Carpooling, a vehicle routing approach

Author : Corentin Mulders
Supervisor : Pierre Schaus
Readers : Florence Massen
          Bernard Lambeau

Thesis submitted for the
Master in Computer Science and Engineering
option Artificial Intelligence

2012-2013
Acknowledgements

I would like to thank my supervisor, Pierre Schaus for his help and advice. He was very helpful both for helping me to solve technical problems and for keeping me working with regular meetings. I would also like to thank Djengo, and Tanguy Goretti in particular, for their help. The presentation of their carpooling solution was very helpful to me to get an idea of where to start working. Finally, I thank Renaud Hartert, my fellow student, for his explanations about the vehicle routing problem in constraint programming.
Table of contents

1 Introduction ........................................ 3
   1.1 Problem Statement .............................. 3
   1.2 Structure of the document ..................... 4

2 Carpooling problem description ..................... 5
   2.1 What is carpooling? ............................ 5
   2.2 Literature review ................................ 5
      2.2.1 "A Matching-Algorithm based on the Cloud and Positioning Systems to Improve Carpooling" ......................... 6
      2.2.2 "A Decision-Support System for the Car Pooling Problem" ............ 7
      2.2.3 "Carsharing and Carpooling optimization : A 5 years research experience" 8
      2.2.4 "Optimization of trips to the university: A new algorithm for a carpooling service based on the Variable Neighborhood Search" ........ 9
      2.2.5 Literature review synthesis .................. 10
   2.3 Problem description ............................ 11

3 Tools used to solve the carpooling problem ....... 13
   3.1 Tools used ..................................... 13
      3.1.1 Scala ..................................... 13
      3.1.2 Akka ..................................... 14
      3.1.3 OscaR .................................... 14
      3.1.4 Google maps and Mapquest ................. 16

4 Solutions to the carpooling problem .............. 21
   4.1 Filtering and sorting method ................... 21
      4.1.1 Types of filters ........................... 23
      4.1.2 Sorting process ........................... 25
   4.2 Clustering method ............................. 27
      4.2.1 Clustering as a complete solution ........ 27
      4.2.2 Clustering as a semi-automated solution .... 29
      4.2.3 Clustering as a pre-processing step before using another method on the clusters separately ......................... 29
      4.2.4 Conclusions about clustering .............. 30
   4.3 Vehicle routing approach ....................... 30
      4.3.1 Vehicle routing problem definition ....... 30
      4.3.2 Adaptation of the carpooling problem as a vehicle routing problem ... 35
Chapter 1

Introduction

Carpooling, as opposed to going to work alone in one’s car, offers several benefits. These benefits can be financial, environmental, social or can simply be the reduction of the number of cars on the roads.

The financial benefits are obvious, if two people share a car for a journey, they also share the fuel costs. For frequent journeys, being able to reduce the costs can be very interesting.

The environmental benefits are linked to pollution generated cars. The most known effect on the environment of driving a car is the carbon dioxide emission. Carbon dioxide contributes to global warming. If more than one people ride in a car, the number of cars needed by those people is reduced, and so is the threat to the environment.

Social benefits depend on people. Some will prefer to be alone to go to work. On the other hand for some travelling in a car with someone else driving is an occasion to take a nap during their trip. For others who prefer to talk with their trip companions, being with other people to go to work can be a way to enjoy the trip.

The last benefit is obvious. Every person riding in a car with someone else reduces the number of cars doing the journey by one. This means that there will be less cars on the road, which reduces the probability of traffic jams. Reducing the number of cars on the road can also make the trip faster and less stressful for the driver.

Carpooling can happen spontaneously, between family members, friends or coworkers. However, when carpooling with acquaintances is not enough, services can be used to find partners.

1.1 Problem Statement

A lot of carpooling solutions already exist. Some of them are made by firms that target enterprises and sell them their solution for their employees’ uses. Examples of those solutions are the ones offered by Django \(^1\) or Karzoo \(^2\). Other solutions go directly to the users with

\(^1\) www.djengo.be
\(^2\) www.karzoo.be
free or paid web-based services. A lot of them can be found on the web. Examples are www.covoiturage.com, carpooling.com, carpool.be, www.covoiturage.fr, www.carpooling.fr, ...

Those solutions use more or less automated methods to help users find carpooling partners.

