dit précédemment, le TSP est très étudié et a donc une littérature très riche, [16] fait un état de l'art des meilleures techniques connues pour résoudre ce problème.

2.4.2 Problème de *bin packing*

Le problème de *bin packing* est un problème lié spécifiquement au VRP où l'on considère la contrainte de capacité sur les véhicules (CVRP). Il consiste à minimiser le nombre de conteneurs à capacité limitée utilisés pour ranger des articles ayant un poids assigné. Dans le cas du CVRP, les conteneurs sont les véhicules et le poids des articles correspond à la demande des clients. Voyons un exemple de solution d'une instance du problème de *bin packing* :

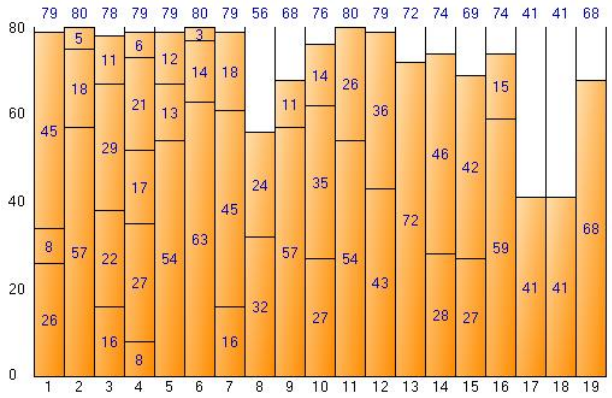


FIGURE 2.3 – Exemple de solution d'une instance du problème de *bin packing*. Source : [10].

Dans cet exemple, les conteneurs ont une capacité maximale de 80, chaque colonne verticale correspond à un conteneur et chaque rectangle orange est un article avec son poids assigné au centre. Cette solution utilise 19 conteneurs (la solution optimale en utilise 17).

Pour un problème ayant une liste d'articles $1, 2, ..., n$ de poids c_i et des conteneurs de capacité C, on peut formuler mathématiquement le problème en utilisant les variables suivantes :

— x_{ij} vaut 1 si l'article i est rangé dans le conteneur j
— y_j vaut 1 si le conteneur j est utilisé.
Le modèle est :

$$\min \sum_{j=1}^{n} y_j \tag{2.12}$$

$$\sum_{i=1}^{n} c_i x_{ij} \leq y_j C \qquad j = 1, ..., n \tag{2.13}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, ..., n \tag{2.14}$$

$$x_{ij} \in \{0, 1\} \qquad i = 1, ..., n, j = 1, ..., n \tag{2.15}$$

$$y_j \in \{0, 1\} \qquad j = 1, ..., n \tag{2.16}$$

L'équation 2.12 signifie qu'il faut minimiser le nombre de conteneurs utilisés.

La contrainte 2.13 impose qu'on ne peut pas excéder la capacité d'un conteneur. Elle impose également qu'un conteneur marqué comme vide (sa variable correspondante y_j vaut 0) soit bel et bien vide.

La contrainte 2.14 impose qu'un article doit être inclus dans un seul conteneur.

Les contraintes 2.15 et 2.16 imposent que les variables soient binaires.

Ce problème a beaucoup d'applications pratiques, telles que le rangement de fichiers sur un système informatique, le découpage de câbles ou encore la gestion de la cargaison de camions de livraison.

La référence [8] donne un bon aperçu des meilleures techniques pour résoudre ce problème.

2.5 Complexité

Le VRP est un problème très difficile à résoudre optimalement, cette difficulté et ses nombreuses applications font qu'il est un des problèmes combinatoires les plus intéressants et les plus étudiés. Un problème combinatoire est un problème où le but est de trouver la combinaison optimale dans un ensemble de combinaisons possibles. Dans le cas du VRP, une combinaison correspond à une solution, telle que définie au point 2.1.

Pour illustrer cette difficulté, prenons le VRP sous sa forme la plus simple où il n'y a qu'un seul véhicule qui dessert tous les clients (équivalent à un TSP donc). Pour un TSP, le nombre de solutions différentes possibles est donné par $\frac{1}{2}n!$, où n est le nombre de nœuds à visiter,$n!$ est le nombre de routes différentes possibles et le facteur $\frac{1}{2}$ vient du fait que le sens de circulation n'a pas d'importance. Si on a $n = 16$ nœuds comme dans l'exemple à la figure 2.1, il existe déjà 10 461 394 944 000 solutions différentes possibles. Malgré la puissance actuelle des ordinateurs, il est impossible d'énumérer toutes les solutions possibles pour trouver celle qui est optimale lorsque n devient grand. Il est donc nécessaire de développer des algorithmes plus sophistiqués permettant d'explorer plus efficacement toutes ces solutions.

Introduisons quelques notions sur la complexité d'un algorithme.

La **complexité algorithmique** est le prix à payer en termes de ressources pour exécuter un algorithme. Ces ressources peuvent être le nombre d'instructions à exécuter, la quantité de mémoire utilisée, l'utilisation du réseau, etc.

La **complexité temporelle** est la complexité associée au temps d'exécution de l'algorithme, elle est directement liée au nombre d'instructions à exécuter par l'algorithme. On ne s'intéressera ici qu'à ce type de complexité car c'est la ressource temporelle qui est généralement la plus cruciale en pratique. La complexité temporelle est définie en fonction de la taille du problème à

résoudre, dans le cas du VRP, on peut définir cette taille par le nombre de clients qui doivent être desservis.

La **complexité d'un problème** correspond à la complexité de l'algorithme le plus efficace permettant de résoudre ce problème. On utilise la notation $O(\cdot)$ pour représenter les classes de complexité. On utilisera par exemple $O(n^2)$ pour l'ensemble des problèmes ayant une complexité quadratique en la taille n des données d'entrée.

Un **algorithme efficace** est un algorithme dont la complexité est polynomiale.

Le VRP, et les deux problèmes présentés à la section 2.4, le TSP et le problème de *bin packing*, font partie des problèmes dits NP-complets. Les problèmes NP sont des problèmes pour lesquels on peut vérifier efficacement si une solution est valide et dont les solutions possibles peuvent être générées efficacement de manière non déterministe. Les problèmes NP-complets sont les problèmes les plus difficiles de la classe NP, en effet, si un algorithme efficace était connu pour un quelconque problème NP-complet, alors tous les autres problèmes NP pourraient être résolus efficacement.

Les algorithmes connus pour résoudre le VRP ont une complexité temporelle exponentielle dans le pire cas. Ceci signifie qu'il existe des instances pour lesquelles l'algorithme prendra un temps exponentiel par rapport à la taille de l'instance. Pour illustrer l'impact sur les temps d'exécution, comparons quelques complexités polynomiales avec une complexité exponentielle. n est la taille des données d'entrée, l'unité de temps n'est pas précisée car elle n'a pas d'importance.

$O(\cdot)$ \ n	1	10	100	1000
$O(n)$	1	10	100	1000
$O(n^2)$	1	100	10 000	1 000 000
$O(n^3)$	1	1 000	1 000 000	1 000 000 000
$O(2^n)$	2	1 024	$\approx 1,26 \cdot 10^{30}$	$\approx 1,07 \cdot 10^{301}$

TABLE 2.1 – Comparaison de temps d'exécution pour différentes complexités temporelles.

Comme on peut le voir, pour de grandes valeurs de la taille de l'instance, les temps d'exécution deviennent immenses. Nous verrons dans le chapitre suivant les méthodes utilisées en pratique pour résoudre le VRP.

Chapitre 3

État de l'art

Dans ce chapitre, nous allons voir les différentes techniques qui ont été développées pour résoudre le VRP. Il existe beaucoup de méthodes différentes pour résoudre ce problème, on peut les classer en deux types distincts : les méthodes approchées et les méthodes exactes. On va montrer ici uniquement les techniques les plus performantes pour le VRP.

3.1 Méthodes approchées

On appelle méthodes approchées celles qui cherchent une solution approchée au problème. La solution trouvée n'est donc pas nécessairement optimale mais a souvent un coût proche du coût optimal. Pour donner un ordre de grandeur, l'écart entre le coût de la solution trouvée et celui de la solution optimale peut être d'environ 10% pour des méthodes simples et rapides, et jusqu'à moins de 1% pour des méthodes plus sophistiquées et plus longues à exécuter. Bien sûr ces valeurs sont données à titre indicatif, car de nombreux paramètres entrent en compte, notamment :

— la difficulté de l'instance, les instances possédant peu de clients sont plus faciles à résoudre
— le temps d'exécution donné à l'algorithme
— la structure de l'instance, parfois certaines instances possèdent une structure favorisant certaines méthodes, par exemple si les clients sont fortement regroupés.

Parmi les méthodes approchées, on trouve les heuristiques constructives. Celles-ci construisent progressivement une solution ayant un coût relativement proche du coût optimal et en un temps généralement assez court. Elles se basent souvent sur un raisonnement simple et est similaire aux méthodes qui seraient utilisées si on devait résoudre le problème à la main. Nous verrons ici deux exemples différents d'heuristiques constructives.

Par ailleurs, on distingue une grande catégorie d'algorithmes dans les méthodes approchées qui est largement utilisée pour le VRP et ses variantes : les métaheuristiques. Une métaheuristique est un algorithme utilisant généralement des procédés stochastiques pour résoudre approximativement un problème d'optimisation difficile dont on ne connaît pas d'algorithme efficace.

Parmi les métaheuristiques, on trouve les algorithmes de recherche locale. Le but d'une recherche locale est d'explorer l'espace des solutions en modifiant légèrement la solution courante. Deux méthodes basées sur la recherche locale sont particulièrement performantes pour le VRP et seront présentées ici.

Dans l'ensemble des métaheuristiques, on trouve aussi les algorithmes utilisant l'optimisation par colonie de fourmis. Ces techniques ont elles aussi été appliquées avec succès au VRP.

3.1.1 Heuristiques constructives

Heuristique de Clarke et Wright

Cette heuristique constructive a été proposée en 1964 par Clarke et Wright dans [15] pour le CVRP. Elle se base sur les économies réalisées en fusionnant des routes de véhicules différents. Voyons son fonctionnement plus en détail :

Pour démarrer la construction des routes, on considère que chaque client est desservi par un véhicule différent faisant un aller-retour rien que pour lui. On cherche ensuite à fusionner les routes de deux véhicules faisant le plus diminuer le coût total. Pour chaque paire de clients, on définit une économie $s_{ij} = c_{0i} + c_{0j} - c_{ij}$ correspondant à la diminution du coût obtenue en fusionnant les routes de i et j. Pour qu'une fusion de routes soit valide, il faut que :

— i et j soient tous les deux directement connectés au dépôt, sinon la formule de s_{ij} n'est plus valide

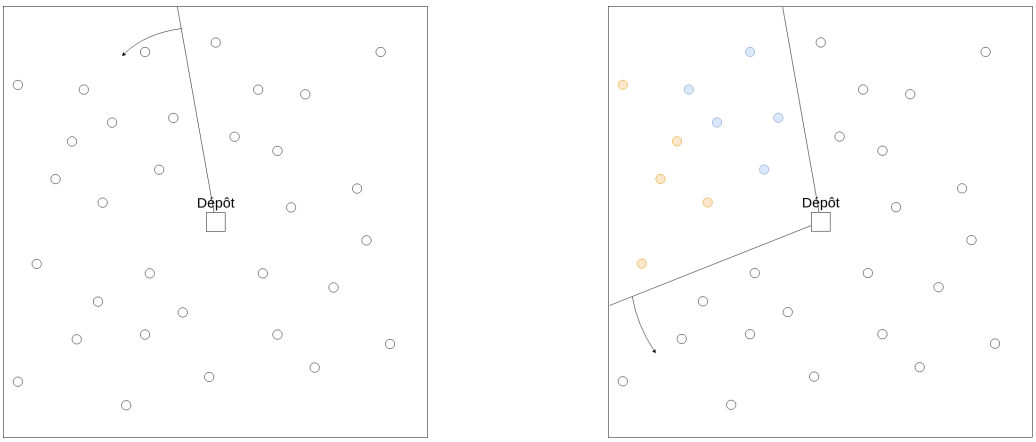
— la demande totale des clients sur la route fusionnée n'excède pas la capacité du véhicule.

On trie alors les économies s_{ij} par ordre décroissant et on fusionne successivement les routes respectant les conditions mentionnées ci-dessus. L'algorithme s'arrête lorsque toute la liste des économies a été parcourue. On peut noter que cette méthode ne tient pas compte du nombre de véhicules K, il se peut donc que la solution trouvée soit inutilisable car elle ne contient pas exactement K véhicules.

Il existe de nombreuses variations de cette heuristique, mais le principe de base reste le même : les économies faites en fusionnant des routes.

Algorithme de balayage

Cet algorithme a été inventé en 1972 par Wren et Holliday dans [5], il est également utilisé pour résoudre le CVRP. Le principe de cette heuristique est simple : on balaye l'espace des clients avec un axe dont l'extrémité se trouve au dépôt et on ajoute les clients aux itinéraires des véhicules dans l'ordre où ils sont rencontrés et tant qu'ils ne font pas excéder la capacité maximale du véhicule. Voyons un exemple pour faciliter la compréhension :



(a) Initialisation de l'axe de balayage.

(b) Balayage de l'espace des clients avec l'axe. On associe au fur et à mesure les clients à un véhicule.

FIGURE 3.1 – Exemple d'application de l'algorithme de balayage.

Dans cet exemple, chaque couleur de la figure 3.1b représente les clients affectés à un véhicule. On passe au véhicule suivant lorsque le client suivant rencontré par l'axe fait excéder la capacité maximale du véhicule. On voit ici que le sixième client rencontré a fait dépasser la capacité du véhicule bleue et a donc été assigné au véhicule suivant, l'orange.

