le cnam

Conservatoire National des Arts et Métiers

Recherche Opérationnelle et Réseaux de Transport

Projet de tournée de véhicules électriques

Louise Fournon, Camille Grange & Mathieu Lerouge

Etudiants au Master MPRO

Enseignants : Alain Faye & Dimitri Watel

Janvier 2020 - Mars 2020

Table des matières

Introduction

Modélisation

Données du problème

A l'instar de Schneider et al., nous formulons le problème au moyen des données suivantes :

- $G = (V_{0,n+1}, A)$ un graphe orienté avec $m = |A|$ le nombre d'arcs ;
- $0 \in V_{0,n+1}$ et $n+1 \in V_{0,n+1}$ les sommets dépôts respectivement départ et arrivée ;
- $I \subset V_{0,n+1}$ un ensemble d'indices des clients et $F \subset V_{0,n+1}$ un ensemble d'indices des stations ;
- $\{d_a\}$ et $\{t_a\}$ des distances et temps de parcours des arcs du graphe.

Notation

Nous utilisons la notation suivante :

- MILP (*Mixed Integer Linear Program*) pour programme linéaire mixte ;
- DP (*Dynamic Programming*) pour programmation dynamique.

Hypothèses

A la différence de l'article de Schneider et al., nous faisons les hypothèses suivantes :

- Le graphe G n'est pas nécessairement complet contrairement à ce qui est supposé dans l'article (nous avons fait cette hypothèse lorsque nous avons vu que les instances *E_data* n'était pas complète, nous allons voir que cela complique le problème) ;
- Les demandes des clients notées, $\{q_i\}_{i \in I}$ dans l'article de Schneider et al., sont nulles ;
- Les temps de service des clients, notés $\{s_i\}_{i \in I}$ dans l'article de Schneider et al., sont nuls ; nous les omettons dans notre code.

A noter qu'une hypothèse de l'article est que si un véhicule emprunte une station de rechargement alors il y reste jusqu'à ce que sa batterie soit pleine. Nous conservons cette hypothèse.

Généralités sur le code du projet

Organisation du code

Le code de notre projet est organisé de la façon suivante :

- Les fichiers *milp.jl*, *column_generation.jl* (associé au fichier *dynamic_programming.jl*) et *heuristic.jl* contiennent l'implémentation des trois méthodes de résolution attendues. Le fichier *routes_implementation.jl* permet de convertir la donnée d'une route en différents formats.
- Le fichier *solve.jl* permet de choisir une instance du problème et de la résoudre avec l'une de ces trois méthodes. Les performances de la résolution peuvent être mesurées.
- Un dossier *data* contient les fichiers *.txt* des instances, le fichier *data_extraction.jl* permettant l'extraction des données de ces instances et le fichier *generate_instance.py* permettant de générer de nouvelles instances du problème.

Structures de données
Comme les graphes des instances ne sont pas complets et même souvent éparses, nous implémentons l'incidence des noeuds ainsi que les données relatives aux arcs (distances, temps de parcours etc.) au moyen de dictionnaires et non de matrices.

Instances

Génération de nouvelles instances
Afin de tester nos différents algorithmes, nous avons généré une batterie d'instances. Cette génération n'est pas celle donnée par les examinateurs puisque, ayant testée cette dernière, nous n'avons trouvé qu'à génerer des instances dont la faisabilité n'était pas trouvée par nos algorithmes (certains sommets clients n'étaient pas atteigables). C'est pourquoi les nouvelles instances ont été générées de sorte à admettre une solution faisable. Chaque instance a la structure suivante :

- choix du nombre de clients ;
- choix de la taille de la grille carrée sur laquelle les sommets du graphe vont être placés ;
- choix du nombre de divisions qui divise la grille en zones ;
- les clients sont placés aléatoirement sur la grille ;
- dans chaque zone, on place aléatoirement une station ;
- le graphe orienté est presque complet, à l'exception qu'aucun arc n'a pour origine le dépôt d'arrivée, aucun arc n'a pour destination le dépôt de départ et qu'aucun arc ne boucle sur un même sommet ;
- les poids des arcs sont tous égaux à 1 ;
- les fenêtres de temps sont toutes égales à $[0, tpsmax]$ où $tpsmax$ est la longueur des fenêtres de temps ;
- les distances des arcs sont les distances euclidiennes entre les sommets.

