UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

TrackFood

Hakim Boulahya
Arabella Brayer
Alexis Defontaine
Léni Poliseno

Superviseurs :
Tom Lenaerts
Martine Labbé

Année académique 2015 - 2016

Résumé

Ce rapport présente une résolution du problème mathématique *Dial-A-Ride Problem* (DARP) à travers l'implémentation d'une application nommée *TrackFood*, dont le problème principal est d'organiser des livraisons de repas. Un état de l'art y est proposé, ainsi que l'exposé des méthodes de résolution employées : insertion exhaustive ou partielle, réparation ; de façon statique ainsi que dynamique. Les choix opérés y sont justifiés. Des tests ont également été effectués et y sont présentés. Les résultats ainsi qu'une discussion à leur sujet permettent de constater que l'implémentation de l'algorithme est efficace et fonctionnelle.

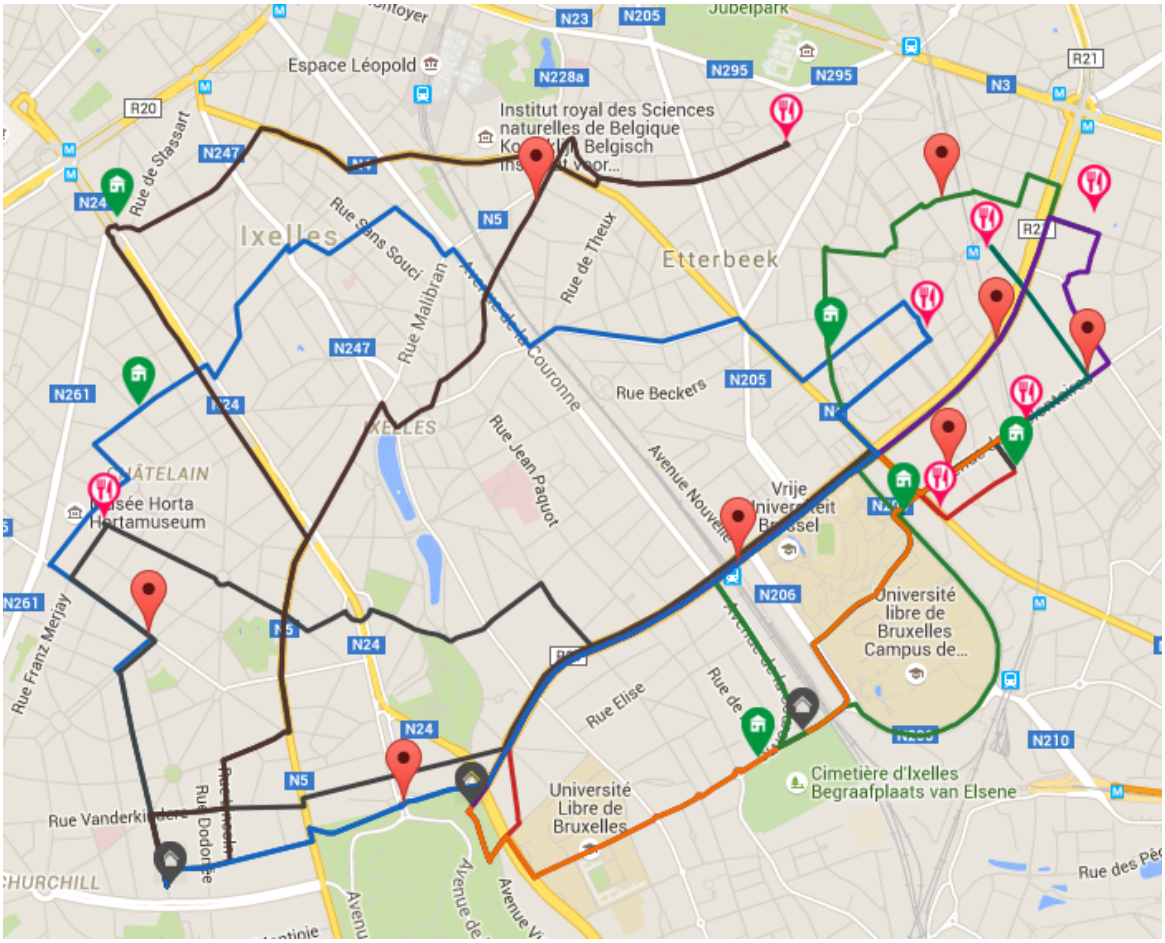


Table des matières

Chapitre 1

Introduction

Le besoin de mobilité est omniprésent dans nos sociétés actuelles. Dans ce contexte, les motivations qui poussent à optimiser les déplacements peuvent être multiples : environnementales, sociales, économiques, etc. Pour les entreprises ayant un but lucratif, il est intéressant de minimiser les coûts liés au transport de leurs produits, pour maximiser leurs profits. Dans le domaine de la santé, il est primordial de pouvoir déterminer un itinéraire rapide pour répondre à une situation d'urgence. Des stratégies d'optimisation peuvent également être mises en place pour offrir un moyen de locomotion partagé entre personnes à mobilité réduite. Il s'agit de pouvoir le proposer au plus grand nombre, en considérant le caractère humain du transport, tout en étant économiquement viable.

Le projet *TrackFood* est l'occasion d'étudier les techniques d'optimisation qui peuvent répondre à de tels enjeux, en particulier au transport de nourriture. Il vise également une vulgarisation du problème au grand public, en étant présenté sous forme d'une application attractive.

Cette application est destinée au client, désirant commander un plat. Il spécifie au passage de la commande le moment auquel il voudrait être livré. Un restaurateur partenaire de l'écosystème réceptionne la demande du client. Il détermine s'il peut la réaliser dans les délais, et estime le moment auquel un livreur pourra réceptionner le plat au restaurant. Les informations de la commande (moment de réception chez le restaurateur et moment de livraison chez le client) sont traitées par la société de livraison. Elle dispose de plusieurs livreurs en déplacement, dont elle connait la position. Elle devra déterminer parmi eux celui qui se chargera de la commande, et à quel moment il devra s'en occuper, sachant que d'autres commandes lui ont potentiellement déjà été attribuées. L'objectif est d'organiser des tournées de livraisons efficaces en temps et en coût, tout en répondant aux attentes du client. Une fois la commande intégrée au système, le client peut suivre la position de son livreur à tout instant, et savoir exactement quand il sera livré.

L'intérêt de l'application *TrackFood* est de montrer l'usage d'un problème mathématique pour répondre à une situation réelle. Le problème d'organisation de livraisons de plats est en effet similaire au problème de transport à la demande, ou *Dial-A-Ride Problem* (DARP). Ce problème étudie plutôt la manière d'organiser le transport de personnes mais peut être facilement transposé au transport de plats, puisque tous deux ont les mêmes contraintes.