L'algorithme continue ainsi jusqu'à ce que tous les clients soient associés à un véhicule. Lorsque cette phase est terminée, on passe à la seconde étape : la création des itinéraires. Maintenant que l'on connaît l'association entre les clients et les véhicules, il s'agit en fait de résoudre un TSP pour chacun des véhicules. Pour ce faire, on peut utiliser un algorithme tel que le 2-opt, qui sera présenté à la sous-section 3.1.2.

On peut facilement générer des routes légèrement différentes en modifiant les paramètres de l'algorithme : on peut changer la position initiale de l'axe de balayage ou son sens de rotation. Les solutions produites seront alors différentes et on peut choisir de ne garder que la meilleure.

On notera que, comme l'heuristique de Clarke et Wright, la solution trouvée peut ne pas utiliser exactement K véhicules et être donc inutilisable.

3.1.2 Recherche locale

La recherche locale est une des méthodes les plus courantes pour résoudre approximativement les problèmes d'optimisation difficiles. Elle se base sur des modifications successives d'une solution connue pour trouver de meilleures solutions.

Nous allons voir ici un des algorithmes basiques utilisant la recherche locale pour trouver une solution approchée du TSP : l'algorithme 2-opt. On se servira de celui-ci pour illustrer les notions liées à la recherche locale. Toutes ces notions sont bien sûrs identiques et directement transposables pour le VRP, le TSP est utilisé uniquement pour faciliter les exemples.

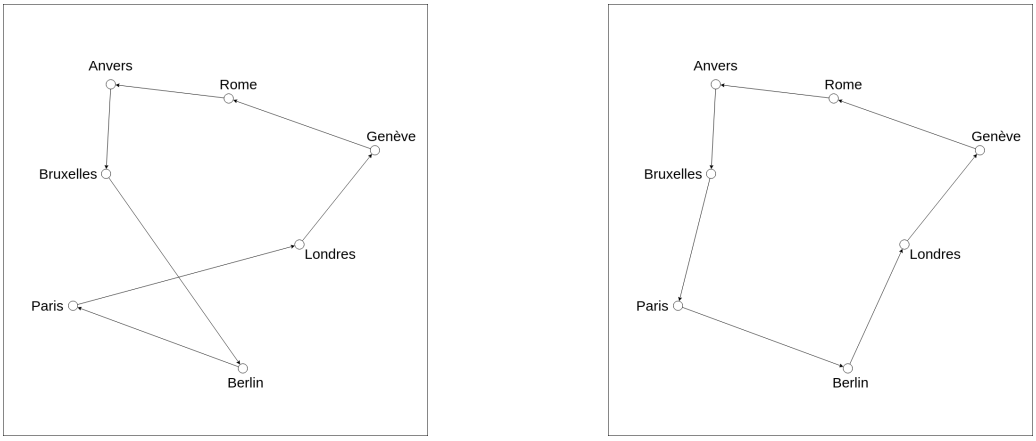
C'est un algorithme itératif qui cherche à chaque étape les deux meilleures arêtes à supprimer et reconnecter de manière croisée. Par "meilleure", on parle en termes de réduction du coût total de l'itinéraire. Cette déconnexion/reconnexion s'effectue comme ceci :

— on choisit deux arêtes à supprimer, notons les (A, B) pour l'arête allant de A vers B et (C, D) pour l'arête allant de C vers D

— on reconnecte A à C et B à D et on inverse le sens des arêtes se trouvant sur le chemin de B à C.

Les deux arêtes qui sont déconnectées/reconnectées ne peuvent pas avoir d'extrémités en commun.

Voici un exemple de l'application d'une étape de 2-opt :



(a) Avant 2-opt (b) Après 2-opt

FIGURE 3.2 – Exemple d'application d'une étape de 2-opt.

L'algorithme s'arrête s'il n'a pas pu améliorer la route entre deux itérations, autrement dit, si toutes les tentatives de déconnexions/reconnexions croisées possibles ont mené à une solution ayant un coût supérieur.

Introduisons maintenant quelques notions liées à la recherche locale et qui seront illustrées avec le TSP et l'algorithme 2-opt [46].

La **fonction objectif** d'un problème d'optimisation est l'équation décrivant le critère à optimiser. Dans le cas du TSP, elle est définie par l'équation 2.1 : $\min \sum_{e \in E} c_e x_e$, qui indique qu'on doit minimiser le coût total du cycle visitant chaque ville.

L'**espace des solutions** est l'ensemble des solutions possibles pour le problème. Pour le TSP, ceci correspond à l'ensemble des cycles différents qui passent par toutes les villes.

La **solution courante** est la solution actuelle qui doit être modifiée par l'algorithme pour essayer de trouver une meilleure solution. La solution obtenue après une modification de la solution courante est appelée **voisin**. Dans l'exemple du 2-opt, la figure 3.2a est la solution courante et la figure 3.2b est un de ses voisins.

Le **voisinage** est l'ensemble des voisins. Pour le 2-opt, il s'agit de l'ensemble des solutions obtenues en effectuant toutes les déconnexions/reconnexions croisées possibles.

L'**opérateur de voisinage** est la fonction définissant les modifications à faire à la solution courante pour générer son voisinage. Pour le 2-opt, l'opérateur de voisinage est la déconnexion/-reconnexion croisée.

Un **mouvement** est le procédé permettant de générer un voisin de la solution courante. Par exemple, "échange du client 5 et 10" pourrait être un mouvement dans lequel on interchange la position des clients 5 et 10 dans l'itinéraire d'un véhicule. La figure 3.2 montre l'application d'un mouvement 2-opt.

Une **recherche locale** est un processus itératif où à chaque itération on applique un mouvement à la solution courante. Pour produire la première solution courante qui permettra de démarrer l'algorithme, on utilise souvent une méthode heuristique constructive, telle que celles présentées au début de ce chapitre, car elles sont rapides à exécuter et la solution trouvée a généralement une fonction objectif déjà assez bonne.

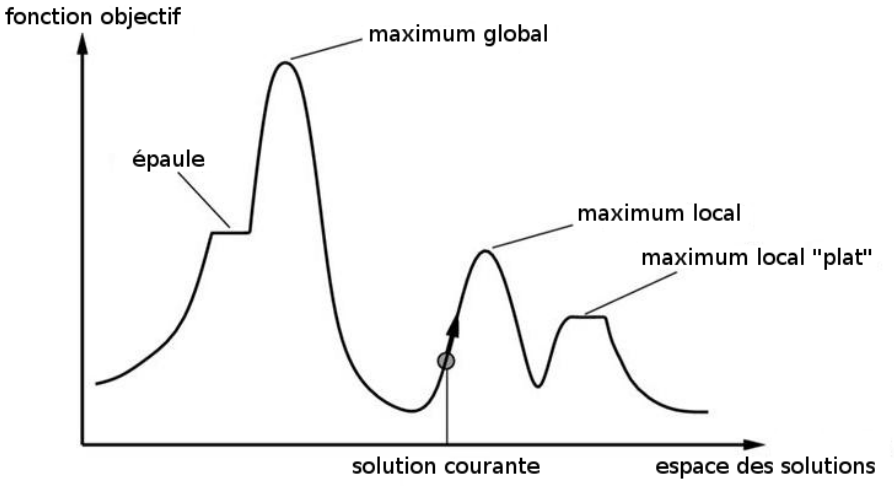


FIGURE 3.3 – Représentation de la fonction objectif dans l'espace des solutions. Source : [46], page 121.

Dans cette figure, on représente en abscisse l'espace des solutions de notre problème. On considère que deux solutions similaires sont proches l'une de l'autre sur cet axe. Autrement dit, les voisins d'une solution sont les points qui sont proches de lui. En ordonnée, on représente la fonction objectif du problème. Dans cet exemple, il s'agit d'un problème de maximisation, on cherche à trouver le **maximum global**, c'est-à-dire la solution pour laquelle la fonction objectif est maximale. On parle de **maximum local** lorsque les points dans le voisinage de ce maximum local ont une valeur de fonction objectif inférieure ou égale.

Établissons une stratégie simple de recherche locale : on choisit parmi les voisins de la solution courante celui dont la fonction objectif est la plus élevée et on répète cela jusqu'à ne plus pouvoir améliorer la solution. On appelle cette stratégie le *hill climbing* (escalade de colline). En utilisant

cette stratégie basique en partant de la solution courante indiquée sur le schéma, on arrive au point correspondant au maximum local.

Ce petit exemple permet de voir les difficultés que cette stratégie basique peut rencontrer. En effet, si la solution courante n'a pas de voisins ayant une meilleure valeur de fonction objectif, l'algorithme y est bloqué et s'arrête. Ce problème peut survenir dans le cas de maximums locaux, mais aussi en cas de zones plates comme une **épaule** et empêchera souvent l'algorithme de trouver le maximum global. Il est donc nécessaire de mettre en place des stratégies plus sophistiquées pour éviter ce genre de problème.

Les algorithmes de recherche locale sont souvent décrits en utilisant les notions d'**intensification** et de **diversification**.

L'**intensification** est le fait d'explorer une zone prometteuse de l'espace des solutions dans le but d'y trouver une meilleure solution.

La **diversification** est le fait d'explorer des zones encore inexplorées de l'espace des solutions pour y trouver éventuellement une meilleure solution.

Avec ces deux nouvelles notions, on voit que notre stratégie de *hill climbing* ne fait en réalité que de l'intensification, on cherche en permanence à améliorer la solution. Elle manque de diversification pour pouvoir se débloquer des maximums locaux et des zones plates.

Un bon algorithme de recherche locale est donc un algorithme qui arrive à trouver l'équilibre entre intensification et diversification. L'intensification doit permettre d'améliorer la meilleure solution connue, tandis que la diversification doit élargir la zone de recherche.

Le fait de pouvoir accepter de dégrader la solution est souvent un élément clé au succès des méthodes de recherche locale, car cela permet d'explorer une plus grande partie de l'espace de recherche et donc de potentiellement trouver le maximum global.

Nous allons voir maintenant des algorithmes de recherche locale plus sophistiqués qui sont utilisés pour le VRP.

Recuit simulé

Cette méthode est un algorithme de recherche locale basé sur un processus métallurgique où l'on contrôle le refroidissement d'un matériau pour atteindre son point le plus stable.

Cette technique a été adaptée aux problèmes d'optimisation. On démarre avec une haute température que l'on va faire progressivement diminuer, cette température peut-être vue comme un paramètre influençant directement la diversification de l'algorithme. Lorsque la température est élevée, la diversification est importante, et inversement lorsqu'elle est basse.

A chaque itération, on choisit aléatoirement un candidat dans le voisinage de la solution courante. Si c'est une meilleure solution, elle est acceptée et devient la solution courante (intensification). Si c'est une solution moins bonne, elle est acceptée avec une certaine probabilité. Cette probabilité est influencée par deux paramètres :

— la température actuelle, plus elle est élevée, plus la probabilité augmente
— la différence de qualité des solutions, si on dégrade beaucoup la solution, la probabilité diminue

Cette méthode, associée à une recherche à voisinage large, a eu récemment de très bonnes performances pour le problème CVRP (VRP avec contrainte de capacité) dans [50].

Recherche tabou

Ici, l'idée est de choisir à chaque étape le voisin améliorant (ou diminuant le moins) la qualité de la solution courante (intensification). On introduit ensuite un mécanisme d'interdiction de certains mouvements (diversification). Effectivement, sans cette restriction, le risque de se trouver dans un cycle où l'on visite tout le temps les mêmes solutions est très grand.

Il y a plusieurs stratégies à mettre en place :

— la stratégie d'interdiction : elle décide quels sont les mouvements à interdire

— la stratégie d'autorisation : elle décide quand un mouvement redevient disponible
— la stratégie à court terme : elle permet d'autoriser un mouvement normalement interdit, par exemple si ce mouvement mènerait à une amélioration de la meilleure solution connue.

Pour voir comment cet algorithme peut être appliqué en pratique pour le VRP, voir les références [33], [26] et [24].

3.1.3 Optimisation par colonie de fourmis

Cette métaheuristique est basée sur le comportement collectif des fourmis pour trouver les chemins les plus courts à une source de nourriture. Cette technique a récemment été combinée avec des algorithmes génétiques pour améliorer ses performances dans [43] et [9] notamment.

C'est une des méthodes qui est appliquée dans ce travail et on y consacrera donc un chapitre (chap. 4) pour voir comment un solveur utilisant cette méthode fonctionne.

3.2 Méthodes exactes

On appelle méthodes exactes celles qui garantissent de trouver la solution optimale du problème.

3.2.1 Optimisation linéaire en nombres entiers

Cette méthode est aussi appliquée dans ce travail et on y consacrera donc également un chapitre (chap. 5) pour voir son fonctionnement.

Cette méthode a été appliquée avec beaucoup de succès au VRP et à ses variantes. Les algorithmes actuels les plus performants combinent cette méthode à la génération de colonnes (voir [42] pour les détails de l'algorithme) et peuvent résoudre optimalement des instances possédant plus de 100 clients.

Chapitre 4

Solveur par optimisation par colonie de fourmis

Dans ce chapitre, nous allons voir en détail le fonctionnement d'un solveur par optimisation par colonie de fourmis (OCF, ou ACO en anglais, Ant Colony Optimization). L'algorithme qui est montré ici est celui qui a été implémenté dans le cadre de ce travail et sera utilisé lors de la collaboration entre solveurs. Il existe beaucoup de variantes de cet algorithme dans la littérature et cette méthode est appliquée pour de nombreux autres problèmes, la référence [32] explique en détail les fondements théoriques de cette méthode et comment l'appliquer pour différents problèmes.