Nettoyage du graphe
Une fonction *clear_graph* a été créée afin de simplifier le graphe initial et d'alléger le travail par la suite. Son rôle est de supprimer les arcs infaisables, i.e les arcs (i, j) qui satisfont l'une des impossibilités suivantes :

- les fenêtres de temps de i et de j sont incompatibles en vue du temps de partir entre ces deux sommets
- prendre l'arc (i, j) ne permet pas d'atteindre le dépot d'arrivée à temps
- la somme des capacités (dans le cas où on charge les voitures) à délivrées aux sommets i et j est supérieure à la capacité maximale de la voiture
- une voiture avec batterie maximale ne peut aller de i à j

Enlever l'arc (i, j) revient à le supprimer du dictionnaire des distances, du dictionnaire des temps, de supprimer j de la liste des voisins sortants de i et de supprimer i de la liste des voisins entrants de j.

Chapitre 1

Programme mixte linéaire

Langage et solveur
Comme le reste du code, le MILP a été écrit en langage Julia et fait appel à la bibliothèque JuMP pour faire l'interface avec le solveur GLPK que nous utilisons pour résoudre les problèmes d'optimisation.

Approche MILP
La formulation MILP est directement tirée de l'article 'The electric vehicule routing problem with time windows and recharging station' de Schneider et al. datant de 2012. Nous ne le réécrivons pas afin de ne pas alourdir le rapport.

Duplication des stations de rechargement
La formulation du problème dans l'article de Schneider et al. suppose qu'une solution ne passe qu'une fois par chaque sommet (chemin hamiltonien). Comme en pratique une station de rechargement peut être visitée par plusieurs véhicules, il peut être nécessaire de dupliquer fictivement les stations pour autoriser indirectement une visite multiple de ces stations. Cette opération de démultiplication est effectuée dans la fonction *extract_data*. Etant données les difficultés rencontrées par les algorithmes de résolution dès que les instances grossissent, nous avons choisi d'implémenter la duplication à l'aide d'un paramètre *nb_duplicates* dans la fonction *extract_data* : pour chaque station s, on crée donc *nb_duplicates* stations fictives avec les mêmes caractéristiques que s, ainsi que les mêmes arcs que ceux adjacent à s. Le fait de passer *nb_duplicates* en argument de la fonction permet d'adapter le nombre de duplicata par station à ajouter au graphe avant de lancer la résolution afin de garder une instance de taille résonnable.
A noter qu'il est possible qu'en ajustant *nb_duplicates* de manière à garder une instance solvable par l'algorithme, nous n'obtenions pas l'optimum du problème qui pourrait nécessiter plus de visites par une même station de rechargement que ce que nous autorisons.
Finalement, nous n'avons pas eu besoin de recourir à la duplication des stations de chargement.

Chapitre 2

Génération de colonnes

2.1 Modélisation

2.1.1 Problème maître

Colonnes

Une colonne correspond à la route d'un véhicule, partant du dépôt départ et revenant au dépôt arrivée, satisfaisant les différentes contraintes (fenêtres de temps, batterie non vide, capacité du véhicule). Formellement, il s'agit d'un vecteur colonne χ tel que

$$\chi = (\chi_{ij} = x_{ij})_{\{(i,j)\in A\}} \in \{0,1\}^m.$$

Remarquons que χ désigne bien un vecteur colonne, non une matrice. Nous notons par abus χ_{ij} le coefficient entier du vecteur colonne χ se trouvant à l'indice associé à l'arc $(i,j) \in A$.