Chapitre 2

DARP : état de l'art

2.1 Du *Traveling Salesman Problem* au DARP

L'optimisation des déplacements débute avec l'étude d'un problème aujourd'hui bien connu : le problème du voyageur de commerce, ou *Traveling Salesman Problem* (TSP). Une littérature très abondante à son sujet a vu le jour dans les années 1980-1990, vraisemblablement motivée par les besoins de l'époque.
Il peut s'énoncer comme suit. Un voyageur de commerce désire visiter un ensemble de villes tout en minimisant sa distance parcourue. Il faut alors déterminer l'itinéraire le plus court, visitant chaque ville une et une seule fois, et retournant à la ville de départ [[Haj Rachid et al., 2008]].
Ce problème est énoncé très simplement, mais est parfois difficile à résoudre exactement. Pour pouvoir sélectionner la meilleure solution, il faut générer toutes les solutions possible. Ce nombre explose pour de grandes instances (lorsque le nombre de villes devient grand). Une solution exacte est alors difficilement trouvable en un temps acceptable. Le TSP se ramène donc à un problème d'optimisation combinatoire [[Laporte and Osman, 1995]]. Il est NP-Complet [[Korte et al., 2010]], comme tous les problèmes qui en découlent.

Le problème de tournées de véhicules, ou *Vehicle Routing Problem* (VRP), est très proche du TSP, mais s'en différencie quelque peu : les lieux à visiter sont cette fois répartis entre plusieurs véhicules. Le but n'est donc plus de déterminer un itinéraire unique, mais un ensemble de tournées, chacune associée à un véhicule. Le VRP est parfois appelé dans la littérature *Delivery Problem*, *Vehicule Scheduling*, ou encore *Vehicule Dispatching* [[Rego and Roucairol, 1994]], [[Pillac et al., 2012]], [[Haj Rachid et al., 2008]]. Il a également été appliqué dans d'autres domaines, comme celui de la robotique, ou encore pour élaborer des horaires de travail [Rego and Roucairol, 1994].

Il existe de nombreuses variantes du VRP [[Haj Rachid et al., 2008]], incluant l'une ou l'autre contrainte supplémentaire. C'est le cas du problème de ramassage et de livraison, ou *Pickup and Delivery Problem* (PDP). Dans celui-ci, les lieux à visiter sont couplés et liés à une même demande : l'un est une origine, l'autre une destination. Le véhicule devra d'abord ramasser un objet à l'origine, pour ensuite le livrer à la destination. Cette contrainte est appelée contrainte de précédence. Une contrainte de capacité est aussi à considérer, étant donné que le chargement des véhicules varie tout au long de leur tournée (il augmente au ramassage et diminue à la livraison).
Notons qu'en général un couple de lieux devra être visité par un même véhicule. Il existe cependant des PDP, dits avec transfert, où les objets transportés peuvent être échangés entre véhicules.

Enfin, le DARP est un PDP, où des personnes sont transportées plutôt que des objets. Le caractère humain du transport est donc à prendre en considération (temps d'attente, temps passé dans les véhicules, etc). La section suivante donne une modélisation du DARP et détaille ses contraintes.

2.2 Modélisation

2.2.1 Terminologie

Le DARP est modélisé par plusieurs éléments, qu'il convient de définir [[Deleplanque, 2014]].

Soit une ville représentée par un **graphe**, constitué de nœuds et d'arcs. Un **nœud** est une **origine** (lieu de chargement), une **destination** (lieu de déchargement), ou un **dépôt** (lieu de départ ou d'arrivée des véhicules). Un **arc** représente le trajet le plus court d'un nœud à un autre. Il possède un certain poids, exprimant la distance séparant les nœuds. Cette distance peut être géographique ou temporelle, et dépend de l'objectif à minimiser. Deux nœuds sont reliés par deux arcs : un arc allant du premier nœud vers le second et un arc dans le sens inverse, puisque les deux trajets correspondants ne sont pas forcément identiques. Le graphe est complet. Deux nœuds quelconques sont ainsi toujours joignables.
Notons que les trajets des destinations à leur origine ne seront jamais parcourus. Les arcs correspondants peuvent donc être enlevés du graphe.

Les clients du système effectuent des demandes de déplacement. Une **demande** spécifie le nombre de personnes à transporter (chargement) ainsi que l'origine et la destination, auxquelles sont associées des fenêtres de temps. Le client choisi généralement une des deux fenêtres, le système se charge de déterminer la seconde.

Une flotte de véhicules prend en charge ces demandes. Chaque **véhicule** possède une capacité de chargement. Il effectue une **tournée**, définie par le trajet partant d'un dépôt initial, passant par un ensemble de nœuds (origines et destinations), et arrivant à un dépôt final. Les deux dépôts peuvent être identiques ou non. Un véhicule traite ainsi un ensemble de demandes lui étant assignées.

2.2.2 Contraintes

Les tournées doivent respecter plusieurs contraintes :

— Contrainte de précédence
  Un véhicule doit passer par l'origine avant la destination d'une demande.

— Contraintes de temps
  Un véhicule doit passer aux origines et destinations dans leur fenêtre de temps. Il peut éventuellement arriver plus tôt, mais cela induit des temps d'attente. Ceux-ci ne sont pas préjudiciables pour les clients, mais rendent les véhicule indisponible pour d'autres demande pendant ce laps de temps. Les retards ne sont pas tolérés.
  Aussi, le temps de trajet entre deux nœuds d'une même demande est limité. Il ne doit pas dépasser une valeur fixée par le système.
  Ces contraintes assurent la qualité du service, en remplissant les conditions du transport définies par les clients.

— Contrainte de capacité
  Un véhicule ne peut avoir un chargement supérieur à sa capacité, et ce à tout moment de sa tournée.

Les tournées respectant ces contraintes sont dites admissibles.

2.2.3 Versions statique et dynamique

Il existe deux types de DARP. Une version statique, où toutes les demandes sont connues avant résolution, et une version dynamique, qui intègre ponctuellement de nouvelles demandes à un système en fonctionnement.

2.3 Résolution

Résoudre une instance de DARP revient à construire un ensemble de tournées, chacune associée à un véhicule. Un tel ensemble constitue une solution du problème. L'objectif est évidemment de trouver une *bonne* solution, notion qui varie selon les critères d'optimisation choisis.
D'une part, il faut veiller à la qualité du service pour satisfaire le client. Elle est par exemple liée au temps passé par chaque client dans un véhicule. On cherchera alors à minimiser la somme de ces temps, pour chaque client.
D'autre part, il faut limiter autant que possible les coûts liées au transport. Ceci peut être fait en minimisant la somme des distances parcourues par l'ensemble de véhicules.
Il est aussi possible de considérer plusieurs critères simultanément en minimisant leur somme, chacun étant pondéré d'un coefficient. La valeur du coefficient d'un critère définit l'importance qui lui est accordée. Celle-ci dépend évidemment du contexte dans lequel le DARP sera utilisé.