4.1 Idée générale

Ce type de solveur est inspiré, comme son nom l'indique, du comportement des fourmis en société. En effet, en observant leur comportement, les chercheurs ont pu constater qu'elles avaient tendance à utiliser le plus court chemin disponible à leur source de nourriture et ce malgré leurs capacités cognitives limitées. Cette intelligence collective est expliquée comme suit :

- les fourmis éclaireuses cherchent des sources de nourriture en se déplaçant aléatoirement
- après avoir trouvé de la nourriture, elles rentrent au nid en laissant derrière elle une piste de phéromones
- les autres fourmis étant attirées par les phéromones vont suivre approximativement la piste et laisser à leur tour des phéromones lors de leur retour au nid
- si plusieurs pistes de phéromones sont disponibles, les pistes les plus courtes auront tendance à être privilégiées car elles pourront être parcourues plus de fois en un temps donné
- les autres pistes de phéromones plus longues auront tendance à disparaître car de moins en moins de fourmis l'emprunteront

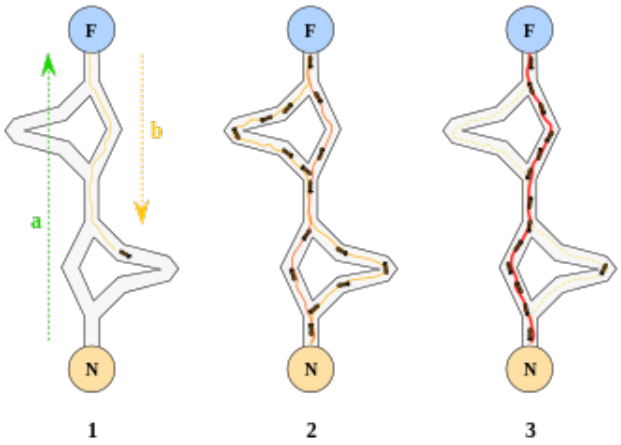


FIGURE 4.1 – Exemple de l'intelligence collective des fourmis leur permettant de trouver le chemin le plus court à une source de nourriture. Source : [6].

Sur cette figure, la première image représente la fourmi cherchant la nourriture F (chemin a) et laissant une piste de phéromones à son retour au nid N (chemin b). La deuxième image montre le renforcement des pistes par les autres fourmis. La dernière image montre la piste finale utilisée par la colonie de fourmis.

4.2 Algorithme

Le solveur décrit ici est basé principalement sur les techniques présentées dans [9], [7], [27] et [23].

Ce solveur utilise un processus itératif : à chaque itération de l'algorithme, on produit une nouvelle solution et on met à jour les quantités de phéromones. Quand le temps d'exécution ou le nombre d'itérations limite est atteint, on retourne la meilleure solution trouvée. On peut représenter le schéma d'exécution comme ceci :

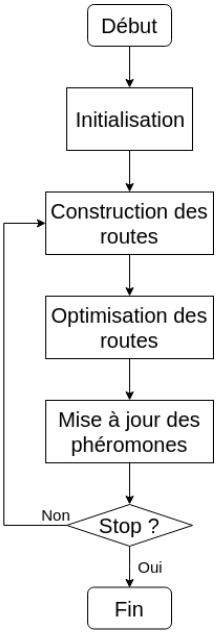


FIGURE 4.2 – Schéma global de l'exécution du solveur

Voyons maintenant en quoi consiste chacune de ces étapes.

4.2.1 Initialisation

Dans cette étape, on s'intéresse à l'initialisation de la quantité de phéromones présentes sur les arêtes. On peut décider de mettre une quantité de phéromones égale sur chaque arête afin de laisser les fourmis découvrir par elles-mêmes les meilleurs chemins à emprunter ou alors d'utiliser un algorithme pour déposer des phéromones sur les chemins prometteurs. Pour ce faire, on peut par exemple utiliser une des méthodes constructives vues au point 3.1.1 et déposer les phéromones sur les routes trouvées par cet algorithme. Ici, c'est l'heuristique de Clarke et Wright qui est utilisée, les phéromones sont déposées exactement de la même manière que si les routes avaient été générées par la colonie de fourmis. Ceci sera détaillé dans la partie consacrée à la mise à jour des phéromones.

4.2.2 Construction des routes

Pour construire les routes, on considère que chaque véhicule est représenté par une fourmi. On construit les routes de chaque fourmi itérativement, elle commence avec une route vide et décide à chaque itération le prochain client à desservir ou de retourner au dépôt.

Pour choisir son prochain client, la fourmi se base sur deux informations : les pistes de phéromones et la visibilité. Ces deux notions sont définies entre chaque paire de nœuds (par "nœud", on entend un client ou le dépôt). Les phéromones doivent indiquer les routes empruntées précédemment qui ont mené à une bonne solution. La visibilité sert à contraindre la fourmi à choisir les clients les plus proches d'elle.

La visibilité peut se définir de beaucoup de manières différentes et a un impact significatif sur les performances. En principe, elle doit être plus élevée pour les clients étant de bons candidats pour la prochaine destination d'un véhicule. Le but est donc de favoriser les clients dont le coût pour les atteindre est faible et étant bien situés. Dans l'implémentation de cette méthode, c'est la définition proposée par [23] qui est utilisée :

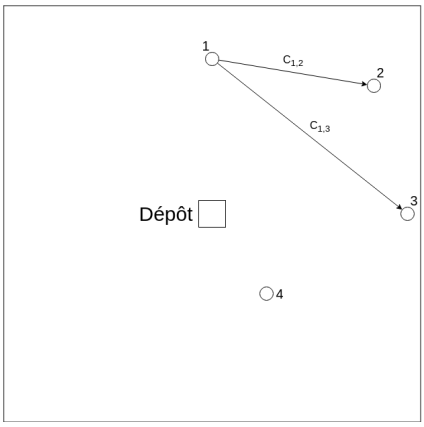
$$\eta_{ij} = c_{i0} + c_{0j} - g \cdot c_{ij} + f \cdot |c_{i0} - c_{0j}| \tag{4.1}$$

— η_{ij} est la visibilité entre les nœuds i et j
— c_{ij} est le coût entre les nœuds i et j
— g et f sont des paramètres positifs à fixer.

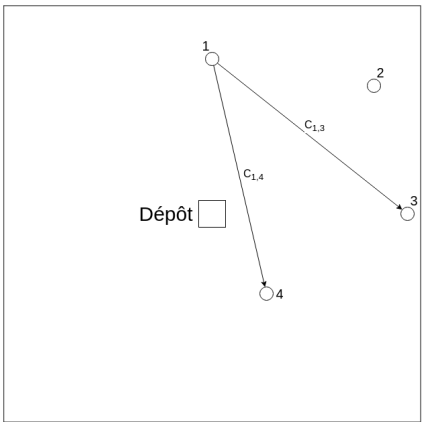
Les paramètres g et f ont ici été fixés à $f = g = 2$ car ce sont les valeurs ayant les meilleures performances.

Pour comprendre l'intuition derrière cette formule, supposons que le coût c_{ij} entre les nœuds i et j soit défini par la distance qui les sépare. Dans ce cas, on souhaite favoriser les clients étant proches et étant bien situés par rapport au dépôt. Un client bien situé est un client qui ne force pas le véhicule à passer près du dépôt. En effet, en passant près du dépôt, on fait en quelque sorte un détour, puisqu'il faudra plus tard y terminer la tournée du véhicule. Il est donc généralement préférable que ce client soit desservi par un autre véhicule.

Voici des exemples pour illustrer ces deux aspects :



(a) En partant du client 1, le client 2 est un meilleur candidat que le client 3 car il est plus proche.



(b) En partant du client 1, le client 3 est un meilleur candidat que le client 4 car il est mieux situé.

FIGURE 4.3 – Illustration de l'intuition de la formule de visibilité 4.1.

Le choix du prochain client j en se trouvant en i est fait selon la formule probabiliste suivante :

$$p_{ij} = \begin{cases} \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_j \tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}} & \text{si } j \text{ est un candidat} \\ 0 & \text{sinon} \end{cases} \tag{4.2}$$

τ_{ij} est la quantité de phéromones sur l'arête (i, j), η_{ij} est la visibilité comme définie précédemment, α et β sont des paramètres permettant d'influencer la pondération entre les pistes de phéromones et la visibilité.

Un client j est candidat à trois conditions :

— il fait partie des nœuds ayant la meilleure visibilité depuis le nœud actuel i

— la demande de ce client ne fait pas excéder la capacité du véhicule

— le client n'a pas encore été desservi.

Cette formule probabiliste 4.2 vise donc à favoriser d'une part les arêtes dont la visibilité est élevée, qui fait donc du client un bon candidat, et à la fois les arêtes ayant une grande quantité de phéromones, qui ont donc déjà contribué à l'élaboration de bonnes routes dans de précédentes itérations.

En pratique, on fixe une taille limite à la liste de candidats considérés à chaque client, par exemple, on décide de ne garder au maximum que 25% des voisins de chaque nœud. C'est un paramètre à fixer par l'utilisateur. Il a deux conséquences directes sur les performances :

— une réduction du temps de calcul pour une itération, car on considère moins de voisins

— une amélioration globale de la qualité des solutions trouvées pour un nombre d'itérations fixe, car lorsque la fourmi choisit un client lointain, cela mène généralement à une solution de moins bonne qualité.

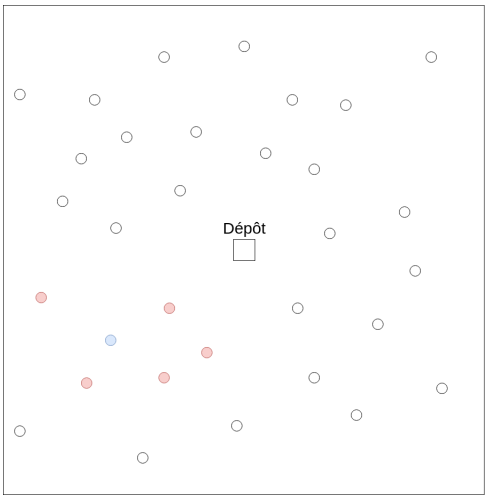


FIGURE 4.4 – Exemple de liste de candidats. Les points rouges sont les candidats considérés en partant du client bleu

La fourmi rentre au dépôt lorsqu'il n'y a plus aucun candidat au client où elle se trouve. On commence alors la construction de la route de la fourmi suivante de la même manière et ce jusqu'à avoir atteint le nombre de véhicules requis (donnée reçue en entrée). Il est possible que la solution trouvée n'utilise pas exactement K véhicules, par exemple lorsque les fourmis sont rentrées au dépôt alors qu'il leur restait un peu de capacité. Dans ce cas, l'algorithme efface les routes générées et recommence la construction des routes depuis le début.

4.2.3 Optimisation des routes

Cette étape vise à améliorer les routes construites en optimisant chacune des routes individuellement. Chaque route est considérée comme un TSP, le but est donc de trouver un itinéraire de coût inférieur passant par les clients assignés à ce véhicule.

Pour ce faire, on utilise ici l'algorithme de recherche locale 2-opt, il a été présenté à la sous-section 3.1.2.

Cette étape du solveur n'est en soit pas indispensable mais permet généralement d'améliorer les solutions générées pour un faible coût computationnel. Il existe beaucoup d'autres algorithmes utilisables pour cette étape, puisqu'on peut utiliser tous ceux utilisés pour résoudre le TSP. On notera notamment l'heuristique de Lin-Kernighan qui est parfois utilisée pour ce type de solveur. Elle est particulièrement adaptée pour des problèmes où le nombre de clients desservis par véhicule est élevé. Elle ne sera pas utilisée ici car son implémentation est plus complexe et nous utiliserons des instances de taille relativement modeste lors des expériences. Une description détaillée de cette heuristique est disponible dans [44].

4.2.4 Mise à jour des phéromones

La dernière étape du solveur est de mettre à jour les pistes de phéromones. Il existe de nombreuses manières différentes de procéder, c'est généralement cette étape qui différencie le plus les différents solveurs utilisant l'optimisation par colonie de fourmis. Dans mon implémentation, j'ai utilisé la méthode proposée par [9].

Le but ici est donc de modifier les quantités de phéromones présentes sur les arêtes afin de mieux guider les prochaines fourmis qui construiront leur route. Dans un premier temps, on les fait s'évaporer car, comme dans la réalité, un chemin qui n'est plus emprunté perd au fur et à mesure sa piste de phéromones.

Ensuite, on augmente leur quantité sur les arêtes appartenant à la solution construite. On distingue deux contributions à la quantité de phéromones que l'on ajoute sur une arête de la

solution :

— si la solution est de bonne qualité, on en ajoute plus

— si l'arête participe à la qualité de la route à laquelle elle appartient, on en ajoute plus.

Mathématiquement, voici la formule utilisée pour mettre à jour les phéromones sur chaque arête :

$$\tau_{ij}^{new} = \begin{cases} \rho \cdot \tau_{ij}^{old} + \Delta\tau_{ij} & \text{si l'arête (i, j) est dans la solution} \\ \rho \cdot \tau_{ij}^{old} & \text{sinon} \end{cases} \tag{4.3}$$

ρ est un paramètre à fixer pour simuler l'évaporation, il doit être compris entre 0 et 1. Une valeur de 0,9 correspond par exemple à une évaporation de 10% de la quantité déjà présente.