Variables du problème maître

Supposons que l'on dispose de $K \in \mathbb{N}$ routes admissibles $\mathcal{X} = \left\{\chi^{(k)}\right\}_{k\in[\![1,K]\!]}$.
Les variables du problème maître sont, pour $k \in [\![1,K]\!]$,

$$\delta_k \in \{0,1\}, \text{ où } \delta_k = 1 \text{ signifie que la route } k \text{ est sélectionnée.}$$

Contraintes du problème maître

Les contraintes du problème maître sont les **contraintes de livraison des clients**.
Chaque client doit être livré par un unique véhicule,

$$\forall\ i \in I, \quad \sum_{k=1}^{K} \left(\sum_{\substack{j\in V_{n+1} \\ (i,j)\in A}} \chi_{ij}^{(k)} \right) \delta_k = 1 \qquad (\pi_i)$$

où pour $i \in I$ et $k \in [\![1,K]\!]$, la somme en facteur de δ_k est égale à 1 si la route $\chi^{(k)}$ livre le client i, 0 sinon. Nous notons $\pi = (\pi_i)_{i\in I}$ les variables duales associées à ces contraintes. Remarquons que l'on pourrait également considérer que la contrainte de livraison soit une inégalité, où l'on impose que le terme de gauche soit supérieur au terme de droite, plutôt qu'une égalité.

Objectif du problème maître
La fonction objectif est :

$$\min_{\delta} \sum_{k=1}^{K} \left(\sum_{(i,j)\in A} \chi_{ij}^{(k)} d_{ij} \right) \delta_k$$

Initialisation de l'ensemble des routes admissibles
La première étape de la méthode de génération de colonnes est de résoudre le problème maître. Il nous faut donc définir un ensemble de routes initiales tel qu'il existe une solution admissible pour le problème maître, c'est-à-dire relativement aux contraintes de livraison.
Pour $i \in I$, on associe un entier $k_i \in [\![1, |I|]\!]$ (de sorte que $i \mapsto k_i$ soit une bijection de I vers $[\![1, |I|]\!]$). Nous proposons de considérer $|I|$ routes que l'on note $\chi^{(k_i)}$ pour $i \in I$. Chaque route $\chi^{(k_i)}$ permet de livrer uniquement le client i en étant de la forme

$$\chi^{(k_i)} = (0, \dots, 0, \underbrace{1}_{\text{en } (0,i)}, 0, \dots, 0, \underbrace{1}_{\text{en } (i,n+1)}, 0, \dots, 0) \in \{0,1\}^m.$$

Ces $|I|$ routes forment une solution admissible pour le problème maître puisque chaque client $i \in I$ est couvert par une route $\chi^{(k_i)}$.
Néanmoins, le graphe ne contient pas nécessairement d'arcs de la forme $(0, i)$ et $(i, n+1)$ pour $i \in I$. Lorsque ces arcs n'existent pas dans l'instance, il nous faut donc les ajouter au graphe, ce que l'on fait en leur attribuant des distances et des temps infinis (très grands relativement au poids de l'instance).
Certes, ces routes ne sont pas admissibles pour le problème original puisqu'elles empruntent des arcs non-existants initialement. Ce ne sont pas des routes que l'on aurait pu obtenir en résolvant le sous-problème. Néanmoins, comme le problème maître ne vérifie pas la faisabilité de telles routes, rien ne nous empêche de les considérer et de les utiliser pour initialiser le processus.
Bien entendu, il faut être sûr que ces routes non-admissibles soient évacuées au cours de la résolution, au fur et à mesure que de nouvelles routes vont être générées, ce qui va se produire puisque ces routes non-admissibles ont des coûts infinis (significativement plus grands que toute autre route admissible).

2.1.2 Sous-problème comme MILP

Supposons que l'on dispose des valeurs des variables duales $\{\pi_i\}_{i\in I}$ associées aux n contraintes du problème maître résolu.