The goal of this thesis will be to implement different automated methods that can group users in carpools. Those methods will either be the ones used by some of the previous services, improvement of them, or completely different ones. This goal will be described with more details in section 2.

All of the source code produced to implement the various solutions is available at https://bitbucket.org/cmulders/carpoolingthesis under LGPL license.

1.2 Structure of the document

This document will be split into three main chapters. The first one will give a formal definition of carpooling and of the problem that will be solved in this thesis.

The second chapter will describe different tools that are used for the implementation of the solutions to the carpooling problem.

The third chapter will describe those solutions in detail. It will be subdivided into three main sections, each one corresponding to one type of method that can be used to solve the carpooling problem.
Chapter 2

Carpooling problem description

2.1 What is carpooling?

Carpooling is, by definition, car journeys with arrangements so that more than one person can travel in a car. This can be done either for commuting or for occasional long journeys.

This can happen spontaneously. For example if two colleagues find out they live not far from each other and decide to go to work together. Or carpooling can be achieved by using specialized tools to find people to share a ride with. An example of such tools are websites such as www.carpooling.fr where users can post offers and ride requests for others to see.

The options available for carpooling fall into distinct categories:

- Public websites such as the aforementioned www.carpooling.fr. Where people offer or find rides which can be free, based on a fictional currency from the website, or paid.
- Private websites for employees or students.
- Carpooling agencies with human operators.
- Public pick up points to find a ride without pre-arranging it.
- Spontaneous pre-arranged carpooling between people that know each other.

If we limit ourselves to automated tools, the first two options are to be considered. They are basically the same if we don’t consider the restrictive use of the second. There are several ways to build such tools depending on what is desired.

2.2 Literature review

There are several studies that have already been done on the broad subject of carpooling. The following ones will be presented:

Each one offers a different method to solve the carpooling problem. Those methods will be summarized in the following sections.

2.2.1 "A Matching-Algorithm based on the Cloud and Positioning Systems to Improve Carpooling"

In this paper [1], the idea is to try to find passengers among people that are linked to the driver on online social networks. The goal will be to develop a web platform named "Lift4U" that will offer an advanced carpooling solution.

Their algorithm is then the following : for a given driver with a starting point, a maximum seating capacity, and a maximum detour distance :

1. Compute the path from the driver starting point to the destination. Bing maps [5] is the service used to compute paths.

2. Look for candidates in social networks whose starting point belongs to a circle drawn around the driver starting point with a radius defined with regards to his maximum detour.

   If a candidate is found, recompute the path to pick him up, if this makes the detour bigger than the maximum detour allowed, discard the candidate.

   If more than one candidate are found, compute the paths for each one and pick the one causing the smallest detour.
3. If the maximum detour is not reached and the car is not full, go back to step 2 but draw the circle from the last candidate picked up.

These three steps are illustrated in figure 2.1.

In summary they simply filter the set of candidates using a maximum distance from the starting point, then choose the best one by computing every path.

The solution used to find the passengers to pick up is basic but the focus is on a good web integration, with a program connected with social services and offering lots of options to the user such as minimizing the $CO_2$ emissions and trip cost, or the travel time.

2.2.2 "A Decision-Support System for the Car Pooling Problem"

In the second paper [2], the authors focus on a way to decrease pollution. They see carpooling as a way to reduce the number of private vehicles generating it.

They start by separating two distinct cases. The daily car pooling problem (find people to pick up their colleagues to go to work every day) and the long term car pooling problem (in which each user is in turn driver and passenger). The user pools are then defined such that each user drives equally. The second case is the problem that they work on.

Since the authors’ objective is to reduce the pollution, they try to minimize the total distance travelled by all users, not the one within the pools. They see carpooling as a very efficient strategy to improve the sustainability of private mobility.

The method they use to select pools of users can be described as a 3-steps process:

1. **Data collection**: Gather the following informations:
   - A list of users with informations such as their geographical location and whether or not they have a car.
   - The location of the common destination of these users.
   - The map of routes with their performances.

2. **Clustering**: Cluster users using a similarity evaluation:
   - They build a similarity matrix using the Pearson correlation coefficient on the distances between users. Then they use this similarity matrix with the complete linkage clustering heuristic to build clusters. These clusters are made such as they correspond to the user pools that will share a ride.

3. **Routing**: In this last step, for each users pool, a ”current provider” is selected. He is the owner of the current shared car. Then a Travelling Salesman Problem is solved for the users of the group to find the best route to pick them all up.