$\Delta\tau_{ij}$ est l'incrément de la quantité de phéromones pour l'arête (i, j), il est défini comme ceci :

$$\Delta\tau_{ij} = \frac{Q \cdot K}{L} \cdot \frac{D - c_{ij}}{m \cdot D} \tag{4.4}$$

— Q est un paramètre permettant de fixer l'échelle

— K est le nombre de véhicules utilisés

— L est le coût total de la solution

— D est le coût total de la route à laquelle l'arête appartient

— m est le nombre de clients desservis sur la route

— c_{ij} est le coût entre i et j

On voit dans cette formule que $\frac{Q \cdot K}{L}$ est la contribution due à la qualité globale de la solution. Tandis que $\frac{D - c_{ij}}{m \cdot D}$ est la contribution liée à la qualité de la route de l'arête.

Finalement, on fixe une borne minimale et maximale aux quantités qui peuvent se trouver sur une arête.

La borne minimale sert à éviter qu'une arête ne puisse plus jamais être sélectionnée par une fourmi. En effet, si la quantité de phéromones vaut 0, d'après la formule probabiliste 4.2 montrée à l'étape de construction des routes, la probabilité de sélectionner l'arête est de 0.

La borne maximale sert à éviter qu'une arête soit tout le temps sélectionnée. Même si la sélection de l'arête mène souvent à de bonnes solutions, elle risque d'empêcher la découverte d'autres routes pouvant être de meilleure qualité.

Ces bornes sont fixés comme ceci :

$$\tau_{min} = \frac{Q}{\sum_i 2c_{0i}} \text{ et } \tau_{max} = \frac{Q}{\sum_i c_{0i}}$$

— Q est la même constante que celle définie ci-dessus

— c_{0i} est le coût entre le dépôt et le client i

Ces bornes ont été déterminées de manière empirique par [9] et ont été reprises pour cette implémentation, elles n'ont pas de signification particulière. En effet, on remarque que ce qui importe dans la formule probabiliste 4.2, ce sont les quantités relatives de phéromones et non les valeurs en elles-mêmes, il est donc cohérent de les fixer "arbitrairement".

4.2.5 Redémarrages

Après un grand nombre d'itérations de cet algorithme, il est intéressant de faire un redémarrage. En effet, certaines arêtes étant fortement marquées par les phéromones sont empruntées presque systématiquement car la probabilité qu'elles soient sélectionnées est très élevée. Les solutions trouvées sont donc souvent similaires et ne permettent plus d'améliorer la meilleure solution connue.

Un redémarrage consiste à réinitialiser toutes les valeurs de phéromones associées aux arêtes afin de permettre aux fourmis de découvrir, éventuellement, de nouvelles routes.

Chapitre 5

Solveur par optimisation linéaire en nombres entiers

Dans ce chapitre, nous allons voir le fonctionnement d'un solveur par optimisation par optimisation linéaire en nombres entiers (OLNE, ou ILP en anglais, Integer Linear Programming). Un solveur utilisant cette technique a été implémenté dans le cadre de ce travail et sera utilisé lors de la collaboration.

5.1 Idée générale

L'optimisation linéaire en nombres entiers est une méthode que l'on peut appliquer uniquement aux problèmes pouvant être décrits par un modèle mathématique ayant certaines propriétés :

— les variables utilisées doivent prendre des valeurs positives entières
— les contraintes imposées sur ces variables doivent être linéaires
— la fonction objectif doit également être linéaire.

Un problème ILP peut être écrit sous sa forme dite canonique :

$$\min \mathbf{c^T x} \tag{5.1}$$

tel que

$$\mathbf{Ax} \leq \mathbf{b} \tag{5.2}$$

$$\mathbf{x} \geq \mathbf{0} \tag{5.3}$$

$$\mathbf{x} \in \mathbb{Z}^n \tag{5.4}$$

Notons n le nombre de variables et m le nombre de contraintes. $\mathbf{x}$ est un vecteur de dimension n contenant les variables. Les vecteurs colonne $\mathbf{b}$ de taille m et $\mathbf{c}$ de taille n, et la matrice $\mathbf{A}$ de taille $m \times n$ contiennent des coefficients à valeurs entières.

L'équation 5.1 indique que la fonction objectif est une combinaison linéaire des variables. Il s'agit ici d'un problème de minimisation, mais la définition de la forme canonique pour les problèmes de maximisation est bien sûr identique.

L'équation 5.2 est l'équation indiquant que les contraintes sur les variables doivent être linéaires.

Les équations 5.3 et 5.4 imposent que les variables soient positives et entières. On appelle généralement la contrainte 5.4, la contrainte d'intégralité.

On remarque que la formulation mathématique du VRP présentée à la section 2.2 peut être facilement mise sous cette forme canonique, l'optimisation linéaire en nombres entiers peut donc être utilisée pour le résoudre.

Voyons un exemple de problème ILP sous forme canonique :

$$\max x_2$$

$$-x_1 + x_2 \leq 1$$

$$3x_1 + 2x_2 \leq 12$$

$$2x_1 + 3x_2 \leq 12$$

$$x_1, x_2 \geq 0$$

$$x_1, x_2 \in \mathbb{Z}$$

Pour cet exemple, $\mathbf{x}$, $\mathbf{c}$, $\mathbf{b}$ et $\mathbf{A}$ sont donnés par :

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad \mathbf{c} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 12 \\ 12 \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} -1 & 1 \\ 3 & 2 \\ 2 & 3 \end{bmatrix}$$

On peut représenter graphiquement cet exemple dans un espace à deux dimensions où les axes correspondent aux variables x_1 et x_2 :

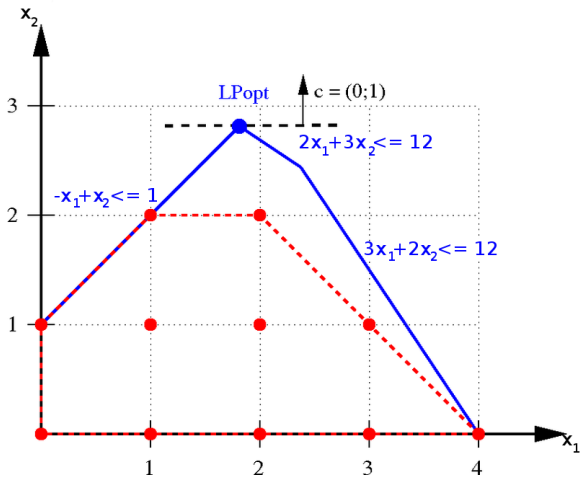


FIGURE 5.1 – Représentation graphique de l'exemple de problème ILP décrit ci-dessus. Source : [25].

Les contraintes sont représentées en bleu. La ligne en pointillé noir représente la fonction objectif, la flèche vers le haut indique qu'il s'agit d'une maximisation. Les points rouges sont les points dits **réalisables**, ce sont les points qui respectent toutes les contraintes, y compris celle d'intégralité.

Les lignes en pointillés rouges représentent l'**enveloppe complexe**, il s'agit du plus petit polyèdre contenant tous les points réalisables. Pour mieux visualiser ce que représente cette enveloppe complexe, il peut être utile de faire une analogie avec un élastique entourant un ensemble de points :

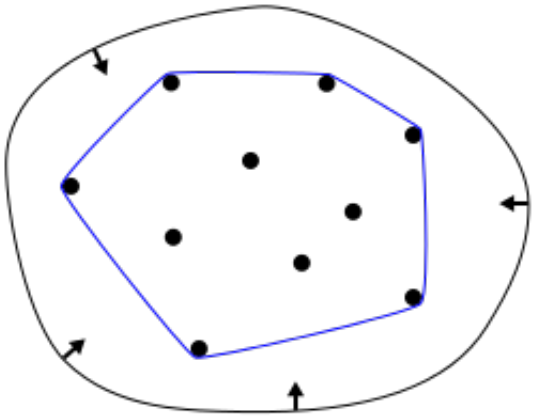


FIGURE 5.2 – Analogie entre un élastique entourant un ensemble de points et leur enveloppe complexe. Source : [13].

Voyons maintenant le fonctionnement de l'algorithme qui est utilisé dans le solveur.

5.2 Algorithme

L'algorithme présenté ici est basé sur le principe de **séparation et évaluation** (**branch and bound**, en anglais). Ce principe est formulé en deux étapes qui sont présentées ci-dessous.

L'étape de **séparation** (ou **branchement**) consiste à partitionner un problème P en sous-problèmes P_i tel que le coût de la solution trouvée en résolvant P est le même que celui de la meilleure solution trouvée en résolvant chacun des sous-problèmes P_i.

Par exemple, pour le problème ILP ci-dessous, une procédure de séparation valide est de partitionner P en sous-ensembles $P_1, ..., P_k$:

$$\min \mathbf{c^T x}$$
$$\mathbf{x} \in P$$

est équivalent à résoudre les sous-problèmes pour $i \in 1, ..., k$ et garder la meilleur solution trouvée :

$$\min \mathbf{c^T x}$$
$$\mathbf{x} \in P_i$$

On peut ensuite continuer à appliquer ce principe de séparation aux sous-problèmes, on se retrouve alors avec une structure en arbre comme ceci :

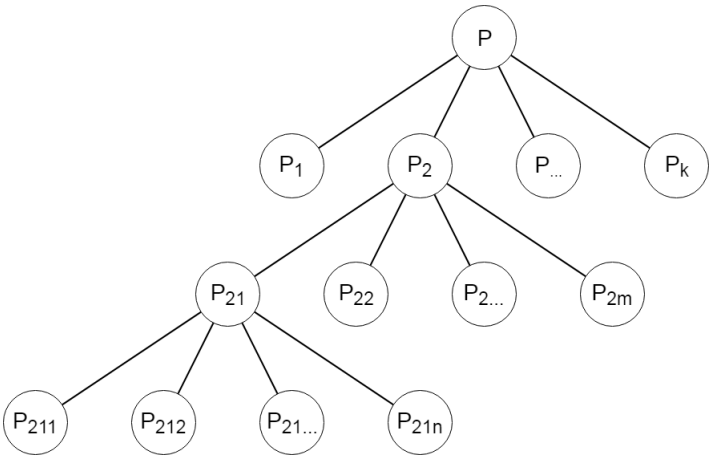


FIGURE 5.3 – Structure en arbre formée lors des séparations successives des problèmes en sous-problèmes.

On appelle communément cette structure l'**arbre de recherche**.

L'étape d'**évaluation** consiste à calculer une borne sur la fonction objectif pour un sous-problème. Le but de cette borne est d'essayer de déterminer si la meilleure solution connue actuellement peut être améliorée à partir de ce sous-problème. Si la borne obtenue indique que ce n'est pas possible, alors il n'est pas nécessaire de le résoudre.

Pour le problème présenté ci-dessus, cette borne se traduit par :

$$b(P_i) \leq \min_{x \in P_i} \mathbf{c^T x}$$

$b(P_i)$ est la procédure permettant de générer une borne pour un sous-problème P_i. On cherche à ce que cette procédure ait les deux caractéristiques suivantes :

— elle doit être rapide à calculer

— elle doit donner une borne inférieure proche de l'optimum pour le sous-problème.

Dans le cas de problème ILP, on utilise couramment la **relaxation linéaire**, cela consiste à résoudre le problème en retirant la contrainte d'intégralité sur les variables. Si une variable x a pour domaine $\{0, 1\}$, sa version relaxée aura l'intervalle $[0, 1]$ pour domaine. Effectivement, en retirant la contrainte d'intégralité, cela revient à résoudre un problème d'optimisation linéaire classique, pour lequel des algorithmes efficaces sont connus. L'un d'entre eux est l'algorithme du simplexe.

L'**algorithme du simplexe** se base sur la notion de polytope convexe. Un **polytope convexe** est un objet géométrique défini dans n'importe quel nombre de dimensions, ayant des faces planes polygonales et dont l'ensemble de points qui le constitue forment un ensemble de points convexe. L'ensemble des contraintes d'un problème d'optimisation linéaire forme un polytope convexe. Sur la figure 5.1, le polytope convexe formé est celui défini par les contraintes, en bleu, et les axes x et y. L'enveloppe complexe, en pointillé rouge est également un polytope convexe. L'algorithme du simplexe se base sur une propriété très importante du polytope convexe formé par les contraintes : au moins un de ses sommets est nécessairement une solution optimale du problème. Cette propriété a également un intérêt particulier pour l'enveloppe complexe. Puisqu'il s'agit d'un polytope convexe et que tous ses sommets sont des points réalisables, une des solutions optimales d'un problème ILP est un des sommets de son enveloppe complexe.

L'algorithme du simplexe est un processus itératif démarrant en l'un des sommets du polytope convexe formé par les contraintes linéaires. A chaque itération, on se déplace le long de l'arête menant à la plus grande amélioration de la fonction objectif. Lorsqu'il n'est plus possible de trouver un sommet améliorant la fonction objectif, l'algorithme s'arrête et le sommet courant est la solution optimale.

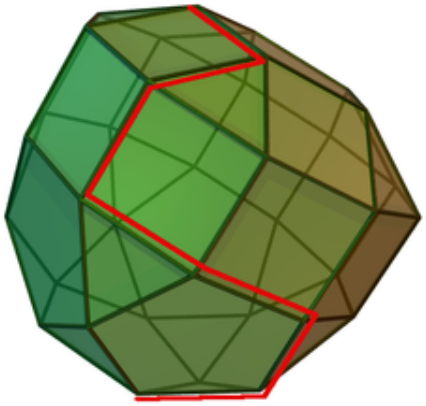


FIGURE 5.4 – Exemple du chemin emprunté par l'algorithme du simplexe sur un polytope convexe en trois dimensions. Source : [48].

Pour plus de détails sur l'optimisation linéaire et l'algorithme du simplexe, voir [29].