Variables du sous-problème
Les variables du sous-problème problème sont : pour $(i, j) \in A$, $x_{ij} \in \{0, 1\}$, où $x_{ij} = 1$ signifie que la route du véhicule passe par l'arc (i, j).

Contraintes du sous-problème
Les contraintes du sous-problème correspondent à une partie de celles de la modélisation MILP du problème :

- **Les contraintes de conservation du flux**.
- **Les contraintes de temps**.
- **Les contraintes de stock**.
- **Les contraintes de charge**.

Objectif du sous-problème

La fonction objectif du sous-problème minimise le coût réduit associé à la colonne calculée

$$\min_x \underbrace{\sum_{(i,j)\in A} x_{ij} d_{ij}}_{\text{coût de la route}} - \sum_{i\in I} \pi_i \left(\sum_{\substack{j\in V_{n+1} \\ (i,j)\in A}} x_{ij} \right) = \min_x \sum_{(i,j)\in A} d_{ij} x_{ij} - \sum_{i\in I} \sum_{\substack{j\in V_{n+1} \\ (i,j)\in A}} \pi_i x_{ij}.$$

2.1.3 Sous-problème comme plus court chemin sous contraintes

Reformulation du sous-problème comme problème de plus court chemin sous contraintes

La fonction objectif du sous-problème peut être ré-écrite de la façon suivante

$$\min_x \sum_{i\in I} \sum_{\substack{j\in V_{n+1} \\ (i,j)\in A}} (d_{ij} - \pi_i) x_{ij} + \sum_{i\in F\cup\{0\}} \sum_{(i,j)\in A} d_{ij} x_{ij}.$$

Ainsi reformulé, le sous-problème correspond à un problème de plus court chemin sous contraintes, de temps, de stock et de charge, où le poids des arêtes du graphe sont :

- $d_{ij} - \pi_i$ pour les arcs de la forme (i, j) avec $i \in I$ et $j \in V_{n+1}$;
- d_{ij} pour les autres arcs.

Afin de résoudre ce problème de plus court chemin sous contraintes, nous avons développé (ou plutôt tenté de développer) un algorithme inspiré de l'article de Desrosiers et al., lui même inspiré de l'algorithme de Ford-Bellman.

Algorithme de Ford-Bellman

Dans ce paragraphe, nous rappelons le fonctionnement de l'algorithme de Ford-Bellman.

Dans un graphe orienté, le plus court chemin n'a de sens que si le graphe ne présente pas de circuit de longueur négative. Autrement, si un tel circuit négatif existe alors on peut obtenir un chemin aussi court (négatif) que l'on souhaite en empruntant le circuit négatif un nombre de fois arbitrairement grand. L'algorithme de Ford-Bellman, calculant les plus courts chemins d'un sommet source s à tous les autres sommets du graphe, fonctionne pour toute instance de graphe orienté qui ne présente pas de circuit de longueur négative.

L'algorithme de Ford-Bellman consiste à associer une étiquette à chaque sommet. Au cours de l'exécution de l'algorithme, à un sommet v donné, l'étiquette de v stocke la plus petite distance, actuellement connue, nécessaire pour arriver à v à partir du sommet source s ainsi que le sommet qui précède v dans le plus court chemin, actuellement connu, de s à v. Initialement, tous les sommets sauf s sont à une distance infinie et n'ont pas de prédécesseur ; le sommet source s a une distance nulle et est ajouté à la file des sommets dont les étiquettes ont été récemment modifiées. Puis itérativement, l'algorithme retire le premier sommet de cette file et actualise les étiquettes de ces voisins. L'algorithme procède ainsi de proche en proche jusqu'à ce que tous les sommets aient été visités. Une fois l'actualisation des étiquettes terminées, il est possible d'obtenir le plus court chemin de s à v : il suffit de partir de v et d'aller de prédécesseurs en prédécesseurs jusqu'à atteindre le sommet source s.