   This basic process is then refined by tweaking some parameters. Here are some interesting things to add or modify:
   - Usage of a shortest or quickest strategy for the last step.
   - Usage of a different clustering heuristic.
• Modification of the similarity index used for clustering to introduce factors taking other attributes of the users into account, such as their age or hobbies. This allows to form more homogeneous groups.

• Set a threshold level for the similarity in the clustering heuristic. This way the vehicles are not always filled to their maximum capacity if other users are too far away.

2.2.3 "Carsharing and Carpooling optimization : A 5 years research experience"

As the title announces, two different problems are discussed in the third paper [3]. The carsharing problem, which consists of having a public pool of cars for people to use. And the carpooling problem, which is the only one we have interest in.

The authors describe the problem as a combinatorial optimization problem. They use the following variables:

• $C$ : set of users

• $K$ : set of the pools of users

• $x_{ijk}^h$ : binary variable = 1 iff arc $(ij)$ belongs to the shortest hamiltonian path for the driver $h$ in the pool $k$

• $y_{ik}$ : binary variable = 1 iff user $i$ belongs to pool $k$

• $\xi_i$ : binary variable = 1 iff user $i$ is not in any pool

• $p_i$ : path cost for user $i$ to go to his destination alone

• $t_{ij}$ : time to go from $i$ to $j$

• $Q_i$ : capacity of the car of user $i$
The problem definition given by the authors is similar to the following problem:

\[
\begin{align*}
\text{minimize} & \quad \sum_{k \in K} \sum_{h \in C} \sum_{(ij) \in C} t_{ij} x_{ij}^k + \sum_{i \in C} p_i \xi_i \\
\text{subject to} & \quad \sum_{j \in C \cap \{h\}} x_{ij}^k = y_{ik}, \quad i, h \in C; k \in K \\
& \quad \sum_{j \in C \cap \{0\}} x_{ji}^k = y_{ik}, \quad i, h \in C; k \in K \\
& \quad \sum_{k \in K} y_{ik} + \xi_i = 1, \quad i \in C \\
& \quad \sum_{(ij) \in A} x_{ij}^k = Q_h, \quad h \in C; k \in K \\
& \quad \sum_{i \in C \cap \{h\}} Y_i^k \geq Y_h^k, \quad h \in C; k \in K \\
& \quad \text{User } i \text{ has to be in pool } k \text{ if there is an optimal path } ij \text{ originated from } h. \\
& \quad \text{Paths must be continuous.} \\
& \quad \text{Sets } \xi_i \text{ to its designated value.} \\
& \quad \text{Car capacity limitation.} \\
& \quad \text{No group with only one user.}
\end{align*}
\]

User $i$ has to be in pool $k$ if there is an optimal path $ij$ originated from $h$.
Paths must be continuous.
Sets $\xi_i$ to its designated value.
Car capacity limitation.
No group with only one user.

The constraints represent the carpooling problem. And the objective function is a sum of two elements:

1. The sum for each pool of the average path time for each driver in this pool.
2. The sum of the path costs of users alone in their cars.

In summary, what is minimized is the total driving time for all the cars used.

To solve this a divide-and-conquer strategy is used. The users are sorted into smaller sets by a clustering. The algorithm used is k-means. This is necessary to avoid having unrealistic computation times for the minimisation.

Then a solution can be computed. Mixed Integer Programming is used for this. This solution is also compared to a simulation of the situation to ensure its efficiency.

2.2.4 "Optimization of trips to the university: A new algorithm for a carpooling service based on the Variable Neighborhood Search"

In this [4] case the problem analysis is targeted towards students. This means that the users can have very different timetables. They might want to travel with their friends or might be willing to use public transports, which means that they could be dropped at stations and so would have more than one destination.

The solution they developed is called the "PoliUniPool Project" and is a carpooling system that includes time windows, friend lists and blacklists of users to accommodate for enemies, as
well as for friends.

The method consists in solving an optimization problem with a linear combination of elements as objective function. These elements are the followings:

1. A formula to represent the gain in kilometers. The gain corresponding to the difference between the total distance driven without carpooling and the total distance driven with carpooling.
2. The matching coefficient, the number of users successfully placed in a car divided by the total number of users.
3. The level of service: used as a measure of the average detour time.
4. The friendship preferences.
5. The history of pools, to match the same people again if they were in the same pool before.