Il est possible de résoudre une instance de DARP de façon exacte. Dans ce cas, la solution trouvée est la meilleure, parmi toutes les solutions admissibles.
Parfois, il est préférable d'utiliser des heuristiques, moins coûteuses en calculs et donc en temps. La solution est obtenue plus rapidement mais n'est souvent pas optimale. Elle peut néanmoins être suffisante.

Les sous-sections suivantes donnes des exemples de méthodes de résolution exacte et d'heuristiques.

2.3.1 Résolution exacte

Résolution par séparation et évaluation

Dans le domaine de l'optimisation combinatoire, une méthode de résolution exacte souvent employée est celle de la séparation et évaluation, ou *Branch and Bound*. Elle commence par énumérer toutes les solutions en les séparant en sous-ensembles. Ceux-ci sont ensuite évalués, et dès que l'un d'eux est repéré comme non admissible, il est rejeté. L'évaluation des solution qu'il contient sont ainsi évitées.

Cette méthode peut être utilisée pour résoudre DARP, en suivant les même étapes et en considérant ses contraintes :

- Énumérer toutes les solutions. Ce nombre peut être très élevé si les demandes sont nombreuses. Certaines solutions peuvent directement être rejetées en utilisant les contraintes les plus simples, comme celle de précédence.
- Séparer les solutions en branches, par exemple à l'ajout d'une demande dans une tournée, puis réduire jusqu'à ajouter une seule opération (origine ou destination).
- Évaluer l'arbre - selon une stratégie de parcours choisie (largeur [1] /profondeur/meilleure évaluation...) - et couper dès que possible, par exemple lorsque que la fenêtre de temps est dépassée.

Il est possible (et souhaitable) de construire les solutions les unes à la suite des autres. Cela permet de ne générer que les branches à tester.
La méthode reste applicable pour résoudre de grandes instances, en s'arrêtant prématurément si nécessaire. Elle a été éprouvée pratiquement, comme l'expliquent Cordeau et Laporte : *On real-life instances, the algorithm cannot always be run to completion so that it must stop prematurely with the best known solution. It was applied to instances including between 859 and 1771 transportation requests per day in Berlin* [[Cordeau and Laporte, 2007]]. Dans ce cas, l'algorithme ne garantit pas d'avoir trouvé une solution exacte, et s'apparente donc à une méthode heuristique.

1. La stratégie de recherche en largeur ne semble plus usitée, car elle est trop gourmande

2.3.2 Heuristiques

Résolution par recherche taboue

Les algorithmes de recherche taboue ont été introduits initialement par Glover en 1986. Il s'agit d'une méthode de résolution heuristique, dès lors très utilisée pour résoudre des problèmes d'optimisation combinatoire [[Cordeau and Laporte, 2002]].

Cordeau et Laporte ont adapté cette méthode au DARP statique. Ils partent d'une solution initiale (simple à produire mais violant certaines contraintes), qu'ils raffinent en itérant un certain nombre de fois. Dans ce processus d'itération, certaines solutions sont écartée : elles deviennent *taboues*. Ainsi, les mauvaises solutions ne sont pas recalculées.
Afin de limiter la convergence vers un optimum local, il faut diversifier les méthodes de raffinement de la solution en construction : déplacer une demande (origine et destination) d'une tournée à une autre, échanger deux nœuds de tournées différentes ou dans une même tournée, etc.
Une fois une nouvelle solution construite, elle est comparée à la précédente selon son coût. Celui-ci est calculé en tenant compte des temps de trajet de chaque tournée, additionnés à des pénalités attribuées à chaque fois qu'une contrainte n'est pas respectée. Ces pénalités peuvent être pondérées.

Cette méthode de résolution présente un certain degré d'incertitude quant à la validité de la solution trouvée après un nombre d'itérations arbitrairement choisi. Ce nombre est difficile à dimensionner et donc le temps d'obtention d'une solution admissible n'est pas garanti.

Résolution par insertions

Une autre heuristique, exposée par Zhao [[Zhao, 2011]], utilise des techniques d'insertions.
Soit un ensemble de demandes à répartir parmi k véhicules. La solution de départ est alors initialisée à k tournées vides ne contenant que le dépôt de départ, et celui d'arrivée (qui peuvent éventuellement être identiques). Le principe est de sélectionner une demande d et de l'insérer dans une des tournées, sans modifier l'ordre des demandes précédentes. De cette façon, la solution, qui se construit insertion après insertion, respecte toutes les contraintes à chaque itération.

Il existe de nombreuses possibilités de placement pour l'origine et la destination de d au sein d'une tournée. Toutes ces combinaisons sont évaluées, et celle rendant la tournée la plus courte est sélectionnée. Il est possible de limiter le nombre de ces combinaisons en éliminant directement les tournées violant les contraintes de fenêtre de temps. Les performances de l'algorithme sont ainsi rehaussées.
Le choix de la tournée intégrant d reste à définir. L'insertion partielle essaie d'insérer d dans la plus courte tournée qui soit compatible. L'insertion exhaustive tente d'insérer d dans chaque tournée. Elle compare alors le temps total des solutions ainsi générées, et sélectionne la plus courte en temps.

Il se peut que d ne soit intégrable dans aucune des tournées déjà établies. Des techniques de réparation sont alors appelées. L'un d'elles consiste à déplacer les demandes d'une tournée $T1$, non compatibles avec d, dans une tournée $T2$. La demande d est ensuite insérée dans $T1$. Si cette opération échoue pour chaque couple $(T1, T2)$, un autre processus de réparation a lieu : deux demandes aléatoires sont échangées entre deux tournées et le processus d'insertion de d est appelé à nouveau.

Zhao compare la recherche taboue ci-dessus à la méthode d'insertion. La première montre de meilleures performances du point de vue de la distance totale parcourue. La seconde minimise plutôt les durées des tournées, les temps d'attente et le temps moyen pour traiter une demande. Elle détermine aussi une solution beaucoup plus rapidement. En ce sens, elle est plus optimisée du point de vue client.
Pour ces raisons, et parce qu'elle a l'avantage d'être facilement adaptable au cas dynamique, elle a été choisie comme modèle d'implémentation du projet TrackFood.

Chapitre 3

Méthodes implémentées

3.1 Motivation

Dans le cadre du projet *TrackFood*, une première étape a consisté à trouver le type d'implémentation adapté aux besoins de l'application. Il a été expliqué plus haut que le problème central de cette application est de prévoir des livraisons, afin de répartir le travail entre les livreurs, de façon efficace. Or, s'il est déjà coûteux de résoudre le problème de façon exacte pour un petit nombre de livraisons, il devient parfaitement impossible de le faire dès lors que l'ensemble de livraisons s'étoffe.

Du point de vue de l'utilisateur, cette contrainte est parfaitement secondaire. En effet, il est inconcevable de lui imposer d'attendre un temps long en raison de difficultés à résoudre un problème mathématique dans l'application. Il devient donc tout à fait indispensable de chercher à gagner du temps, quitte à perdre en efficacité. Cela signifie qu'il faut se tourner vers des heuristiques rapides.