Existence de circuits de longueurs négatives dans le graphe du sous-problème

Revenons à notre problème de tournées de véhicules électriques. Les graphes des instances étudiées ont des distances positives sur les arcs. Néanmoins, dans le cadre du sous-problème, nous altérons les distances de certains arcs en leur soustrayant des quantités (potentiellement positives) comme expliqué au début de cette section. Par conséquent, il est possible que les distances sur les arcs deviennent négatives et que des circuits de longueurs négatives apparaissent, ne permettant plus d'utiliser l'algorithme.

Nous avons observé sur quelques instances que de tels circuits de longueurs négatives peuvent bien apparaître...

Modification de l'algorithme de Ford-Bellman pour gérer la présence de circuits négatifs
Dans le cadre de notre problème de tournées de véhicules électriques, nous souhaitons que chaque client ne soit visité qu'une fois. Autrement dit, dans le sous-problème, nous ne cherchons pas simplement le plus court chemin mais le plus court chemin qui passe au plus une fois par chaque client.
En imposant cette contraite d'unique visite des clients, nous éliminons le problème des circuits de longueur négatives puisqu'un circuit ferait parcourir à plusieurs reprises un même client. En fait, le problème des circuits négatifs n'est pas tout à fait évacué car il pourrait exister un circuit négatif entre des stations de rechargement mais excluons ce cas.
Afin de garantir que les clients ne soit visités qu'au plus une fois dans le plus court chemin depuis le sommet source s, nous pouvons stocker dans les étiquettes des sommets v non pas simplement leur prédécesseur dans le plus court chemin connu de s à v mais le plus court chemin connu de s à v lui-même. Peut-être y a-t-il une meilleure solution que celle-ci, mais nous ne voyons pas comment faire autrement.
Finalement, nous n'avons pas implémenté ces étiquettes stockant les chemins car nous avons rencontré d'autres difficultés plus prioritaires...

Algorithme de Desrosiers et al.
L'algorithme présenté dans l'article de Desrosiers et al. permet de résoudre le problème de plus court chemin sous contraintes de fenêtre de temps uniquement.
Le principe de fonctionnement de l'algorithme proposé s'inspire fortement de celui de Ford-Bellman. La différence repose dans les étiquettes des noeuds. Les noeuds possèdent non pas une mais plusieurs étiquettes. Cela est du au fait que nous avons désormais deux critères pour juger la qualité d'un chemin, sans qu'un critère soit préférable à l'autre : la distance (la plus courte) et le temps (le plus court). L'algorithme va donc mettre en évidence des chemins paréto-optimaux que l'on ne peut pas réduire.
Nous avons implémenté cette algorithme dans un premier temps dans le fichier *dynamic_programming.jl*.

Adaption de l'algorithme de Desrosiers et al.
Il nous faut ensuite ajouter les contraintes de consommation électrique / batterie des véhicules. Désormais, nous aurons non pas deux mais trois critères pour juger de la qualité d'un chemin, sans qu'un critère soit préférable à l'autre à nouveau : la distance (la plus courte), le temps (le plus court) et la batterie (au plus haut niveau). Il y aura donc d'autant plus d'étiquettes à gérer.
Dans le fichier *dynamic_programming.jl*, nous avons commencé à implémenter les structures de données nécessaires pour gérer ce troisième critère. Néanmoins, cela requiert des structures de données assez complexes pour gérer l'ordre des étiquettes à chaque noeud.
A cela s'ajoute, pour chaque étique, le stockage du chemin depuis le sommet source s comme nous l'avons expliqué précédemment. Le code nous semble prendre une très grande dimension...

Chapitre 3

Heuristique

3.1 Motivations pour la définition de l'heuristique

Obtention d'une solution admissible difficile

Après avoir réfléchi à plusieurs reprises sur le problème, nous avons constaté que la construction d'une solution admissible du problème par un procédé algorithmique (glouton par exemple) est relativement compliquée. A fortiori, la construction d'une bonne solution admissible du problème est également compliquée. La difficulté provient du fait que le graphe n'est pas complet et que, pour la construction d'une route entre les dépôts départ et arrivée, il faut anticiper la compatibilité des contraintes de fenêtres de temps des différents sommets qui vont constituer la route ainsi que la consommation de la batterie ; il semble nécessaire d'avoir une approche globale, par exemple une formulation MILP, plutôt que locale, par exemple gloutonne, pour construire une route admissible.