This objective function is then minimized by using variable neighborhood searches. This consist in finding local minimum using local search then shake to move to a state in the neighborhood. This shaking is done by selecting a random neighbor, neighbors are elements from the solutions set with a low distance heuristic measurement to the current solution. If the new optimum is better it replaces the old one, else another shake is done around the previous solution.

2.2.5 Literature review synthesis

From the 4 carpooling solutions that were analysed we can draw some synthetical features.

1. The solution from section 2.2.1:
   - Easy to compute with a selected user. Adapted to a website where the user is only interested in getting rides concerning him without waiting for computation.
   - No global optimisation.
2. The solution from section 2.2.2:
   - Has to be computed for all users at the same time.
   - Good global optimization.
   - The use of a clustering algorithm makes it difficult to add complex constraints. The customization is limited.
3. The solution from section 2.2.3:
   - Has to be computed for all users at the same time.
   - Good global optimization.
   - Customization is easy for constraints, as well as for the objective function.
4. The solution from section 2.2.4:
• Has to be computed for all users at the same time.

• Good global optimization.

• Limited customization. No constraints introduced, the only possible customizations are about the objective function. It can be hard to choose good coefficients for the minimized elements in the linear combination that is the objective function.

In summary, the first solution is tailored for a website usage where users want a fast answer to their request.

The other three are best suited for computation on a whole set of users at a time to have a solution with a lower total travel distance and with various constraints for the last two.

The situations that will be considered in this paper are closer to the second category. We will mostly try to compute carpooling groups for a whole set of users at a time.

2.3 Problem description

The carpooling problem that will be considered here can be now defined in details.

The approach that will be considered is one with a business/school point of view. The goal is to bring people with the same destination together. Different approaches will be proposed to work on computing all the routes. We will either try to group users separately then remove users from the available pool, or build all groups at the same time.

Since building random groups is not interesting for the users, the total distance driven by the cars will be to minimize. Another possibility would be to try to minimize the detours length of each conductor, but this wouldn’t necessarily lead to an optimum when summing them all.

The problem that will be solved is the following one:

• There is a pool of users with the following information available:
  – The departure location of the user.
  – Their destination, which is the same for each user.
  – Whether or not the user has a car, and if he has, its capacity.
  – Optional additional constraints such as a time window to leave home or to get to work, a maximum delay from detours for drivers, ...

• The goals are to build groups of users that will go to the destination together and to find the order in which the conductor picks up passengers within each group.

• This must be done while trying to minimize the sum of the duration of each trip.

• The users that have no car must be grouped with a user that has one.
This describes a general carpooling problem. Note that the return trips, users coming from the workplace and going back home, are symmetrical problems. The focus will be on building groups to go to work, the same ones can be used for the return trips, or the problem can be applied again for those since the results are valid regardless of the direction.

The optional additional constraints for the users mean that refinements of the definition are possible. Cases of the problem without such additional constraints will be considered, along with others more complex. Both have their applications since adding constraints to the problem is not free and may require more expensive models in terms of computation.
Chapter 3

Tools used to solve the carpooling problem

3.1 Tools used

This section will describe the main tools used to work on the different solutions tested for the carpooling problem.

First there is the programming language that was used, Scala [6]. Then Akka [7], a toolkit to build concurrent applications was also used. Then comes OscaR [8], a toolkit for solving Operations Research problems, which was used for its constraint programming solver and its visualisation capabilities. And finally the Google maps [9] and Mapquest [10] APIs were used to compute the trip durations between two points.

3.1.1 Scala

Scala is a programming language built on top of the java virtual machine. It is fully interoperable with java and has a functionnal programming component. It not only has the common features of java but has also a lot of other ones, some examples are:

- Anonymous functions
- Type inference
- Lists comprehension
- Lazy initialization
- Pattern matching
- Case classes
- Unified type system
- Optional parameters
Named parameters

Scala is designed to allow writing common programming patterns in a concise way. It was created by Martin Odersky from the "École Polytechnique Fédérale de Lausanne".

Why Scala was chosen

There were multiple reasons to choose Scala, here are the main ones:

- The complete interoperability with java was reassuring to someone used to this programming language. Moreover, Scala inherits from all the benefits of java, such as being multi-platform.
- OscaR, that will be described below, was first written in java but is now being translated to Scala and for this reason it was convenient to learn Scala, even though it could also be used from java.
- Expressivity: Scala allows to write code more easily. Mixing some imperative programming with the functional programming allows to keep some programming habits while benefiting from the elegant expressions of Scala. For example the use of the map and filter functions on collections allows to manipulate sets of data more easily than using loops that manually browse them.