Le domaine des heuristiques - dans les problèmes de ce type - est vaste. Par exemple, la recherche taboue du point [2.3.2] consiste à construire une première solution acceptable pour ensuite l'améliorer et la rendre admissible. La contrainte de devoir fournir une première solution acceptable est un inconvénient notable. C'est principalement celle-ci qui a motivé le choix de l'implémentation d'une méthode d'insertion au détriment de la recherche taboue dans le projet *TrackFood*. En outre, la méthode d'insertion statique peut aisément être modifiée afin de devenir dynamique.

Il est ici utile de rappeler les conditions de développement de l'application *TrackFood*. Un des objectifs principaux consistait en une présentation publique à vocation pédagogique au Printemps des Sciences. Ceci a fortement guidé les choix d'implémentation. En effet, afin de rendre la démonstration attractive, celle-ci démarre en chargeant un ensemble de demandes et propose une première solution. Le problème est ici considéré comme statique, et la solution proposée est calculée en insérant successivement chaque demande comme décrit au point [2.2.3]. Cela comporte l'avantage de pouvoir montrer très rapidement comment se déroule le calcul d'un premier ensemble de tournées.

3.1.1 Fonctionnement général

Globalement, le programme fonctionne de cette façon :

- chargement des données
- première résolution statique
- ajout aléatoire de nouvelles commandes
- résolution dynamique

Un premier scénario pré-encodé est donc avant tout chargé et résolu. Le programme se déroule dans le temps grâce à un `timer`, afin de simuler une période de travail.

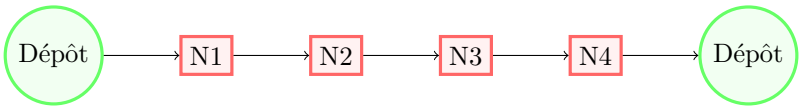
Dans tous les cas, une étape importante consiste à construire de nouveaux chemins [1], puis à les comparer afin de conserver uniquement le meilleur.

La méthode d'insertion choisie ne garantit pas que la solution retournée en cas de réussite soit la meilleure. En revanche, comparée à une méthode "exacte", elle renvoie bien plus souvent une solution existante. Elle souffre cependant de limites, entre autres, d'être fortement liée à l'ordre d'arrivée des commandes. Ce point est soulevé ultérieurement au point [6.2].

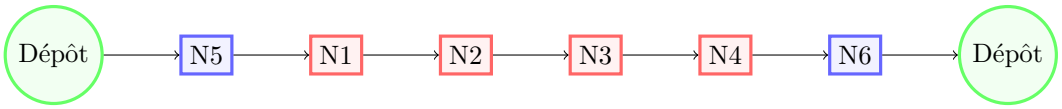
3.2 Recherche d'insertion

Dans l'algorithme apparaît un *itérateur* [2] de chemins. Cet objet permet de générer un ensemble de nouveaux chemins auxquels ont été ajoutés une livraison. Comme son nom l'indique, ces chemins sont générés au fur et à mesure de la demande, et itérés afin de ne pas surcharger inutilement la mémoire du programme. Afin de procéder à la création de tels chemins, une étape est nécessaire : la "recherche d'insertion".

La recherche d'insertion est une étape qui permet de connaître les possibilités d'insertion d'une nouvelle livraison dans un chemin déjà établi. Intuitivement, le nombre de combinaisons possibles apparaît très élevé. En effet, selon la combinatoire, le nombre de possibilités montre une croissance factorielle :



Dans cet exemple, essayons d'insérer une nouvelle livraison composée des nœuds N5 et N6 :
N5 peut se placer aussi bien derrière le premier dépôt que devant le dernier. N6 doit donc arriver après N5. Il peut y avoir d'autres nœuds entre eux.



De nombreux essais inutiles peuvent être évités grâce à l'intervention des contraintes.[2.2.2] L'une des principales et non triviale concerne les fenêtres de temps. En effet, les nœuds ont chacun une fourchette de temps, correspondant à l'heure maximale d'arrivée ainsi que de départ. À chaque nœud est associé un délai de traitement. Une livraison pèse un certain poids et un livreur est limité dans sa capacité de transport. Plusieurs confits peuvent-ils apparaître :

Soit x, un restaurant, et y, $\in N$, un client, issus d'une livraison d_i, et soit P un chemin composé de $n \in N$, ses nœuds triés par ordre de visite. Soit $n.min$ et $n.max$ les horaires respectivement minimum et maximum d'arrivée au nœud. Un *conflit_temps* a lieu dès que :
$x.min + trajet + traitement > n.max$ ou
$y.min + trajet + traitement > n.max$ et ce indépendamment du chemin dans lequel ce conflit apparaît. Il est possible de retirer l'arc de l'ensemble du graphe afin de ne plus jamais risquer de le proposer, puisqu'il est impossible.
L'apparition du conflit détermine la limite au delà de laquelle il est inutile de tester l'insertion.

1. un **chemin** est une tournée assignée à un livreur, constitué des nœuds par lesquels il doit passer ainsi que les horaires.
2. Un **itérateur** est objet qui parcourt une liste d'éléments. Ceux-ci sont retournés au fur et à mesure de la demande. Il possède une méthode permettant l'accès aux données, et le déplacement vers l'élément suivant.

Essayons s'insérer X et Y dans le même chemin, et un conflit_sommet apparaît entre $N3$ et X. C'est N3 qui servira de borne supérieure d'insertion. Cela signifie qu'aucun essai d'insertion de X ne sera produit au delà de cette limite. En procédant de la même façon avec Y, il faut rajouter la contrainte qu'Y ne peut être inséré qu'après X et trouver sa limite également.

Ce n'est cependant pas la seule condition que le chemin doit remplir afin d'être valide. Une fois le chemin "produit", les heures exactes de départ et d'arrivée à chaque nœud sont calculées. De nouveaux conflits de temps apparaissent à ce moment. Ils ne peuvent pas être repérés plus tôt dans le processus, car ils sont inhérents au chemin produit.
Enfin, la contrainte de capacité peut également intervenir et invalider un chemin.