Par ailleurs, supposons que l'on dispose d'une solution admissible du problème, c'est à dire un ensemble de routes sélectionnées couvrant l'ensemble des clients, il n'est pas non plus évident de concevoir un voisinage de solutions pour pouvoir faire de la recherche locale. La difficulté provient du fait qu'une petite perturbation d'une route (ajout d'un sommet intermédiaire, suppression d'un sommet intermédiaire) provoque quasiment systématiquement l'infaisabilté de la route.

Du fait de ces constats, il ne nous semble pas judicieux de développer une métaheuristique pour résoudre le problème, à moins de s'engager dans des méthodes hybrides (recherche locale, taboo et recuit simulé) acceptant l'infaisabilité des solutions au prix d'une pénalisation de l'objectif comme cela est fait dans l'article de Schneider et al. Nous nous contenterons de développer une heuristique de résolution.

Relaxation de l'hypothèse d'unique visite des clients

Nous décidons de relâcher la contrainte d'unique visite des clients. Nous autorisons que plusieurs routes puissent passer par le même client. Cela nous semble raisonnable dans la mesure où, le graphe n'étant pas complet (tous les clients ne sont pas connecté aux dépôts départ et arrivées ainsi qu'à toutes les stations de rechargement), il n'est pas impossible qu'un groupe de clients ne puissent être couvert par une seule route et que ces clients ne soient accessibles qu'en passant nécessairement par un certain client.

3.2 Fonctionnement de l'heuristique

Fonctionnement

L'heuristique que nous proposons s'inpire de la génération de colonnes. Initialement, aucune route n'est définie et aucun client n'est couvert. Puis, nous itérons les étapes suivantes jusqu'à ce que tous les clients soient couverts. Pour chaque client non-couvert, nous générons à l'aide d'un MILP une route admissible passant par ce client et nous lui attribuons un score. Parmi les routes générées, nous choisissons de façon gloutonne celle qui a le meilleure score. Cette nouvelle route sélectionée permet de couvrir un certain nombre de clients qui ne l'étaient pas. A chaque itération, nous réduisons donc strictement le nombre de clients non-couverts. Le programme termine.

Génération d'une route admissible - Contrainte de visite par un client

Pour le fonctionnement de l'heuristique, nous devons être capables de générer une route admissible passant par un client donné. Pour cela, nous utilisons un MILP.

Pour garantir que la route soit admissible, une partie des contraintes correspondent à celles du sous-problème de la génération de colonnes à savoir :

- **Les contraintes de conservation du flux** ;
- **Les contraintes de temps** ;
- **Les contraintes de stock** ;
- **Les contraintes de charge**.

Pour forcer la visite d'un client, nous ajoutons la **contrainte de visite d'un client** $\tilde{i} \in I$ comme suit

$$\sum_{\substack{j \in V_{n+1} \\ (\tilde{i},j) \in E}} x_{\tilde{i}j} = 1.$$

Génération d'une route admissible - Fonction objective et score

Pour que la formulation MILP soit complète, il nous faut définir une fonction objective à optimiser. Nous proposons les critères suivants :

- **Le plus court chemin.** Etant donné que l'objectif du problème est de minimiser la distance totale parcourue (et par conséquent le nombre de véhicules), un choix naturelle de critère pour la génération d'une route est de choisir la route de plus courte distance. Cela dit, cette approche a des limites puisque l'on risque de générer beaucoup de routes courtes qui, ensembles, formeront une distance totale importante.
- **Le plus de nouveaux clients couverts**. Afin de réduire le nombre de véhicules, il semble judicieux de valoriser le fait que des clients jusqu'alors non-couverts le soient désormais par la route générée. Autrement dit, on peut chercher à maximiser les nouveaux clients couverts. Néanmoins, avec cette approche, nous risquons de générer peu de routes, certes, mais longues, de sorte que, ensembles, formeront une distance totale importante.
- **Le meilleur compromis entre les deux.** Naturellement, nous pouvons définir un critère qui soit un compromis entre les deux objectifs précédents, au moyen d'une somme pondérée des deux critères.