3.1.2 Akka

Akka is a runtime library offering several tools to build concurrent, distributed applications. The part of Akka that was used is the actors and message passing library. These are used to access the web API of Google maps and Mapquest using actors. Akka allows to have a pool of workers that access the remote servers concurrently without having to do much work since the messages queue is managed by Akka. These actors can not only run in separate threads, they can also easily be made to run on different computers. This will be described in more details in the section 3.1.4.

3.1.3 OscaR

OscaR [8] stands for "Scala in OR", it’s a Scala toolkit designed to solve Operational Research problems. OscaR is an open source project and has an active team of developers. OscaR regroups various packages:

- Constraint Programming
- Constrained Based Local Search
- Linear (Integer) Programming
- Discrete Event Simulation
- Derivative Free Optimization
- Visualization
The features of OscaR used for the carpooling solution are the constraint programming solver and the visualizations.

**OscaR constraint programming component**

This component is used for an approach of the carpooling problem modeled as a constraint programming problem.

First the problem was translated in a set of variables and constraints to be used as a constraint programming model. OscaR allowed then to solve it using basic constraint programming as well as constraint programming with large neighborhood search. In our case the carpooling problem was seen as a constrained routing problem, this will be further discussed in section 4.3.

OscaR allows to implement easily search heurstics and relaxtion heuristics adapted to each particular problem.

**OscaR visualizations**

One of the visualization tools offered by OscaR is the possibility to display a map from OpenStreetMap in a JPanel. The user can drag the map and zoom in or out of it and different elements can be added to it:

- **Waypoints**: these can be used to represent points on the map, along with an optional text label.
- **Lines**: Allow to draw straight lines on the map between two points.
- **Paths**: Connect two points following the roads on the map.

As a demonstration, figure 3.1 represents three points on a map, one has a label, a pair is linked by a straight line and another is linked by a path following roads.

![Figure 3.1: OscaR map visualizations: waypoints, line and path.](image)
All of these elements can be easily manipulated by the application, which allows to represent a carpooling solution more intuitively on a map. It also makes it easy to represent a problem such as a travelling salesman problem on a map and, for example, highlight the parts of the path that are being modified.

3.1.4 Google maps and Mapquest

Google maps [9] and Mapquest [10] both offer a web API allowing to easily compute the time needed to get from one place to another by car. This could also be done locally with the appropriate geographical data but using web services is easier since there is no need to worry about keeping the data up to date with roadworks and other traffic modifications.

This section will describe both services and compare their offers. It will also describe how they were used in our carpooling solution.

Google maps

API description

Google maps is not only a website and service from Google that allows to visualize maps and plan itineraries. It also offers a web API that makes these services and other ones available. The part of this API that we are interested in is the "Google Maps API Web Services", and more precisely the "Directions" and "Distance Matrix" APIs since those are the ones that allow to compute the duration of a car journey between two points.

This API is used by opening an URL in which the request is defined and which gives an output either in JSON or XML format.

Here are two examples of such requests :

- Directions request, for an itinerary between Ottignies and Bruxelles passing through Wavre a directions request could be the following :

  origin=ottignies&destination=bruxelles&waypoints=wavre

Note that either addresses (in this case, only the city name) or coordinates can be used to define a point, in this example we used city names for readability.

Accessing this url will get as answer an xml file containing informations about the itinerary. It would not be convenient to show the whole file here because it is quite big since it contains many details intended to be used by a GPS device. Here is the part of the response we are interested in :

```xml
<DirectionsResponse>
  <status>OK</status>
  <route>
```

16
Among the retrieved information, we are able to collect the duration of each part of the itinerary, in this case from Ottignies to Wavre and from Wavre to Bruxelles.

- **Distance Matrix request**, this API is used to get a matrix of distances or durations between a set of origin points and a set of destination points, for example here is a request for directions to Bruxelles and Wavre starting from Ottignies:

  ```
  ```

  In this case the received response will only contain the durations and the distances without the routing informations given by a Directions request.

**Usage limits**
The Google Maps API can be used either for free or with a Business license, those two different licenses have different usage limits for the service.