Passons maintenant au solveur pour le VRP en lui-même, voici un schéma montrant l'architecture globale :

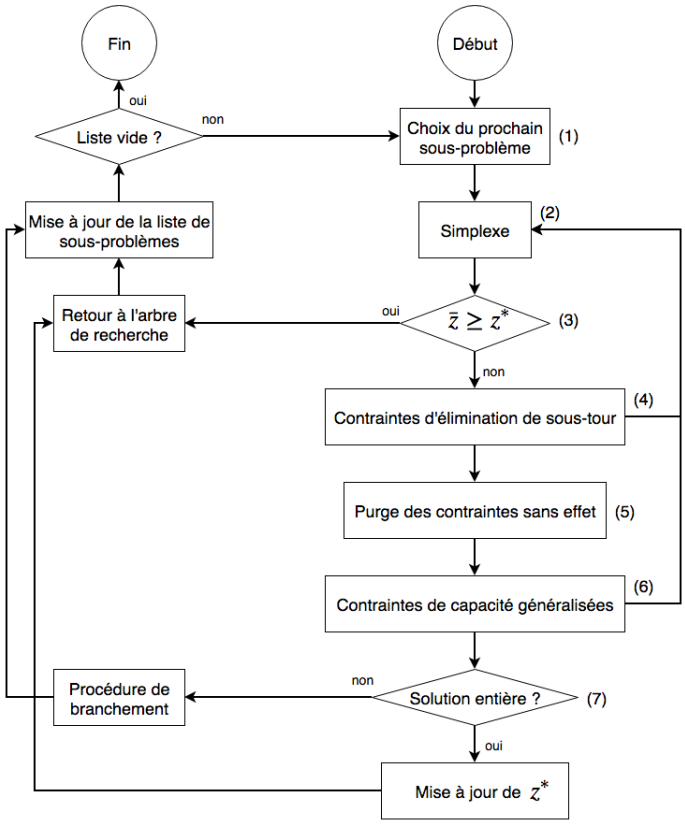


FIGURE 5.5 – Schéma global du solveur. Source : [17].

On retrouve dans ce schéma les notions et algorithmes expliqués précédemment, on va passer en revue les différentes étapes.

Étape (1). On commence par choisir le prochain sous-problème à traiter, par rapport à notre analogie avec l'arbre de recherche, cela correspond donc à un nœud de l'arbre.

Étape (2). On utilise la relaxation linéaire sur ce nœud pour trouver une borne, on utilise donc l'algorithme du simplexe pour résoudre le problème d'optimisation linéaire. Les contraintes

utilisées sont donc celles présentées dans le modèle mathématique du VRP, section 2.2, mais sans les contraintes d'intégrité.

Étape (3). On compare la borne obtenue, notée $\bar{z}$ avec la meilleure solution connue actuellement, notée z^*. Si la borne est moins bonne que z^*, il n'est pas nécessaire de continuer à résoudre pour ce nœud et on retourne à l'arbre de recherche.

Étape (4). On vérifie ensuite que la solution ne contient pas de sous-tour. Par sous-tour, on entend un cycle de clients ne contenant pas le dépôt et donnant donc lieu à une solution invalide. Il est nécessaire de vérifier cela car les contraintes d'élimination de sous-tour sont introduites de manière *lazy*, c'est-à-dire qu'elles sont introduites seulement lorsque c'est nécessaire. En effet, l'équation 2.4 empêchant les sous-tours définie dans le modèle mathématique du VRP donne lieu en pratique à un nombre exponentiel de contraintes. Il serait donc inapproprié d'essayer de tenir compte d'autant de contraintes dans le modèle de base.

Étape (5). La purge des contraintes sans effet consiste à retirer du modèle les contraintes n'ayant plus d'utilité. Ce cas peut se présenter par exemple avec les contraintes de sous-tour qui sont ajoutées au cours de l'exécution, si certaines d'entre elles ne sont plus violées depuis un grand nombre d'itérations, elles peuvent être retirées du modèle. Elles seront à nouveau ajoutées plus tard si nécessaire. Cela permet d'alléger le modèle et donc d'accélérer la recherche.

Étape (6). On vérifie maintenant que la solution respecte les contraintes de capacité. Pour la même raison qu'à l'étape 4, on rajoute ces contraintes de manière *lazy* pour ne pas surcharger le modèle.

Étape (7). Finalement, on regarde si la solution est entière ou non. Une solution entière est une solution dont les variables prennent toutes des valeurs entières et qui respecte donc les contraintes d'intégrité. Cette étape est effectivement nécessaire puisqu'on a utilisé la relaxation linéaire à la deuxième étape. Si elle est entière, alors on met à jour la meilleure solution connue si la solution trouvée est meilleure, sinon, on applique la procédure de branchement et on rajoute les nouveaux sous-problèmes à la liste de sous-problèmes.

Procédure de branchement. Supposons que la solution optimale du problème linéairement relaxé, noté P, contienne une variable x_1 de valeur 5.7. On peut alors séparer le problème en deux sous-problèmes P_1 et P_2, dans P_1 on ajoutera la contrainte $x \leq 5$ et dans P_2, on ajoutera la contrainte $x \geq 6$. Comme expliqué précédemment, la meilleure solution trouvée en résolvant P_1 et P_2 sera la même que celle trouvée en résolvant directement P.

Dans le modèle utilisé à l'étape 6, des contraintes ont été ajoutées afin d'améliorer les performances. Ces contraintes, appelées *user cuts*, sont ajoutées uniquement dans le but d'accélérer la recherche et ne sont pas requises pour garantir l'optimalité de la solution trouvée. Elles permettent de réduire la taille de l'espace de recherche sans en retirer de solution réalisable. Celles qui ont été utilisées ici sont détaillées dans l'étape 5 de la section 3 de [17] (ceci n'est pas expliqué ici, car cette partie du solveur sert uniquement à améliorer ses performances).

L'algorithme qui a été présenté est une version assez générique de ce qui est utilisé dans les solveurs d'optimisation linéaire. Le solveur utilisé pour ce travail est *Gurobi* (voir leur site internet pour plus de détails : [21]).

Chapitre 6

Collaboration entre solveurs

6.1 Intérêt et travaux existants

Comme expliqué précédemment, il existe deux grands types d'approches pour résoudre le VRP : les méthodes exactes et les méthodes approchées. Les méthodes exactes ont l'avantage de donner la solution optimale du problème, mais au prix d'un temps d'exécution souvent très long pour les instances de grande taille. Les méthodes approchées permettent de résoudre ces instances de grande taille, mais la solution trouvée n'est pas nécessairement la solution optimale.

L'idée de réunir ces deux types de méthodes semble donc très intéressante pour créer des algorithmes plus performants. Dans [11], les auteurs proposent une approche permettant la collaboration de solveurs de problème de satisfaction de contraintes (CSP, Constraint Satisfaction Problem) et de la version avec optimisation de fonction objectif (CSOP, Constraint Satisfaction Optimization Problem). Leur approche utilise deux types de solveurs : un solveur aidé, noté *i-solver* et le solveur aidant, noté *c-solver*.

Une des techniques proposées est d'utiliser un solveur exact comme *i-solver* et un solveur approché comme *c-solver*. Lorsque *i-solver* se trouve bloqué, ne parvenant plus à trouver de meilleures solutions, il passe le contrôle à *c-solver*, qui utilisera une méthode approchée, telle que de la recherche locale pour trouver rapidement une nouvelle solution. Le but est que l'aide apportée par *c-solver* permette de réduire la taille de l'arbre de recherche en se focalisant sur les branches ayant une plus grande probabilité de contenir une solution optimale.

Dans [45], une autre méthode combinant méthode exacte et approchée est proposée : la recherche locale est utilisée pour accélérer un algorithme *branch and bound* au prix de la garantie d'optimalité. Au moment de la publication, cette méthode avait les meilleures performances pour certains CSP tels que le *n-queens problem*.

Il existe également d'autres approches hybrides qui ont été utilisées pour résoudre le VRP :

— [43] utilise l'optimisation par colonie de fourmis et un algorithme génétique
— [24] combine deux algorithmes de recherche locale : la recherche tabou et le recuit simulé
— [12] utilise l'optimisation par colonie de fourmis et le recuit simulé pour résoudre le VRP avec dépôts multiples et contrainte de fenêtre de temps
— [37] utilise une succession d'heuristiques constructives et perturbatives.

6.2 Approche choisie

L'approche qui a été choisie dans ce travail est une collaboration entre une méthode exacte, l'optimisation linéaire en nombres entiers et une méthode approchée, l'optimisation par colonie de fourmis.

Le rôle du solveur ACO sera de fournir des solutions de qualité acceptable rapidement au solveur ILP.

Le rôle du solveur ILP sera d'essayer de trouver la solution optimale du problème avec l'aide des solutions trouvées par le solveur ACO. Comme expliqué dans la section 5.2, un des points clés de l'algorithme *branch and bound* est l'étape d'évaluation. C'est dans cette partie que l'on pourra utiliser les solutions trouvées par le solveur ACO. Le coût de ces solutions pourra être utilisé comme borne de la fonction objectif, et lorsqu'un sous-problème ne pourra pas conduire à une solution de coût inférieur à cette borne, il ne sera pas nécessaire de le résoudre. Ceci devrait avoir pour effet de réduire la taille de l'arbre de recherche et donc d'accélérer le temps de résolution des instances du VRP.

6.3 Implémentation de la communication entre solveurs

6.3.1 Architecture

Dans cette section, on va présenter l'architecture qui est utilisée pour faire communiquer les solveurs entre eux durant les expériences. Sur le schéma suivant, on peut voir l'architecture globale :

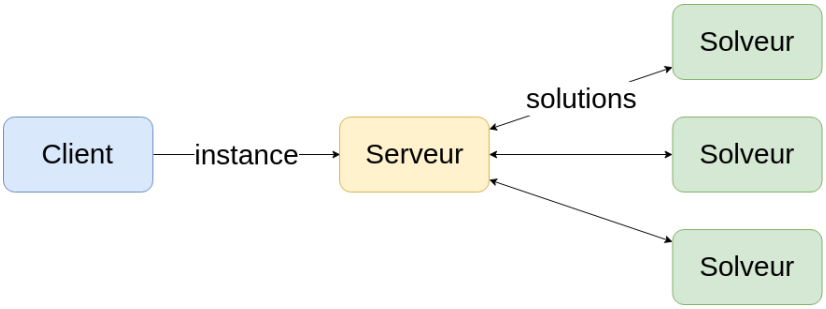


FIGURE 6.1 – Architecture du protocole de communication.

Le **client** est le programme qui servira à contrôler sur quelle instance les solveurs travaillent. Il envoie au serveur le nom de l'instance que les solveurs doivent résoudre et reçoit la solution optimale lorsque ceux-ci ont terminé. Si le programme est interrompu manuellement, la meilleure solution connue est renvoyée. Un seul client est autorisé.

Les **solveurs** correspondent à ceux présentés dans les chapitres précédents et qui ont été développés spécifiquement dans le cadre de ce travail pour permettre la collaboration. Sur le schéma, seul trois solveurs sont indiqués, mais en pratique, un nombre arbitraire de solveurs de chaque type peut être utilisé. Lorsque l'un de ceux-ci trouve une meilleure solution que la meilleure solution actuellement connue, il en informe le serveur qui pourra alors la transmettre aux autres solveurs. Les solveurs peuvent également être ajoutés dynamiquement, c'est-à-dire que l'on peut en connecter même lorsque la résolution d'une instance a déjà commencé. Les messages échangés ici sont uniquement des solutions complètes d'une instance de VRP, le but étant de rester générique afin de pouvoir utiliser ce protocole pour n'importe quel type de solveur du VRP.

Le **serveur** sert à établir la coordination entre les différents solveurs. C'est lui qui est chargé de propager les solutions connues et d'écouter le client pour déterminer quelle instance doit être résolue.

6.3.2 API

L'implémentation du protocole de communication est faite en utilisant l'API C++ de *gRPC* [19]. Cette technologie permet de faire des appels de procédure vers un serveur distant. Elle est utilisée ici pour définir l'API à laquelle les solveurs ont accès et pour gérer les connexions entre le serveur et les solveurs. Cette technologie utilise *Protocol Buffers* [20] comme langage de définition

d'interface, c'est-à-dire que tous les messages envoyés via le protocole de communication et l'API du serveur doivent être décrits en utilisant ce langage. Ceci offre l'avantage que des solveurs écrits dans des langages de programmation différents peuvent tout de même collaborer.

Voici l'API proposée par le serveur et les messages définis en utilisant *Protocol Buffers* :

```
service VRP {
    rpc solveInstance(Instance) returns (Empty) {}
    rpc getInstance(Empty) returns (Instance) {}
    rpc getBestSol(Empty) returns (Solution) {}
    rpc sendSol(Solution) returns (Empty) {}
}
message Empty {

}
message Instance {
    string instance_file = 1;
}
message Route {
    double cost = 1;
    int32 capacity = 2;
    repeated int32 route = 3;
}
message Solution {
    bool optimal = 1;
    double cost = 2;
    repeated Route routes = 3;
}
```

Quatre types de messages différents sont utilisés par le protocole :

- `Empty` définit un message vide, celui-ci est utilisé lorsqu'il n'est pas nécessaire d'envoyer de données au serveur, ou que l'on n'attend pas de réponse de sa part.
- `Instance` définit une instance, celle-ci est représentée par une chaîne de caractère correspondant au nom du fichier contenant les données de cette instance.
- `Route` définit l'itinéraire d'un véhicule. Il est représenté par trois champs : le coût associé, la capacité totale utilisée et la route en elle-même, elle est définie par la liste des clients desservis, chaque client est représenté par son indice.
- `Solution` définit une solution complète. Elle est représentée à nouveau par trois champs : une valeur booléenne indiquant si la solution est optimale ou non, le coût total et la liste des routes.