Il nous reste à définir un score par route. Nous proposons les scores suivants :

- **La valeur optimale de l'objectif.** Le score d'une route peut être tout simplement la valeur optimale de la fonction objective à la génération de la route.
- **Un compromis.** Dans le cas du critère du plus court chemin, on peut également définir les scores des routes comme les ratios de leur longueur sur le nombre de nouveaux clients découverts. Cette approche permet de considérer le compromis expliqué juste avant, non pas dans la fonction objectif de la génération des routes, mais dans le score de la route.

Finalement, nous utilisons le critère du plus court chemin et le score compromis. Cela permet de n'avoir à

générer que $|I|$ routes et donc de ne résoudre que $|I|$ MILP. Initialement, nous générons les $|I|$ routes associés à chaque client et nous leur attribuons leurs scores respectifs. Puis à chaque fois que l'on sélectionne la route de meilleure score, nous actualisons les scores des routes en recalculant le nombre de nouveaux clients qui seraient couverts par chaque route. Bien entendu, ce gain se fait au prix d'une grande simplicité du code et donc de solutions admissibles de moyennes qualités.

Limites

Nous pouvons observer plusieurs limites à cette heuristique :

- L'heuristique requiert de résoudre des MILP à plusieurs reprises. Or ces MILP peuvent être relativement couteux à résoudre si l'instance commence à être grande.
- Cette heuristique n'est pas très éloignée de la génération de colonnes. Elle peut d'ailleurs constituer une initialisation de la génération de colonnes.

Pistes d'évolution

Voici quelques pistes d'évolution que nous avons envisagées :

- Dans l'heuristique, nous avons décidé de générer une route admissible par client non-visité. Nous pourrions également générer des routes admissibles devant passer par plusieurs clients non-visités à la fois. Nous pourrions également forcer la route à ne pas passer par certains clients déjà visités. Après chaque sélection de route de meilleur score, nous pourrions ainsi générer une plus grande variété de routes pouvant amener à de meilleurs résultats.
- Si l'on suppose que le graphe est complet comme dans l'article de Schneider et al. alors il est plus facile de concevoir un voisinage pour une solution admissible puisque l'ajout ou la suppression d'un sommet à une route admissible sera beaucoup plus souvent une route admissible. Il semble donc plus facile de développer une métaheuristiques de recherche locale pour améliorer la solution obtenue avec l'heuristique présentée.

Chapitre 4

Performances

Instances
Nous présentons dans cette section les performances des différentes approches présentées pour un ensemble de 7 instances : MILP, génération de colonnes (avec MILP/avec DP) et heuristique (avec deux critères différents). Les instances sur lesquelles nous avons testé nos algorithmes étant en nombre réduit, il conviendra de prendre les analyses suivantes avec beaucoup de précautions. La raison pour laquelle nous avons testé nos algorithmes sur un nombre aussi restreint d'instances est que dès que nous voulions faire grossir l'instance, les différentes méthodes explosaient en temps de résolution.