3.3 Méthodes d'insertion

La méthode d'insertion implémentée repose sur l'ordre d'arrivée des commandes. Voici globalement son fonctionnement :

Data: L : un ensemble de livraisons à insérer
P : un stack de tournées (vides) de livreurs, trié par poids
Result: res : un booléen
P : l'ensemble de tournées mis à jour
$temp \leftarrow NULL$;
while $L \neq NULL$ **do**
  $res \leftarrow false$;
  $l_i \leftarrow L.next$;
  $p \leftarrow P.next$;
  while $p \neq NULL$ **do**
    $iterator \leftarrow pathIterator(l, p)$;
    $newPath \leftarrow interator.next$;
    while $newPath \neq NULL$ **do**
      if $temp = NULLortemp.weight > newPath.weight$ **then**
        $temp \leftarrow newPath$;
        $res = true$;
      else
        continue;
      end
      $newPath \leftarrow iterator.next$;
    end
  end
  if $res == true$ **then**
    $P.swap(p, newPath)$;
    $res \leftarrow false$;
  else
    return $false$;
  end
  $p \leftarrow P.next$;
end
return res;

Algorithm 1: Insertion exhaustive

Le programme peut appeler deux méthodes différentes (*insertion exhaustive* ou *insertion partielle*) en fonction du nombre de livraisons prévues. Ce choix est discutable. Il est abordé plus loin dans ce

document [4.5.1][5.4]. C'est cependant la solution qui a été retenue dans le projet *TrackFood*, car il est imaginable que la complexité augmente en même temps que le nombre de livraisons. Or, comme leur nom l'indique, la *méthode exhaustive* est plus complexe que la *méthode partielle*.

La méthode d'insertion en général créé une première solution pas à pas, puis l'évalue, pour enfin tenter de l'améliorer. La différence entre les deux implémentations *exhaustive* et *partielle* concerne le nombre de tentatives d'améliorations.

La méthode exhaustive permet de générer un nombre relativement important de comparaisons. En effet, pour chaque livraison à ajouter, toutes les possibilités d'ajout chez tous les livreurs disponibles seront comparées afin d'obtenir la meilleure. Un nombre important de possibilités ne sont toutefois pas testées puisque les commandes sont insérées dans un certain ordre, et que celui-ci influence fortement le choix des solutions testées. [3.5]

Dans la méthode partielle, la procédure est la même, mais en moyenne, il y a moins de comparaisons car dès qu'un chemin a pu accueillir la livraison, la nouvelle solution est retournée. Cette méthode est donc beaucoup moins exigeante. Elle présente l'intérêt majeur de prendre en moyenne significativement moins de temps[4.5.1]. Or, comme stipulé, l'application *TrackFood* a des exigences importantes en terme de rapidité, et sans doute moins au niveau de l'efficacité. Il est acceptable de ne pas disposer de la meilleure solution, il faut cependant la trouver vite.

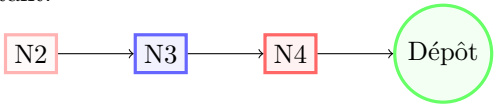
3.4 Passage de dynamique à statique

L'insertion présente plusieurs avantages. Le premier est la bonne adéquation de la méthode statique à une adaptation vers une implémentation dynamique.

L'implémentation **statique** consiste à trouver une solution lorsque toutes les données sont connues d'avance. Concrètement, cela signifie que l'algorithme est lancé alors que toutes les livraisons sont déjà connues. Il est aisé de se convaincre que ce type d'implémentation n'est pas satisfaisant dans le cas d'une application telle que *TrackFood* puisque celle-ci est censée être capable d'accueillir de nouvelles commandes au fur et à mesure de l'utilisation.

Durant la suite de la démonstration/simulation de nouvelles commandes sont ajoutées de façon aléatoire, dynamiquement. Aussi l'algorithme central de l'application peut-il fonctionner aussi bien de façon statique que dynamique, puisque les deux méthodes sont implémentées.

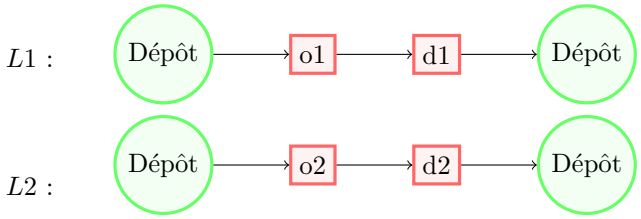
Ce passage est relativement aisé. Lorsqu'une solution est retournée par l'algorithme, tous les livreurs possèdent leur parcours, avec les horaires à respecter pour chaque étape. Dans le cas où une nouvelle livraison arrive, il suffit de l'ajouter, en utilisant soit la méthode exhaustive, soit partielle. Toutefois, il est difficile d'imaginer un livreur changer en cours de route de parcours car une nouvelle livraison aurait été ajoutée. Dans un souci de cohérence avec une utilisation réelle, l'algorithme tente d'ajouter une nouvelle livraison en considérant que le premier nœud est en réalité le prochain nœud dans lequel sera le livreur, s'il est en chemin. La fenêtre de temps de ce nœud est adaptée en fonction des estimations actuelles. Concrètement, si le livreur est actuellement entre le nœud $N2$ et $N3$, c'est $N3$ qui sera utilisé comme nœud de départ pour les calculs. Sa fenêtre de temps sera ajustée en fonction du temps de déplacement restant.



3.5 Méthode de réparation

L'exemple suivant montre que l'ordre d'insertion peut fortement modifier la qualité des résultats retournés.

Soit deux livreurs $L1$ et $L2$, et trois commandes $D1 = \{o1, d1\}$, $D2 = \{o2, d2\}$, $D3 = \{o3, d3\}$. Soit une solution, où $D1$ est associée à $L1$ et $D2$ à $L2$:



En tentant d'insérer D3, l'itérateur ne propose aucun chemin, car les arcs sont tels que la seule solution possible aurait été de regrouper D2 et D3 dans un même chemin, et de confier D1 à un autre livreur. Un tel cas peut se présenter et si aucune méthode autre que l'insertion est appelée, il apparaît qu'aucune solution ne sera retournée alors qu'il en existe.
Pour pallier ce problème, lorsqu'aucune solution n'est trouvée, l'algorithme fait appel à la méthode de réparation. Elle fonctionne de cette façon : Une livraison est retirée d'un chemin. S'il est possible de l'ajouter à un autre, l'échange est fait, et l'itérateur est sollicité afin d'essayer d'ajouter la livraison problématique. Tous ces nouveaux chemins sont sauvegardés si le nouvel ensemble permet d'accueillir la livraison problématique.
Un problème subsiste concernant le choix de la première livraison à retirer du chemin. Afin d'améliorer les chances d'efficacité, un choix cohérent est de retirer une livraison dont au moins un des nœuds est en conflit avec la nouvelle livraison.

Data: rejected : une livraison à insérer
P : un stack de tournées de livreurs
Result: res : un booléen
P : l'ensemble de tournées mis à jour
$res \leftarrow false$;
$p1 \leftarrow P.next$;
while $p1 \neq NULL\ AND\ res = false$ **do**
  $p2 \leftarrow P.next$;
  if $swapIsPossible(p1, p2, rejected)$ **then**
    $P.update$;
    $res \leftarrow true$;
  end
  return res;
end

Algorithm 2: reparation

Cependant, le choix initial de retirer une livraison conflictuelle ne garantit pas qu'une solution sera forcément trouvée. Une autre méthode pourrait alors être appelée. Celle-ci retirerait une livraison de façon aléatoire et tenterait la même opération de *switch*, mais sur un choix aléatoire. Certaines implémentations de **DARP** fonctionnent de cette manière. Toutefois, dans *TrackFood*, cette seconde méthode de réparation n'a pas été implémentée.