Here are the limits associated to the parts of the API that we use:

<table>
<thead>
<tr>
<th></th>
<th>Google Maps API</th>
<th>Google Maps API for Business</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directions : requests/day</td>
<td>2500</td>
<td>100000</td>
</tr>
<tr>
<td>Directions : waypoints/request</td>
<td>10</td>
<td>23</td>
</tr>
<tr>
<td>Distance Matrix : elements/query</td>
<td>100</td>
<td>625</td>
</tr>
<tr>
<td>Distance Matrix : elements/10s</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>Distance Matrix : elements/day</td>
<td>2500</td>
<td>100000</td>
</tr>
</tbody>
</table>

*(An element in the case of the Distance Matrix API corresponds to an element of the output matrix)*
We quickly notice that, if we use the Directions API without waypoints, we can compute the same number of durations as with the Distance Matrix API each day. However using waypoints as a trick to get more than one duration by request allows us to get more durations.

In the case where we want to compute durations for non-sequential (the starting point of one is not the ending point of another) itineraries, which is the worst scenario, we can compute \( \frac{\text{number of waypoints}}{2} \) durations in one Direction request.

In other words, with the free version one Directions request allows to compute 6 durations at once and in the business version 12 durations can be computed at once. This is why it is better to use the Directions API instead of the Distance Matrix API, even though the later seemed more suited to our needs.

**Mapquest**

**API description**

Mapquest is a very similar service to Google Maps. They offer both a website where you can visualize itineraries on a map and a web API. The characteristic of mapquest is that, on top of being able to use it normally with their underlying proprietary maps, it can be used with the OpenStreetMap mapping data. This is done simply by adding "open." at the start of the normal mapquest API url.

This API is used the same way as Google Maps, it has a `directions` service which includes an option to get a matrix of distances.

**Usage limits**

The mapquest API can be used either with licensed data for free, or with licensed data with a fee, or with OpenStreetMap data for free. The limitations for the parts of the service that are of interest to us are the same for the licensed paying solution and the OpenStreetMap one.

So we will only compare the free solutions:

<table>
<thead>
<tr>
<th></th>
<th>Licensed Data</th>
<th>OpenStreetMap Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directions: requests/day</td>
<td>5000</td>
<td>( \infty )</td>
</tr>
<tr>
<td>Route Matrix: elements/day</td>
<td>5000</td>
<td>( \infty )</td>
</tr>
</tbody>
</table>

There is no clear defined limit of the number of requests accepted by second but mapquest specifies that they reserve the right to limit access to prevent service degradation.

The fact that the service can be used without limits with OpenStreetMaps makes it for us the more interesting option to use. The licensed data might be more precise but for our testing needs the unrestricted option is convenient.
Google Maps and Mapquest comparison

To choose which service to use there were two aspects to consider. The first was to know whether or not using mapquest with OpenStreetMap would yield significantly different results from using Google Maps. The assumption being that Google Maps precision is the frame of reference, mapquest had to give similar results to be able to use it safely.

The second thing to compare was the time taken to retrieve sets of durations from both services.

Precision and general comparison:
The test conducted consisted of computing 100 durations using both services.

Two values were computed:
• $averageDiff = \text{the mean of the differences between the results from each service.}$
• $averageDuration = \text{the mean of the durations obtained from Google Maps.}$

The ratio between those two was used as measurement of the difference between the two map services:

$$\text{difference} = \frac{averageDiff}{averageDuration}$$

The average $difference$ value measured during experiment is around 0.05. This means that the results from both services are 95% similar. In other words, mapquest can reasonably be used since its results are close to those obtained through Google Maps.

However there are some downsides to the use of Mapquest that were noticed when using their API on large data sets:

1. Mapquest geocoding is not as good as Google Map’s. This means that addresses must be written in full or coordinates must be used.

2. When using OpenStreetMaps, Mapquest was sometimes unable to find a route between two coordinates pairs if one of the points was not close enough to a road. To avoid being stuck, the fix that was implemented when this problems occurs (which can be frequent with randomly generated points) was to decrease the precision of the coordinates. This allows mapquest to find some route, even though there is a loss of precision.

Time comparison:
One of the uses of the map services is to compute the exact detour durations when someone goes out of his way to pick up a passenger. This test was based on the implementation of such a computation.