Le service `VRP` correspond au service fourni par le serveur. Celui-ci propose quatre fonctions différentes :

- `rpc solveInstance(Instance) returns (Empty)` est la fonction destinée au client. Elle lui permet d'indiquer quelle est l'instance qui doit être résolue.
- `rpc getInstance(Empty) returns (Instance)` est une fonction appelée régulièrement (toutes les 0,5 s) par les solveurs afin de déterminer quelle instance doit être résolue. Elle est appelée régulièrement pour détecter une éventuelle interruption de la résolution de l'instance demandée par le client ou parce que la solution optimale a été trouvée.
- `rpc getBestSol(Empty) returns (Solution)` est la fonction permettant aux solveurs et au client d'accéder à la meilleure solution actuellement connue.
- `rpc sendSol(Solution) returns (Empty)` est la fonction utilisée par un solveur pour envoyer une solution au serveur. Si cette solution est meilleure que celle actuellement connue par le serveur, celle-ci est mise à jour.

6.3.3 Utilisation en pratique

Pour démarrer la résolution d'une instance, le serveur est exécuté en premier et attend la connexion du client ou de solveurs. Les solveurs peuvent se connecter et se déconnecter à tout moment. Lorsque le client se connecte, il envoie l'instance qu'il souhaite résoudre au serveur. Les solveurs connectés s'en informent et commencent la résolution de cette instance. La meilleure solution connue est gardée en mémoire par le serveur, les solveurs connectés peuvent y accéder à tout moment. Lorsque la solution optimale a été trouvée, le serveur l'envoie au client, et celui-ci se déconnecte. Il peut se reconnecter s'il souhaite résoudre une autre instance.

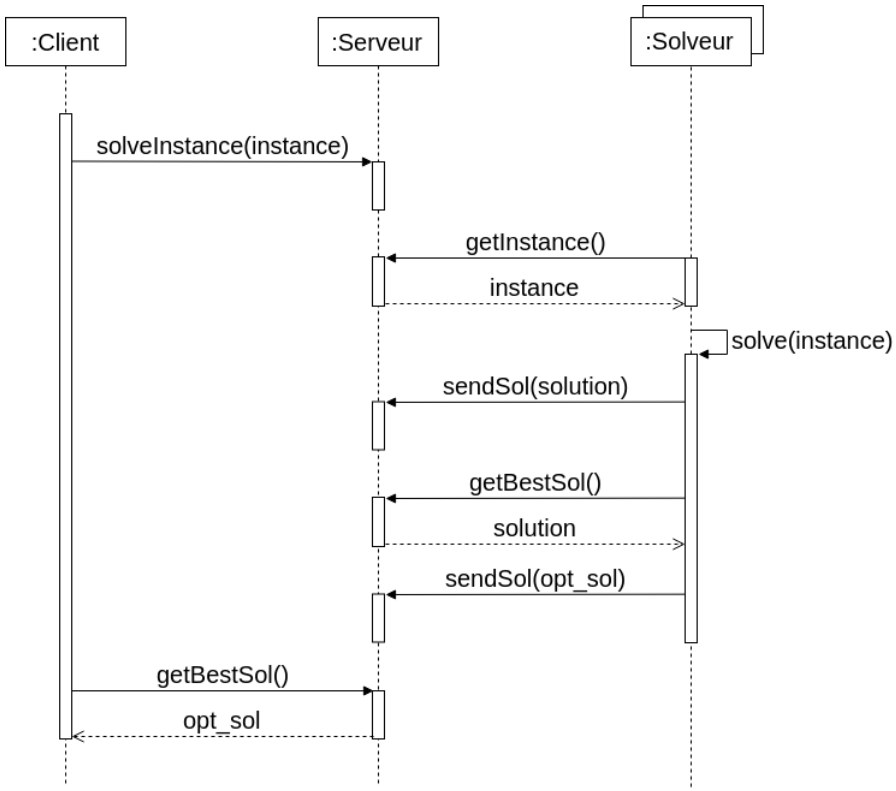


FIGURE 6.2 – Diagramme de séquence d'une résolution d'instance du VRP avec la collaboration entre solveurs.

Ce diagramme de séquence montre un exemple simplifié de résolution d'instance du VRP avec le protocole de communication. Dans cette exécution, la première solution envoyée par un des solveurs, notée "solution" dans le diagramme, est une solution non optimale. La seconde solution trouvée, notée "opt_sol" est la solution optimale pour l'instance. Les solveurs et le client sont donc arrêtés car l'instance est résolue.

6.3.4 Introduction de nouveaux solveurs au protocole

Pour pouvoir introduire un nouveau solveur dans le protocole de communication, il faut que ce solveur :

— puisse établir la connexion avec le serveur
— ait une fonction de conversion entre la représentation locale des solutions du VRP du solveur et la représentation définie avec *Protocol Buffers* présentée ci-dessus.

gRPC fournit des API pour les langages suivants : C, C++, Java, C#, Dart, Go, Ruby, Node.js, PHP et Python. N'importe quel solveur écrit dans un de ces langages peut faire appel à notre API et être donc utilisé pour la collaboration.

Prenons par exemple un solveur écrit en C++ que l'on voudrait connecter au protocole de communication. La marche à suivre pour des solveurs écrits dans d'autres langages est la même, à quelques spécificités du langage près.

La première chose à faire est de compiler le fichier *Protocol Buffers* contenant la description du service et des messages, c'est le fichier contenant la définition de l'API présentée à la section précédente. Cette compilation permet de générer les fichiers nécessaires au fonctionnement de *gRPC* et ceux-ci doivent être importés par le solveur. Ces fichiers contiendront les définitions des classes `gRPC_Solution`, `gRPC_Route`, `gRPC_Instance` et `gRPC_Empty`, ce qui permettra de manipuler le contenu des messages envoyés et reçus avec *gRPC*.

Il faut ensuite établir la connexion avec le serveur. En C++, cette ligne de code permet de le faire :

```
std::unique_ptr<VRP::Stub> stub_(VRP::NewStub(grpc::CreateChannel(
server_address, grpc::InsecureChannelCredentials())));
```

Il faut ensuite créer les fonctions de conversion des solutions, en C++, une représentation possible d'une solution du VRP est donnée par :

```
class Route{
    std::vector<int> route;
    double cost;
    int capacity;
}

class Solution{
    std::vector<Route> routes;
    double cost;
}
```

Les routes sont représentées par la liste ordonnée des clients desservis.

Les fonctions de conversion de cette représentation vers la représentation *Protocol Buffers* et vice-versa sont alors :

```
gRPC_Solution convertTogRPC(Solution sol, bool optimal){
    gRPC_Solution gsol;
    gsol.set_optimal(optimal);
    gsol.set_cost(sol.getCost());
    vector<Route> routes = sol.getRoutes();
    for (int i = 0;i < routes.size();i++){
        gRPC_Route* gr = gsol.add_routes();
        gr->set_cost(routes[i].getCost());
        gr->set_capacity(routes[i].getCapacity());
        vector<int> route = routes[i].getRoute();
        for (int j = 0;j < route.size();j++){
            gr->add_route(route[j]);
        }
    }
    return gsol;
}

Solution convertToLocal(gRPC_Solution gsol){
    vector<Route> routes;
    for (gRPC_Route gr : gsol.routes()){
        vector<int> route;
```

```
        for (int node : gr.route()){
            route.push_back(node);
        }
        Route r (route);
        routes.push_back(r);
    }
    Solution sol (routes, gsol.cost());
    return sol;
}
```

`gRPC_Solution` correspond à la représentation *Protocol Buffers* d'une solution du VRP tandis que `Solution` est la représentation utilisée par le solveur.

Finalement, on peut faire appel à notre API. Par exemple pour demander au serveur quelle instance doit être résolue (la variable `_stub` est celle créée lors de l'établissement de la connexion avec le serveur) :

```
gRPC_Empty empty;
gRPC_Instance inst;
ClientContext context;
stub_->getInstance(&context, empty, &inst);
std::string instance_file = inst.instance_file();
```

Pour envoyer une solution au serveur :

```
Solution sol = ...;
bool optimal = ...;
gRPC_Solution gsol = convertTogRPC(sol, optimal);
gRPC_Empty empty;
ClientContext context;
stub_->sendSol(&context, gsol, &empty);
```

Pour demander la meilleure solution connue au serveur :

```
gRPC_Solution gsol;
gRPC_Empty empty;
ClientContext context;
stub_->getBestSol(&context, empty, &gsol);
Solution sol = convertToLocal(gsol);
```

Chapitre 7

Expériences et résultats

7.1 Expériences

Dans cette section, nous allons faire des expériences en utilisant les solveurs pour résoudre des instances de différentes tailles afin de mesurer leurs performances. Les fichiers contenant ces instances peuvent être trouvés sur le site web [1]. L'intérêt de ces expériences n'est pas les performances individuelles des solveurs, mais bien la comparaison entre les performances avec ou sans collaboration. Chaque expérience a été répétée 5 fois. Les temps mesurés sont des temps réels et la machine n'était pas utilisée durant les expériences. Le processeur utilisé est un Intel Core i7 à 2,4 GHz.

Dans les tableaux de résultats, la colonne "instance" fait référence au nom du fichier contenant l'instance. Les colonnes n, K et C sont respectivement le nombre de nœuds, le nombre de véhicules à utiliser et la capacité des véhicules.

7.1.1 Solveur par optimisation par colonie de fourmis

Pour mesurer les performances de ce solveur, la métrique utilisée sera le coût de la meilleure solution obtenue à la fin de l'exécution. Les paramètres du solveur utilisés sont :

- $n = 10000$, le nombre d'itérations
- $n_{restarts} = 50$, le nombre de redémarrage
- $\rho = 0.95$, le facteur d'évaporation
- $p = 0.25$, la proportion de clients considérés à chaque sélection d'une nouvelle destination
- $\alpha = \beta = 5$, indiquant la pondération entre les pistes de phéromones et la visibilité (voir équation 4.2)
- $Q = 1000000$, paramètre fixant l'échelle et n'ayant pas d'impact sur les performances.

Pour chaque instance testée, nous reporterons le meilleur, la moyenne et le pire coût obtenu après les 5 exécutions. Le coût de la solution optimale est également indiqué à titre indicatif. Le temps d'exécution moyen est aussi indiqué.

Ce solveur est implémenté en C++.

instance	n	K	C	optimal	meilleur	moyenne	pire	temps (s)
A-n32-k5	32	5	100	787,08	787,08	787,08	787,08	94,13
A-n36-k5	36	5	100	802,13	802,13	807,57	811,04	78,91
A-n37-k6	37	6	100	950,85	950,85	959,20	967,51	85,82
A-n38-k5	38	5	100	733,94	737,07	737,46	737,56	77,80
A-n44-k7	44	7	100	938,18	953,11	955,88	963,12	111,97
A-n45-k6	45	6	100	944,88	950,22	956,27	960,69	94,66
A-n46-k7	46	7	100	917,72	925,23	925,65	925,80	156,78
B-n43-k6	43	6	100	746,69	749,03	750,60	751,79	96,07
B-n45-k5	45	5	100	753,96	770,75	770,75	770,75	97,49
B-n52-k7	52	7	100	749,97	783,47	783,53	783,65	149,77
B-n56-k7	56	7	100	712,92	734,99	737,20	740,35	317,10
E-n51-k5	51	5	160	524,61	524,81	529,28	536,02	141,06
P-n45-k5	45	5	150	512,79	512,79	512,79	512,79	118,60
P-n50-k7	50	7	150	559,86	562,64	564,64	565,68	160,14
P-n55-k7	55	7	150	570,27	579,96	584,43	589,44	212,15

TABLE 7.1 – Résultats des tests de performance sur 15 instances différentes du VRP pour le solveur par optimisation par colonie de fourmis. Paramètres utilisés : $n = 10000$, $n_{restarts} = 50$, $\rho = 0.95$, $p = 0.25$, $\alpha = \beta = 5$, $Q = 1000000$.

La table 7.1 montre que ce solveur est en mesure de fournir des solutions de qualité acceptable en un temps raisonnable, ce qui est le rôle de ce solveur dans le cadre de la collaboration entre solveurs. Pour l'instance B-n56-k7, $p = 0,5$ a été utilisé car la contrainte de capacité des véhicules était plus difficile à satisfaire, il fallait donc considérer un plus grand nombre de clients à chaque sélection de nouvelle destination pour la respecter.

7.1.2 Solveur par optimisation linéaire en nombres entiers

Pour mesurer les performances de ce solveur, nous utiliserons ici le temps d'exécution plutôt que le coût de la meilleure solution, puisque celui-ci trouve toujours la solution optimale (si l'exécution se termine sans interruption). Le temps mesuré est donc celui pris pour trouver la solution optimale. Nous reporterons le meilleur, la moyenne et le pire temps des 5 exécutions. Celui-ci est exprimé en secondes.

Ce solveur est implémenté en utilisant l'API C++ de *Gurobi*.

instance	n	K	C	meilleur	moyenne	pire
A-n32-k5	32	5	100	5,42	5,42	5,43
A-n36-k5	36	5	100	16,50	16,63	16,98
A-n37-k6	37	6	100	498,77	503,42	509,46
A-n38-k5	38	5	100	20,88	20,95	21,04
A-n44-k7	44	7	100	518,72	524,29	528,81
A-n45-k6	45	6	100	318,24	320,82	322,15
A-n46-k7	46	7	100	171,10	171,67	172,58
B-n43-k6	43	6	100	114,64	115,31	115,76
B-n45-k5	45	5	100	60,82	61,07	61,31
B-n52-k7	52	7	100	20,37	20,42	20,46
B-n56-k7	56	7	100	77,57	78,01	78,46
E-n51-k5	51	5	160	68,34	68,87	69,13
P-n45-k5	45	5	150	17,26	17,32	17,36
P-n50-k7	50	7	150	112,27	113,00	113,73
P-n55-k7	55	7	150	5298,70	5346,39	5412,88

TABLE 7.2 – Résultats des temps d'exécution (en secondes) pour la résolution de 15 instances différentes du VRP par le solveur par optimisation linéaire en nombres entiers.