Chapitre 4

Résultats expérimentaux

4.1 Résumé

Pour tester notre algorithme, nous avons mis en place une batterie de tests qui nous permettent de visualiser les performances de notre implémentation. Pour ce faire nous devons tester les différents modes de notre algorithme *i.e.* celui implémentant la *recherche partielle* ainsi que le mode implémentant la *recherche exhaustive* d'une solution admissible.

Dans notre algorithme final, nous proposons en réalité une combinaison des deux modes, qui nous permettent ainsi d'avoir un meilleur équilibre entre la rapidité du logiciel et la qualité de la solution. Le passage entre chaque mode dépend d'une variable $BOUND$ qui indique le nombre de livraisons avec lesquels le mode exhaustif peut proposer une solution en un temps acceptable. Le choix s'effectue via cette variable, car la génération d'une solution dépend de *l'itérateur de chemins* [3.2], et plus le nombre de livraisons est élevé plus il y aura de chemins générés [5.4].

La comparaison entre DARP statique et dynamique n'a pas lieu d'être dans les tests de performance. En effet une insertion d'une livraison dans un problème statique représente une insertion d'une livraison dans un problème dynamique à un instant t.

4.2 Objectifs

Les objectifs de ces tests sont de montrer la différence qu'il peut y avoir entre les modes implémentés de l'algorithme *i.e.* partiel et exhaustif. Cela permet d'effectuer des choix appropriés quant aux variables de changement de mode. En effet, ici, il existe uniquement la variable $BOUND$, or, la pertinence d'un tel choix peut-être discutée en appuyant cette discussion par des résultats expérimentaux. Cela permet également de montrer que cette implémentation d'une heuristique du problème de DARP peut être utilisée dans une situation réelle, et fournir un service correct à une entreprise de livraison.

4.3 Environnement de test

Comme pour notre algorithme, le programme de test a été développé en `Java` dans sa version 1.8. La distribution Linux exécutant nos instances de test est le système d'exploitation `Ubuntu Desktop` dans la version `14.04 LTS`.

L'ordinateur utilisé est un ordinateur muni d'un processeur Intel Core i5 2450M cadencé à 2.50GHz muni de 2 coeurs. Le processeur est un modèle 64 bits. Cette ordinateur a une mémoire vive de 4Go de type DDR3.

4.4 Données de test

Les instances de test générées contiennent entre 20 et 150 demandes de livraisons. Ceci correspond à une approximation du nombre de livraison que peuvent effectuer 10 livreurs en une journée.

Les fenêtres des livraisons sont générées à partir de l'intervalle [0, 700] qui indique la limite inférieur de la fenêtre. Les unités des fenêtres de temps sont des minutes. Cet intervalle a été choisi car il correspond à une journée de travail d'environ 12 heures. Cela permet donc de générer des livraisons durant tout une journée de travail.
La limite supérieure de la fenêtre est de $+20 minutes$ par rapport au temps initial, ce qui correspond au standard du temps maximal de préparation d'un plat. Seule la fenêtre du nœud d'origine *i.e.* un restaurant est générée. La fenêtre du nœud de destination *i.e.* un client est construite via la fenêtre de l'origine associée. La limite inférieure de la fenêtre d'un nœud client correspond à la limite inférieure du nœud d'origine associé. La limite supérieure de la fenêtre d'un nœud client correspond à la limite supérieur du nœud d'origine +30.

Il existe 22 lieux de commandes potentiels ainsi que 22 restaurants disponibles. Tous sont centrés autour d'un même point. Ces lieux correspondent à de réelles adresses et les temps de parcours d'un nœud à un autre correspondent au chemin le plus court entre les deux adresses avec le vélo comme moyen de transport. Le calcul des temps de parcours est effectué via l'**API Google Directions**. C'est depuis ces lieux que les nœuds des livraisons sont générés.

Il existe 8 livreurs potentiels. Ils sont tous disponibles dans l'intervalle [0, 720], qui correspond à 12 heures de travail. Les livreurs ont comme nœud d'origine un dépôt. Il existe trois dépôts différents situés dans des lieux différents. Chaque livreur a une capacité de stockage de poids de 10. Le poids d'une livraison varie entre 1 et 6.

4.5 Tests de performance

Nous avons effectué 10 instances tests pour chaque mode. Les tests nommés Pi, où i représente le numéro du test, indiquent une instance de test de performance du mode partiel. Les tests nommés Ei indique une instance d'un test de performance du mode exhaustif.

Dans les tableaux 4.1 et 4.2 les colonnes *Nb. livraisons* indiquent le nombre de livraisons *i.e.* le nombre de nœuds ajoutés au graphe représentant le problème. Toutes les livraisons sont insérées avant l'exécution des tests. L'algorithme utilisé est uniquement statique. Étant donné que l'algorithme dynamique n'est qu'une variante de l'algorithme statique, il n'est donc pas nécessaire de tester les performances de ces dernières. Les colonnes *Nb. chemins gen.* indiquent le nombre de chemins générés *i.e.* le nombre d'insertions effectuées lors de l'exécution de l'algorithme. Les colonnes *Temps total* indiquent le temps d'exécution total de l'algorithme pour effectuer l'insertion de toutes les livraisons et proposer une solution.

4.5.1 Comparaisons des performances

Les tableaux 4.1 et 4.2 contiennent les valeurs des tests effectués. Il y apparaît que le mode exhaustif génère beaucoup plus de chemins que le mode partiel. Le mode exhaustif effectue donc plus d'insertions que le mode partiel, ce qui lui permet de proposer une solution plus optimale.

Les résultats montrent que lorsque le nombre de livraisons à insérer est inférieur à 50, les temps d'exécution sont plus ou moins équivalents. C'est à partir de plus de 50 livraisons à insérer que les différences sont notables. Il est imaginable de choisir une limite de livraison maximale pour la variable $BOUND$ située entre 50 et 100. Au delà de cette limite, le changement de mode devra être effectué afin d'assurer un temps d'exécution correct. Le choix de la valeur fixée à cette variable dépend de la qualité

Instance	Nb. livraisons	Nb. chemins gen.	Temps total
P1	20	342	2.28s
P2	30	590	4.7s
P3	40	1094	8.64s
P4	50	3445	13.43s
P5	60	4207	19.15s
P6	75	11678	30.04s
P7	90	10922	42.92s
P8	110	20215	1m2s
P9	130	34085	1m32s
P10	150	38237	2m4s

TABLE 4.1 – Performance du mode partiel

Instance	Nb. livraisons	Nb. chemins gen.	Temps total
E1	20	544	3.72s
E2	30	576	5.61s
E3	40	1202	8.90s
E4	50	2398	15.61s
E5	60	6406	26.08s
E6	75	9096	37.13s
E7	90	18934	54.01s
E8	110	26273	1m23s
E9	130	24388	1m59s
E10	150	54460	2m31s

TABLE 4.2 – Performance du mode exhaustif

souhaitée de la solution. Plus une solution optimale est nécessaire, plus la valeur de $BOUND$ sera élevée. Inversement, dans le cas où il est préférable de proposer une solution rapidement.