MILP & Génération de colonnes
Comme le montre le tableau ci-après, les approches exactes MILP et génération de colonnes fournissent les mêmes valeurs optimales. Par ailleurs, un avantage de la génération de colonnes par rapport au MILP (tes que nous les avons codés) est qu'elle fournit directement les routes utilisées dans la solution optimale.
Nous constatons que l'approche MILP fournit la solution optimale en un temps plus faible que la génération de colonnes. La différence de temps de résolution entre le MILP et la génération de colonnes avec MILP est étonnante du fait que l'on s'attendait à un temps plus faible pour la génération en augmentant la taille de l'instance.
Néanmoins, le temps de résolution de la génération de colonnes explose rapidement : une instance à 10 sommets suffit à rendre la résolution impossible. Le temps de résolution du MILP explose pour des instances un peu plus grandes certes mais tout de même relativement petites également : une instance à 10 sommets suffit à rendre la résolution impossible.

Heuristiques
L'heuristique effectuant la génération de routes avec un critère de plus court chemin et la définition de scores comme ratios entre distances et nombres de nouveaux clients (notée "Heuristique avec (1)" dans le tableau ci-après) donne de moins bonnes solutions que celles de l'heuristique effectuant la génération de routes avec un critère compromis ente plus court chemin et nombre de nouveaux clients (notée "Heuristique avec (2)") - les solutions de cette seconde heuristique étant souvent optimales. Néanmoins, le temps de résolution de la première heuristique est plus faible que celui de la seconde heuristique, et n'explose pas aussi rapidement. Pour cause, la première heuristique nécessite de résoudre beaucoup moins de MILPs.
Quelle que soit les critères et les définitions de scores, ces heuristiques ne sont très satisfaisantes car elles demandent des temps de résolution importants et saturent tout autant que les méthodes exactes...

Instances	Résultats	MILP	Génération de colonnes		Heuristique	
			avec MILP	avec DP	avec (1)	avec (2)
E_data	Nb véhicules	1	1	-	1	1
	Dist. totale	5	5	-	5	5
	Tps exec (ms)	1.0	4.0	-	1950	1975
	Nb arcs simplifiés	0	0	0	0	0
	Nb MILP	1	2	-	2	2
E_data_1	Nb véhicules	2	2	-	2	2
	Dist. totale	18	18	-	18	18
	Tps exec (ms)	1.5	8.9	-	2250	2450
	Nb arcs simplifiés	0	0	0	0	0
	Nb MILP	1	3	-	3	4
E_data_2	Nb véhicules	2	2	-	2	2
	Dist. totale	14	14	-	15	15
	Tps exec (ms)	2.4	8.0	-	2215	2335
	Nb arcs simplifiés	0	0	0	0	0
	Nb MILP	1	2	-	3	4
E_data_3	Nb véhicules	3	3	-	4	3
	Dist. totale	17	17	-	20	17
	Tps exec (ms)	7.1	50.0	-	2375	2420
	Nb arcs simplifiés	0	0	0	0	0
	Nb MILP	1	3	-	5	9
instance_8	Nb véhicules	1	1	-	3	3
	Dist. totale	4	4	-	6	6
	Tps exec (ms)	5.6	307.2	-	2330	2340
	Nb arcs simplifiés	0	0	0	0	0
	Nb MILP	1	3	-	3	6
instance_10	Nb véhicules	1	-	-	5	1
	Dist. totale	6	-	-	10	6
	Tps exec (ms)	55.0	∞	-	2350	10^5
	Nb arcs simplifiés	0	0	0	0	0
	Nb MILP	1	-	-	5	5
instance_16	Nb véhicules	-	-	-	11	-
	Dist. totale	-	-	-	22	-
	Tps exec (ms)	∞	∞	-	3000	∞
	Nb arcs simplifiés	0	0	0	0	0
	Nb MILP	-	-	-	11	-

TABLE 4.1 – Résultats des trois méthodes pour différentes instances. "Heuristique avec (1)" correspond à la génération de routes avec un critère de plus court chemin et la définition de scores comme ratio entre distance et nombre de nouveaux clients. "Heuristique avec (2)" correspond à la génération de routes avec un critère compromis entre plus court chemin et nombre de nouveaux clients. Ces performances ont été obtenues sur un MacBook Pro muni d'un processeur 2.5 GHz Intel Core i5 double coeur & d'une RAM de 16Go.