Les résultats de la table 7.2 seront utilisées pour faire la comparaison avec les performances obtenues lors de la collaboration entre solveurs.

7.1.3 Collaboration entre solveurs

Ici aussi, c'est le temps d'exécution qui sera utilisé comme métrique de performances. Nous mesurons à nouveau le meilleur, la moyenne et le pire temps obtenus après les 5 exécutions.

Pour notre première configuration, nous utilisons un solveur ILP et trois solveurs ACO. Ces valeurs semblent adaptées car l'ordinateur utilisé pour les tests de performance a un processeur à 4 cœurs. Cette configuration correspond donc à ceci :

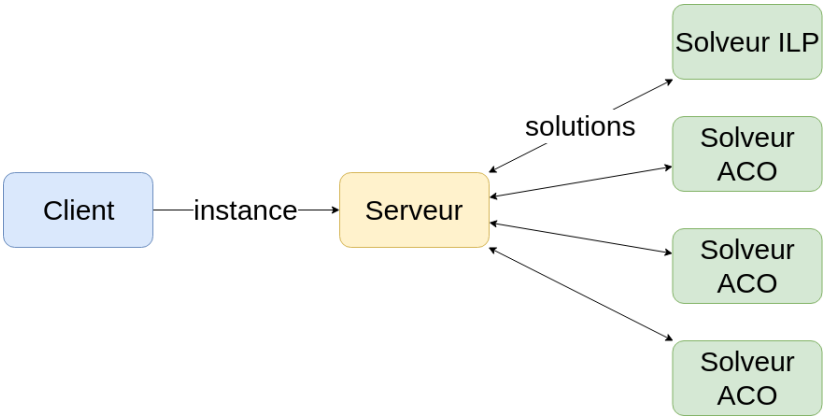


FIGURE 7.1 – Première configuration utilisée pour les expériences de collaboration entre solveurs : un solveur ILP et trois solveurs ACO.

Les paramètres utilisés pour les solveurs ACO sont les mêmes que ceux utilisés pour tester ses performances. Lorsque le nombre d'itérations maximal est atteint, le solveur redémarre en

réinitialisant ses variables contenant les valeurs de phéromones.

instance	n	K	C	meilleur	moyenne	pire
A-n32-k5	32	5	100	3,42	7,76	10,14
A-n36-k5	36	5	100	11,91	16,66	28,06
A-n37-k6	37	6	100	398,99	628,98	1000,30
A-n38-k5	38	5	100	7,19	10,26	15,75
A-n44-k7	44	7	100	260,14	377,98	561,31
A-n45-k6	45	6	100	254,11	441,33	1053,99
A-n46-k7	46	7	100	88,75	281,92	615,52
B-n43-k6	43	6	100	54,54	79,43	108,78
B-n45-k5	45	5	100	18,80	30,94	50,55
B-n52-k7	52	7	100	7,90	10,59	12,83
B-n56-k7	56	7	100	28,54	51,88	78,28
E-n51-k5	51	5	160	28,26	37,22	46,62
P-n45-k5	45	5	150	11,02	18,57	35,72
P-n50-k7	50	7	150	82,88	129,38	188,96
P-n55-k7	55	7	150	5194,05	6694,54	9001,84

TABLE 7.3 – Résultats des tests de performance lors de la collaboration entre solveurs pour la résolution de 15 instances différentes du VRP. Configuration utilisée : un solveur ILP et trois solveurs ACO.

Nous pouvons voir dans les résultats de la table 7.3 que la variance du temps d'exécution est bien plus grande que lorsque nous utilisions le solveur ILP sans collaboration. Cette apparition d'une telle variance est très probablement due au fait que sans la collaboration, ce solveur n'utilise pas de processus aléatoire tandis qu'avec la collaboration, de l'aléatoire est introduit indirectement par les solveurs ACO qui en utilise beaucoup dans leur algorithme.

Pour mesurer l'impact de la collaboration, faisons une comparaison avec les performances obtenues par le solveur ILP sans collaboration. Nous noterons que cette comparaison n'est pas parfaite, puisque dans un cas, un seul solveur utilisait les ressources, tandis que dans le cas de la collaboration, elles étaient partagées entre quatre solveurs.

La comparaison est faite en prenant les performances du solveur ILP sans collaboration pour référence. Les valeurs sont indiquées en pourcentage, donc si la valeur est inférieure à 100%, cela correspond à une amélioration des performances, elles sont marquées en gras.

instance	n	K	C	meilleur	moyenne	pire
A-n32-k5	32	5	100	**63,1**%	143,2%	186,7%
A-n36-k5	36	5	100	**72,2**%	100,2%	165,2%
A-n37-k6	37	6	100	**80,0**%	124,9%	196,3%
A-n38-k5	38	5	100	**34,4**%	**49,0**%	**74,9**%
A-n44-k7	44	7	100	**50,1**%	**72,1**%	106,2%
A-n45-k6	45	6	100	**79,8**%	137,6%	327,2%
A-n46-k7	46	7	100	**51,9**%	164,2%	356,7%
B-n43-k6	43	6	100	**47,6**%	**68,9**%	**94,0**%
B-n45-k5	45	5	100	**30,9**%	**50,7**%	**82,5**%
B-n52-k7	52	7	100	**38,8**%	**51,9**%	**62,7**%
B-n56-k7	56	7	100	**36,8**%	**66,5**%	**99,8**%
E-n51-k5	51	5	160	**41,4**%	**54,0**%	**67,4**%
P-n45-k5	45	5	150	**63,9**%	107,2%	205,8%
P-n50-k7	50	7	150	**73,8**%	114,5%	166,2%
P-n55-k7	55	7	150	**98,0**%	125,2%	166,3%

TABLE 7.4 – Comparaison entre les tests de performance du solveur ILP et ceux obtenus lors de la collaboration entre solveurs pour la résolution de 15 instances différentes du VRP. Configuration utilisée lors de la collaboration entre solveurs : un solveur ILP et trois solveurs ACO.

Comme nous pouvons le voir dans la table 7.4, la collaboration a globalement un impact mitigé sur les performances. Les performances obtenues dans le meilleur des cas sont toujours améliorées. Celles obtenues dans le cas moyen sont parfois améliorées, et de même pour le pire cas.

Il semblerait que la collaboration avec cette configuration soit la plus effective pour des instances étant résolues en un temps modéré, entre 10 secondes et 1 minute. Une interprétation possible est que dans le cas d'instances faciles à résoudre, la collaboration n'a pas le temps d'être efficace et n'apporte donc rien. Dans le cas d'instances difficiles, il est très probable qu'au bout d'un long temps d'exécution les solveurs ACO ne parviennent plus à améliorer leur meilleure solution alors qu'ils consomment encore beaucoup de ressources.

Testons maintenant une deuxième configuration qui devrait être plus adaptée pour les instances plus difficiles à résoudre (celles prenant au minimum 60 secondes pour être résolue sans collaboration). Cette configuration consiste à utiliser quatre solveurs ACO pendant 10 secondes afin de trouver rapidement une bonne solution, puis les déconnecter et laisser place au solveur ILP. On utilise donc cette configuration pendant 10 secondes :

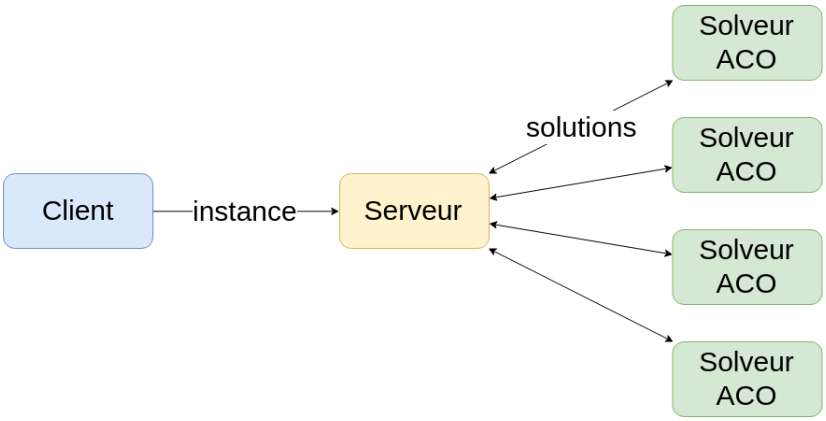


FIGURE 7.2 – Première partie de la deuxième configuration utilisée pour les expériences de collaboration entre solveurs : quatre solveurs ACO pendant 10 secondes.

Puis cette configuration jusqu'à la fin de la résolution :

FIGURE 7.3 – Deuxième partie de la deuxième configuration utilisée pour les expériences de collaboration entre solveurs : un solveur ILP jusqu'à résolution de l'instance.

Le but de cette configuration est donc de laisser du temps aux solveurs ACO de trouver une bonne solution pour que le solveur ILP puisse ensuite s'en servir comme borne. Les temps d'exécution reportés sont la somme des 10 secondes utilisées par les solveurs ACO et le temps d'exécution de la résolution avec le solveur ILP.

instance	n	K	C	meilleur	moyenne	pire
A-n37-k6	37	6	100	219,83	317,92	505,12
A-n44-k7	44	7	100	215,03	308,53	364,74
A-n45-k6	45	6	100	277,96	326,46	371,76
A-n46-k7	46	7	100	213,09	334,43	444,78
B-n43-k6	43	6	100	65,30	75,87	89,83
B-n45-k5	45	5	100	31,51	43,28	73,78
B-n56-k7	56	7	100	41,61	69,63	109,53
E-n51-k5	51	5	160	44,20	55,28	68,02
P-n50-k7	50	7	150	85,18	100,14	113,9
P-n55-k7	55	7	150	2029,71	3825,03	5521,26

TABLE 7.5 – Résultats des tests de performance lors de la collaboration entre solveurs pour la résolution de 10 instances différentes du VRP. Configuration utilisée : quatre solveurs ACO pendant 10 secondes puis un solveur ILP.

On peut voir dans la table 7.5 que la variance dans les temps d'exécution est également présente, pour la même raison que dans notre première configuration. Faisons maintenant à nouveau la comparaison avec les résultats obtenus pour le solveur ILP sans collaboration.

instance	n	K	C	meilleur	moyenne	pire
A-n37-k6	37	6	100	**44,1**%	**63,1**%	**99,2**%
A-n44-k7	44	7	100	**41,5**%	**58,9**%	**69,0**%
A-n45-k6	45	6	100	**87,3**%	101,8%	115,4%
A-n46-k7	46	7	100	124,5%	194,8%	257,7%
B-n43-k6	43	6	100	**57,0**%	**65,8**%	**77,6**%
B-n45-k5	45	5	100	**51,8**%	**70,9**%	120,3%
B-n56-k7	56	7	100	**53,6**%	**89,3**%	139,6%
E-n51-k5	51	5	160	**64,7**%	**80,3**%	**98,4**%
P-n50-k7	50	7	150	**75,9**%	**88,6**%	100,2%
P-n55-k7	55	7	150	**38,3**%	**71,5**%	102,0%

TABLE 7.6 – Comparaison entre les tests de performance du solveur ILP et ceux obtenus lors de la collaboration entre solveurs pour la résolution de 10 instances différentes du VRP. Configuration utilisée lors de la collaboration entre solveurs : quatre solveurs ACO pendant 10 secondes puis un solveur ILP.

Les résultats de la table 7.6 montrent une amélioration des performances pour certaines instances et une dégradation pour d'autres, par rapport à ceux de la table 7.4. On notera aussi que cette configuration est bien moins gourmande en ressources, puisque la majorité du temps d'exécution se déroule avec un seul solveur actif.

Cette deuxième expérience montre que le choix de la configuration utilisée pour la collaboration entre solveurs est un élément capital pour obtenir de bonnes performances, certaines configurations sont mieux adaptées que d'autres pour certaines instances.

7.2 Conclusion et recommandations

Ces expériences ont permis de montrer que la collaboration entre les solveurs obtient de bons résultats mais nécessite peut-être un raffinement pour être plus performante. Je recommanderais de développer un algorithme dédié à la gestion de la configuration des solveurs utilisés. Pour ces expériences, la configuration était gérée manuellement par l'utilisateur. Un algorithme dédié pourrait quant à lui gérer quand tel ou tel solveur doit être utilisé ou éteint. En début d'exécution, il pourrait par exemple lancer un grand nombre de solveurs trouvant des bonnes solutions rapidement, tel que des solveurs par recherche locale ou des solveurs ACO. Il pourrait ensuite les éteindre au fur et à mesure afin de libérer des ressources pour le solveur ILP.

Chapitre 8

Conclusion

Dans ce travail, nous avons vu en quoi consistait le problème de tournées de véhicules, comment celui-ci a été étendu à de nombreuses variantes pour tenir compte des contraintes de la vie réelle et les différentes approches qui sont utilisées pour le résoudre. Un état de l'art de ces différentes méthodes a été fait afin de voir quelles techniques étaient les mieux adaptées pour résoudre ce problème. Parmi celles-ci, deux ont été retenues et ont été implémentés pour être utilisées lors de la collaboration entre solveurs.