Chapitre 5

Discussion

5.1 Résultats obtenus

Les tests effectués montrent que des solutions correctes peuvent être trouvées dans des temps acceptables pour des instances avec un nombre de livraisons relativement réaliste.
Cependant, les temps retournés restent relativement élevés et la pertinence de devoir laisser l'utilisateur attendre plus d'une minute afin de savoir si sa livraison sera possible pose question. Il pourrait être judicieux de changer la procédure dans le cas d'une attente moyenne d'un délai de cet ordre de grandeur : envoyer une notification pour prévenir que la demande a été traitée.

D'autres implémentations de DARP dans la littérature montrent que des solutions ont été mises en place pour le transport de véhicules d'urgence. Aussi, il doit être tout à faisable d'améliorer encore ces résultats en peaufinant l'implémentation.

Il est difficile de comparer l'implémentation des méthodes d'insertion dans *TrackFood* avec d'autres car pour ce faire, il eut fallu comparer des exécutions d'algorithmes déjà implémentés. Toutefois, il est possible de vérifier que la méthode de *Recherche Taboue*, présentée plus haut [2.3.2] est plus gourmande en temps d'exécution, et semble présenter des résultats plus optimaux.

5.2 Adéquation de l'implémentation avec une utilisation en conditions réelles

TrackFood a été pensé comme une application pouvant servir à des sociétés de livraisons. Des services similaires sont à l'heure actuelle proposés sur le marché, toutefois ils ne proposent pas de "*traquer*" le livreur afin d'avoir une estimation aussi rigoureuse que *TrackFood*. Les tests montrent en tout cas que pour la partie "livraison" de l'application, une telle implémentation est tout à fait justifiée, et fonctionnelle.

Toutefois, les objectifs du projet étant tournés vers une présentation grand-public d'un problème mathématique et de sa résolution par le biais d'application a orienté le développement vers une simulation/démonstration plus que vers une réelle application fonctionnelle. Le développement a montré également les limites d'un tel projet envisagé sous cette forme, et ce sont des limites que les tests ne font pas forcément ressortir.

5.3 Le poids des arcs

Une première critique concerne l'acquisition du poids des arcs : les trajets. Dans *TrackFood*, l'algorithme doit commencer par évaluer le poids de tous les arcs qui seront potentiellement utilisés dans la

création de chemins. Ce processus est réalisé à l'aide de l'**API Google Directions**. Il est facile de se rendre compte du nombre très important de données qui doivent être récoltées pour le simple calcul d'un trajet, puisque tous les poids des arcs possibles entre chaque nœud doivent être connus.

Par exemple, pour un ensemble de 5 livraisons (10 nœuds) effectuées depuis un seul et même dépôt de départ et d'arrivée compte :

— tous les arcs sortants du dépôt vers toutes les origines (5)
— tous les arcs sortants de toutes les destinations vers le dépôt (5)
— tous les arcs de toutes les origines vers leur toutes les destinations (25)
— tous les arcs de toutes les destinations vers les origines sauf la leur (5*4=20)
— Toutes les origines vers toutes les origines (5*4)
— Toutes les destinations vers toutes les destinations (5*4)

Ici sont dénombrés 95 arcs.

Tous ces poids sont récoltés à l'aide d'une requête effectuée auprès de Google. L'API retourne un document JSON [1], qui est parsé afin d'en récolter deux informations primordiales : d'abord le poids, mais aussi le trajet en détail (afin de l'afficher sur la carte).

Ceci pose plusieurs problèmes. Le premier étant de dépendre d'un organisme comme Google, qui n'est pas du tout "libre". L'autre étant la lourdeur du processus, car dans des conditions d'utilisation normales, il est quasi certain qu'il y aura bien plus de données à récolter.

Il est utile de préciser que l'**API Google Directions** limite le nombre de requêtes dans une journée par une même application.

En conclusion, par rapport au système mis en place pour la récupération des poids, le développement montre des limites. Il aurait été possible d'envisager de faire appel à un autre système "libre" dont les données auraient été stockées sur le serveur lui-même. Toutefois, un tel travail est un projet à lui seul qui demande un temps et un investissement importants.

Pour *TrackFood*, la solution a consisté à stocker les fichiers JSON sur le serveur, et à ne les télécharger qu'au cas où ces fichiers auraient été inconnus. Cela signifie que l'application aurait eu des difficultés à fonctionner avec des clients ou restaurants non connus à l'avance, dont les données aurait provoqué de nombreuses requêtes. Cela ne pose pas de problème dans le cadre d'une démonstration où les données sont connues à l'avance. Cependant, il demeure tout de même un temps important de chargement des poids, lié au parcours des fichiers.

5.4 Choix du délimiteur

Le nombre problématique de calculs potentiels et leurs durées ont déjà été évoqués plus haut. La solution proposée par *TrackFood* afin d'éviter une surcharge computationnelle a été de fixer une limite arbitraire au delà de laquelle la méthode devenait moins exigeante et plus rapide [3.3].

Cependant, il est notable que le nombre de calculs à effectuer soit moins sensible au nombre de livraisons en tant que telles qu'au rapport entre le nombre de livraisons et celui des livreurs. Toutefois, les tests effectués montrent qu'un gain de temps commence à se faire substantiel au delà de 50 livraisons. Il aurait été intéressant de tester les temps d'exécution avec un ratio différent de livraison/livreurs également, ce qui aurait renseigné sur le bien fondé de cette limite en nombre plutôt qu'en proportion.

1. JSON est l'acronyme de JavaScript Object Notation. C'est un format générique adapté particulièrement au langage JavaScript.

Chapitre 6

Conclusions et perspectives

6.1 Algorithmique

L'adaptation du *Dial a Ride Problem* pour le transport de plats chauds a pu être réalisé conformément au modèle choisi. L'implémentation a été conçue afin de favoriser les performances en termes de temps d'exécution plutôt que d'optimalité de la solution. Les techniques partielles et exhaustives sont tout de même présentes dans l'application, permettant ainsi de réaliser des tests comparatifs entre ces deux algorithmes.

Le passage en dynamique, rendant l'application plus proche d'une utilisation réelle, a permis de mettre en évidence la souplesse de l'algorithme. Cette adaptation est donc réalisable à partir d'un modèle statique enclin au changement. Les modifications portées sur celui-ci se traduisent par la possibilité de mettre à jour les positions des livreurs. Une partie importante en terme de connectivité a aussi du être imaginée afin de ne pas faire tourner l'algorithme sur un simple smartphone. Les données relatives au trafic étant récupérées via l' *API Google Directions*, des parseurs ont dû être intégrés au programme afin de pouvoir gérer de nouvelles commandes.