Le premier solveur implémenté, le solveur par optimisation par colonie de fourmis, a permis de montrer comment le comportement d'une colonie de fourmis pouvait être exploité pour développer un algorithme d'optimisation. Le rôle de ce solveur a été de fournir de bonnes solutions aux autres solveurs lors de la collaboration, les tests de performance ont pu montrer qu'il remplissait bien son rôle.

Ensuite, nous avons vu un deuxième type de solveur, le solveur par optimisation linéaire en nombres entiers. Celui-ci se base sur le modèle mathématique du problème pour le résoudre. Ceci a permis de montrer comment un algorithme de résolution de problème ILP fonctionnait et comment il pouvait être utilisé pour résoudre le VRP. Le rôle de ce solveur était de résoudre l'instance du problème optimalement en utilisant les solutions de bonne qualité fournies par les autres solveurs.

Enfin, nous avons fait collaborer ces deux différents solveurs entre eux. Les expériences ont pu montrer que l'impact de cette collaboration était globalement positif, mais que cela introduisait aussi une grande variance dans les résultats, en ayant parfois des temps d'exécution presque doublés d'une exécution à une autre. Ces résultats semblent indiquer qu'il y a un réel intérêt à faire collaborer différents types de solveurs entre eux.

Améliorations possibles

Une amélioration possible de ce travail aurait été de tester la collaboration entre solveurs avec plus de types de solveurs différents. Ici, seul deux types différents ont été utilisés. Des solveurs basés sur la recherche locale, tels que la recherche tabou ou encore le recuit simulé ont montré qu'ils avaient de très bonnes performances pour résoudre approximativement le VRP. Ceci permettrait probablement d'avoir des performances plus homogènes, puisque certaines méthodes fonctionneront mieux sur certaines instances que d'autres.

Une autre amélioration serait d'explorer d'autres types de collaboration entre les solveurs. Pour le solveur par optimisation par colonie de fourmis, on pourrait par exemple utiliser les solutions fournies par les autres solveurs pour renforcer les pistes de phéromones.

Finalement, le protocole de communication pourrait être amélioré. A l'heure actuelle, les appels de fonction faits à l'API du serveur sont synchrones alors que des appels asynchrones seraient plus adaptés car cela réduirait l'impact de l'utilisation du protocole sur les performances. En particulier si on souhaite faire fonctionner des solveurs sur une machine différente de celle où est exécuté le serveur car il y aurait de la latence introduite par le réseau.

Annexe A

Guide d'utilisation

Tous les fichiers exécutables sont regroupés dans le dossier *bin*. Il est nécessaire d'utiliser ce dossier comme dossier courant lors de l'exécution des fichiers, car les chemins sont relatifs à ce dossier.

A.1 Solveur par optimisation par colonie de fourmis

Le fichier exécutable est `aco_solver`, les options disponibles sont :

`-h` pour afficher l'aide.

`-l` pour utiliser le solveur sans collaboration, il est alors nécessaire de fournir le nom du fichier en argument.

`-a arg` pour modifier la valeur de α à `arg`. Par défaut, $\alpha = 5$.

`-b arg` pour modifier la valeur de β à `arg`. Par défaut, $\beta = 5$.

`-n arg` pour modifier le nombre d'itérations à `arg`. Par défaut, $n = 10000$.

`-p arg` pour modifier à `arg` la proportion de clients considérés à chaque sélection d'une nouvelle destination, une valeur entre 0 et 1 est donc requise. Par défaut, $p = 0.25$.

`-q arg` pour modifier le facteur d'échelle. Par défaut, $Q = 1000000$.

`-r arg` pour modifier le facteur d'évaporation des phéromones. Par défaut, $\rho = 0.95$.

Exemples d'utilisation :

Pour résoudre l'instance contenue dans le fichier `A-n37-k6.vrp` sans collaboration et en utilisant les paramètres par défaut :

```
$ ./aco_solver -l A-n37-k6.vrp
```

Pour utiliser le solveur avec collaboration et en modifiant les valeurs de α, β et ρ :

```
$ ./aco_solver -a 4 -b 4 -p 0.5
```

Le serveur doit être lancé au préalable pour ce mode de fonctionnement.

A.2 Solveur par optimisation linéaire en nombres entiers

Le fichier exécutable est `ilp_solver`, les options disponibles sont :

`-h` pour afficher l'aide.

`-l` pour utiliser le solveur sans collaboration, il est alors nécessaire de fournir le nom du fichier en argument.

Exemple d'utilisations :

Pour résoudre l'instance contenue dans le fichier `A-n37-k6.vrp` sans collaboration :

```
$ ./ilp_solver -l A-n37-k6.vrp
```

Pour utiliser le solveur avec collaboration :

```
$ ./ilp_solver
```

Le serveur doit être lancé au préalable pour ce mode de fonctionnement.

A.3 Serveur

Le fichier exécutable est `server`. Pour le lancer, rien de plus simple :

```
$ ./server
```

Il doit être exécuté en premier lorsque l'on souhaite utiliser la collaboration entre solveurs, afin de leur permettre de s'y connecter.

A.4 Client

Le fichier exécutable est `client`. Pour démarrer la résolution d'une instance, il suffit de passer son nom de fichier en argument, par exemple :

```
$ ./client A-n37-k6.vrp
```

Il doit être exécuté après avoir démarré le serveur.

Bibliographie

[1] URL : http://neo.lcc.uma.es/vrp/vrp-instances/capacitated-vrp-instances/ (visité le 27/07/2018).

[2] Imran A., Salhi S. et Wassan N.A. « A variable neighborhood-based heuristic for the heterogeneous fleet vehicle routing problem ». In : *European Journal of Operational Research* 197.2 (2009), p. 509–518.

[3] Lim A. et Wang F. « Multi-depot vehicle routing problem : a one-stage approach ». In : *IEEE Transactions on Automation Science and Engineering* 2.4 (2000), p. 397–402.

[4] Lim A. et Wang F. « The Multi-depot Periodic Vehicle Routing Problem ». In : *Lecture Notes in Computer Science* 3607 (2005), p. 347–350.

[5] Wren A. et Holliday A. « Computer Scheduling of Vehicles from One or More Depots to a Number of Delivery Points ». In : *Journal of the Operational Research Society* 23.3 (1972), p. 333–344.

[6] *Algorithme de colonies de fourmis*. URL : https://fr.wikipedia.org/wiki/Algorithme_de_colonies_de_fourmis (visité le 25/02/2018).

[7] Bullnheimer B., Hartl R. et Strauss C. « An improved Ant System algorithm for the Vehicle Routing Problem ». In : *Annals of Operations Research* 89 (1999), p. 319–328.

[8] Korte B. et Vygen J. « Bin-Packing ». In : *Algorithms and Combinatorics* 21 (2006), p. 426–441.

[9] Yu B., Yang Z.-Z. et Yao B. « An improved ant colony optimization for vehicle routing problem ». In : *European Journal of Operational Research* 196.1 (2009), p. 171–176.

[10] *Bin Packing*. 2011. URL : http://yetanothermathprogrammingconsultant.blogspot.be/2011/08/bin-packing.html (visité le 14/03/2018).

[11] Castro C. et Riff M.C. « Switching among Solvers : Collaborative Algorithms with Parameter Control ». In : *Optimization and Cooperative Control Strategies* 381 (2009), p. 287–298.

[12] Ting C.J. et Chen C.H. « Combination of Multiple Ant Colony System and Simulated Annealing for the Multidepot Vehicle-Routing Problem with Time Windows ». In : *Transportation Research Record* 2089.1 (2008), p. 85–92.

[13] *Convex hull*. URL : https://en.wikipedia.org/wiki/Convex_hull (visité le 10/05/2018).

[14] Paraskevopoulos D.C. et al. « A reactive variable neighborhood tabu search for the heterogeneous fleet vehicle routing problem with time windows ». In : *Journal of Heuristics* 14.5 (2008), p. 425–455.

[15] Clarke G. et Wright J.W. « Scheduling of Vehicles from a Central Depot to a Number of Delivery Points ». In : *Operations Research* 12.4 (1964), p. 568–581.

[16] Laporte G. « A concise guide to the Traveling Salesman Problem ». In : *Journal of the Operational Research Society* 61.1 (2010), p. 35–40.

[17] Laporte G., Nobert Y. et Desrochers M. « Optimal Routing under Capacity and Distance Restrictions ». In : *Operations Research* 33.5 (1985), p. 1050–1073.

[18] Dantzig G.B. et Ramser J.H. « The Truck Dispatching Problem ». In : *Management Science* 6.1 (1959), p. 80–91.

[19] Google. *gRPC*. URL : `https://grpc.io/`.

[20] Google. *Protocol Buffers*. URL : `https://developers.google.com/protocol-buffers/`.

[21] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2018. URL : `http://www.gurobi.com`.

[22] Li H. et Lim A. « Local search with annealing-like restarts to solve the VRPTW ». In : *European Journal of Operational Research* 150 (2003), p. 115–127.

[23] Paessens H. « The savings algorithm for the vehicle routing problem ». In : *European Journal of Operational Research* 34.3 (1988), p. 336–344.

[24] Osman I.H. « Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem ». In : *Annals of Operations Research* 41.4 (1993), p. 421–451.

[25] *Integer programming*. URL : `https://en.wikipedia.org/wiki/Integer_programming` (visité le 07/05/2018).

[26] Brandão J. « A tabu search algorithm for the heterogeneous fixed fleet vehicle routing problem ». In : *Computers & Operations Research* 38.1 (2011), p. 140–151.

[27] Bell J.E. et McMullen P.R. « Ant colony optimization techniques for the vehicle routing problem ». In : *Advanced Engineering Informatics* 18.1 (2004), p. 41–48.

[28] Braekers K., Ramaekers K. et Van Nieuwenhuyse I. « The vehicle routing problem : State of the art classification and review ». In : *Computers & Industrial Engineering* 99 (2016), p. 300–313.

[29] Murty K.G. *Linear Programming*. Wiley, 1983.

[30] Zhu K.Q. « A New Genetic Algorithm For VRPTW ». In : *Proceedings of the International Conference on Artificial Intelligence* (2000).

[31] Dorigo M. *Ant colony optimization*. 2007. URL : `http://www.scholarpedia.org/article/Ant_colony_optimization` (visité le 25/02/2018).

[32] Dorigo M. et Stützle T. *Ant Colony Optimization*. MIT Press, 2004, p. 155–159.

[33] Gendreau M., Hertz A. et Laporte G. « A Tabu Search Heuristic for the Vehicle Routing Problem ». In : *Management Science* 40.10 (1994), p. 562–579.

[34] Azi N., Gendreau M. et Potvin J.Y. « An exact algorithm for a vehicle routing problem with time windows and multiple use of vehicles ». In : *European Journal of Operational Research* 202 (2010), p. 756–763.

[35] Christofides N., Mingozzi A. et Toth P. « Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations ». In : *Mathematical Programming* 20.1 (1981), p. 255–282.

[36] NEO. 2013. URL : `http://neo.lcc.uma.es/vrp/` (visité le 21/02/2018).

[37] Garrido P. et Castro C. « Stable solving of CVRPs using hyperheuristics ». In : *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* 115 (2009), p. 255–262.

[38] Toth P. et Vigo D. *The Vehicle Routing Problem*. Society for Industrial et Applied Mathematics, 2001.

[39] *Problème de bin packing*. URL : `https://fr.wikipedia.org/wiki/Probl%C3%A8me_de_bin_packing` (visité le 14/03/2018).

[40] *Problème du voyageur de commerce.* URL : https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_voyageur_de_commerce (visité le 13/03/2018).

[41] Chen Q., Li K. et Liu Z. « Model and algorithm for an unpaired pickup and delivery vehicle routing problem with split loads ». In : *Transportation Research Part E : Logistics and Transportation Review* 69 (2014), p. 218–235.

[42] Fukasawa R., Longo H., Lysgaard J. et al. « Robust Branch-and-Cut-and-Price for the Capacitated Vehicle Routing Problem ». In : *Mathematical Programming* 106.3 (2006), p. 491–511.

[43] Kuo R.J. et Zulvia F.E. « Hybrid genetic ant colony optimization algorithm for capacitated vehicle routing problem with fuzzy demand — A case study on garbage collection system ». In : *2017 4th International Conference on Industrial Engineering and Applications (ICIEA)* (2017), p. 244–248.

[44] Lin S. et Kernighan B.W. « An Effective Heuristic Algorithm for the Traveling-Salesman Problem ». In : *Operations Research* 21.2 (1973), p. 498–516.

[45] Prestwich S. « Combining the Scalability of Local Search with the Pruning Techniques of Systematic Search ». In : *Annals of Operations Research* 115 (2002), p. 51–72.

[46] Russel S. et Norvig P. *Artificial Intelligence : A Modern Approach.* 3e éd. Pearson Education, 2003.

[47] Yanik S., Bozkaya B. et deKervenoael R. « A new VRPPD model and a hybrid heuristic solution approach for e-tailing ». In : *European Journal of Operational Research* 236.3 (2014), p. 879–890.

[48] *Simplex algorithm.* URL : https://en.wikipedia.org/wiki/Simplex_algorithm (visité le 29/06/2018).

[49] *Vehicle Routing Problem : Is There A Perfect Solution ?* 2014. URL : https://logisticsmgepsupv.wordpress.com/2014/04/10/vehicle-routing-problem-its-there-a-perfect-solution/ (visité le 21/02/2018).

[50] Yiyong X. et al. « Variable neighbourhood simulated annealing algorithm for capacitated vehicle routing problems ». In : *Engineering Optimization* 46.4 (2013), p. 562–579.

[51] Deville Y. *LINGI1123 - Calculabilité et Complexité : Notes de cours.* 2011-2012.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**