Un autre aspect du coté dynamique, hormis l'ajout aléatoire d'une commande dans le système, est l'affinage des données déterminant les contraintes de celle-ci. L'utilisation d'une implémentation de type *Machine Learning* par le biais des *Modèles de Markov Cachés* pourrait alors être une optimisation envisageable. Cette dernière impliquerait une évolution quant à la précision des solutions obtenues puisque son utilisation régulière alimenterait les données du modèle. À partir de ceux-ci, l'algorithme pourrait déterminer - sous formes de probabilités - si la durée des trajets actuellement trouvés pourraient subir ou non un éventuel retard.

6.2 Importance de l'ordre d'arrivée des commandes et des livreurs

L'algorithme implémenté dans *TrackFood* est très sensible à l'ordre d'arrivée des commandes, particulièrement sous sa forme statique. En effet, il a été expliqué plus haut que des solutions pouvaient même ne pas être prises en considération sans faire à une méthode de réparation. Cela signifie que dans le cas où une solution est possible, mais pas optimale, l'algorithme ne teste pas beaucoup d'alternatives. C'est tout à fait correct si l'on considère que la priorité est la rapidité du service, mais cela peut poser problème en terme de qualité.
Il aurait été possible d'envisager un certain tri avant d'insérer les commandes, comme les trier par temps de traitement requis, ou par capacité, ou d'autres possibilités encore.

De la même façon, l'algorithme va distribuer les livraisons aux livreurs de façon assez aléatoire, puisque le conteneur des livreurs est un *heap*[1]. Au départ, si deux livreurs sont au même dépôt et que l'algorithme tente de vérifier lequel des deux pourra effectuer une livraison, c'est simplement le premier qui aura été testé qui sera celui à effectuer la course, puisque aucun autre critère ne vient influencer ce choix.

Toujours concernant les livreurs, aucun tri n'est fait sur le nombre de livraisons effectuées dans la journée de travail, ce qui pourrait être un critère de choix. En effet, en conditions réelles, une entreprise de livraison souhaiterait sans doute favoriser plutôt l'utilisation de moins de livreurs pour éviter des frais supplémentaires, ou au contraire, favoriser la rapidité du service rendu. Ceci pourrait être modifié assez simplement en modifiant les critères de poids des chemins, qui seraient alourdis par la durée de l'ensemble des chemins au détriment de la durée d'un seul.

6.3 Simulation

Le projet comportait une partie importante quant à la vulgarisation du domaine scientifique traité. Un programme de simulation, constituée d'une interface graphique représentant les déplacements de livreurs en conditions réelles de circulation, a dû être déployé dans un but pédagogique. Lors du *Printemps des Sciences*, cette interface a permis de donner un coté attractif au sujet afin de réaliser une modélisation du problème sur écran. Le but étant de faciliter la compréhension du sujet pour un public varié, il a été possible de leur fournir une représentation visuelle du fonctionnement du problème abordé.

À l'heure actuelle, la simulation permet de mettre en place les connections entre le modèle qui s'exécute sur serveur et l'application sur mobile. Dès lors, celle-ci pourrait évoluer en un programme ciblant les compagnies de livraison. Traquant ainsi toutes commandes effectuées par le biais de l'application, bien qu'automatisées, des options de paramétrages pourraient voir le jour afin de modifier l'état des livreurs ou des restaurateurs. Ces manipulations influeront sur l'état des données utilisé par l'algorithme, et donneraient ainsi le contrôle à la compagnie de livraison.

6.4 Application mobile

En l'état actuel, l'application *TrackFood* n'est pas directement fonctionnelle pour une mise en pratique hors simulation, puisque cette partie du projet n'apportait pas d'intérêt pédagogique ni scientifique.

Il s'agit de l'interface propre au livreur. Le guidage de celui-ci, sous la forme de GPS, doit encore être réalisé. L'implémentation de l'envoi régulier des données relatives à la position d'un coursier utilisant l'application n'est pas finalisée. La simulation n'ayant pas eu besoin des données renvoyées par les livreurs, cette section n'a pas pu voir le jour.

L'application prend en compte la capacité actuelle du trafic. Par exemple, si celle-ci devient plus saturée, les données en seront modifiées. Toutefois, les restaurateurs et livreurs ne possèdent pas de données qui leur sont propres. L'intégration d'un profil permettrait d'indiquer une éventuelle perte de performance, et donc de temps, en fonction par exemple du jour de la semaine. Ces profils alimenteraient les données propres au *Machine Learning*.

TrackFood n'est pas une application qui s'auto-suffit. Son but est d'être employée par une compagnie de livraison ainsi qu'être au centre de la communication Client/Restaurant/Livreur. La rendre fonctionnelle n'est pas réalisable sans un déploiement important de la part de la compagnie. D'un point de vue implémentation, le modèle fonctionne. Celui-ci prend en compte les données envoyées par l'application client et est prêt à recevoir celles du livreur.

1. Heap : Structure ordonnée qui permet d'accéder très rapidement à la tête de liste https ://fr.wikipedia.org/wiki/Tas_%28informatique%29

Bibliographie

[Cordeau and Laporte, 2002] Cordeau, J.-F. and Laporte, G. (2002). A tabu search heuristic for the static multi-vehicle dial-a-ride problem. In *Transportation Research Part B : Methodological*, volume 37, page 579–594.

[Cordeau and Laporte, 2007] Cordeau, J.-F. and Laporte, G. (2007). The dial-a-ride problem : models and algorithms. In *Annals of Operations Research*, volume 153, pages 29–46.

[Deleplanque, 2014] Deleplanque, S. (2014). *Modélisation et résolution de problèmes difficiles de transport à la demande et de Lot-Sizing*. PhD thesis, Université Blaise Pascal - Clermont-Ferrand II.

[Haj Rachid et al., 2008] Haj Rachid, M., Ramdane Cherif, W., Bloch, C., and Chatonnay, P. (2008). Classification de problèmes de tournées de véhicules. *ResearchGate*.

[Korte et al., 2010] Korte, B., Vygen, J., Fonlupt, J., and Skoda, A. (2010). *Optimisation combinatoire : Théorie et algorithmes*, chapter NP-complétude, page 405. Springer.

[Laporte and Osman, 1995] Laporte, G. and Osman, I. H. (1995). Routing problems : A bibliography. In *Annals of Operations Research*, volume 61, pages 227–262.

[Pillac et al., 2012] Pillac, V., Gendreau, M., GuéretChristelle, and Medaglia, A. L. (2012). A review of dynamic vehicle routing problems. *European Journal of Operational Research*.

[Rego and Roucairol, 1994] Rego, C. and Roucairol, C. (1994). Le problème de tournées de véhicules : étude et résolution approchée. Technical report, Institut National de Recherhe en Informatique et en Automatique.

[Zhao, 2011] Zhao, X. (2011). *Algorithmes pour les problèmes de tournées à la demande*. PhD thesis, Université Blaise pascal – Clermont-Ferrand II.