What the test does, is that it computes the detour duration for a set of detours using Google Maps then Mapquest. And it records the time taken by each. The test was run for
Google Maps vs open Mapquest

It is important to note that the methods implemented to retrieve the durations from each service are the same. The same number of threads is used and the same waiting time if a connection is refused.

The results from this test can be seen on figure 3.2. The results show that the access times are not very different between the two. Google Maps seems more efficient for a low number of requests while mapquest is better at handling lots of concurrent requests.

Conclusion:
Mapquest with OpenStreetMaps is found to be as fast as Google Maps. It may be a little bit more unreliable since it was unable to compute some durations without decreasing the level of precision of the coordinates. However the fact that there is no limit to its use is interesting since it can be very well used as a fallback when Google Maps refuses the connections.
Chapter 4

Solutions to the carpooling problem

In this chapter, three different solutions to the carpooling problem will be described. These solutions were implemented and tested against each other. The implementations of all the solutions as well as the code used to compare them are available online at https://bitbucket.org/cmulders/carpoolingthesis.

The first solution is a classic one that was already mentioned in the section 2.2. It consists in finding the best passenger or driver for an user by filtering candidates from the database and then selecting the best one by computing the detour durations for each. The second solution will be to use clustering algorithms to group the users by car. The third one will be to use a constraint programming solver on a vehicle routing based model.

4.1 Filtering and sorting method

The filtering and ordering method is the most common way to group users for carpooling, especially for web-based solution where the user wants immediate results for his request. It is used to form pairs of users, a driver and a passenger. It can also be adapted to having more than one passenger by using it again with an existing pair.

The algorithm consists in applying two steps to find a match for a user $u$:

1. **Filtering step**:
   This step selects a set of candidates from the database. The filtering can be done using a combination of conditions on candidates. The most important of these conditions being that the distance between the candidate journey and the journey of the user $u$ must be under a chosen threshold. Filters will be described more in details in section 4.1.1. An example of this filtering of candidates for the journey of a user $u$ to a destination is shown in figure 4.1.
2. Sorting step:
   Now, for each of the candidates selected in the previous step, the duration of the detour made by the driver to pick up the passenger is computed. Then candidates are sorted using this value. In a fully automated solution the candidate with the shortest detour duration is then chosen. If the solution is not fully automated the ordered list is presented to the user $u$ so that he can choose a candidate.

An example of this sorting on the previously filtered candidates is shown in figure 4.2, in this example the user $u$ is considered to be a driver.

To solve a carpooling problem for a whole set of users, this method can be used on each
user from the set separately. Matched users are then removed from the set.

4.1.1 Types of filters

The strength of this method is its filtering step. During this step, various filters can be applied to stand for real world constraints. The type of distance used to discard users that are considered too far away can also vary, two different distances will be described.

First we will talk about the use of distance measures for the filtering. Remember that we consider that each user has the same destination. The goal when using a distance filter is to eliminate users that will be too far away to be interesting candidates. To do this, we define distance measures between journeys:

• **Starting point distance**:
  We can get a first idea of this distance when we compare two journeys by simply measuring the geographical distance between their starting points.

  This distance could either be the distance by routes or a distance as the crow flies depending on the degree of precision that is required. But for large scale applications it is only realistic to use flying distance since computing by route distances for the whole set of users can be slow. A representation of this distance using flying distances is given on figure 4.3.

  The use of this distance has its strengths and weaknesses. It is a good distance to use if we want an efficient solution in terms of fuel consumption. This is because selecting candidates that have a close starting point will ensure that the car contains most of the time more than one person.

  However, it can be a bad distance to use since it will ignore candidates that would be on the way but not close from the starting point. The second distance tries to fix this problem.

![Figure 4.3: Starting point distance between journeys a and b.](image)

• **Angle distance**:
  This distance tries to make sure that candidates that require a small detour length for
the driver are selected. To do this we consider the whole journey instead of only using the starting points.

What will be measured is the cosine of the angle at the destination between the lines drawn from each starting point to the destination. Since the cosine of the angle will be bigger for smaller angles, it gives us a measure of the similarity between two journeys. To get a measure of the distance, we simply use \(1 - \cos(\text{angle})\). This is represented on figure 4.4

This distance will be better to find a candidate without having to go out of the way too much. However it might not be interesting for people who want to save fuel. If the driver is being paid by the passenger depending on the distance ridden, then he would prefer to have a passenger for most of the drive.

![Figure 4.4: Angle distance between journeys a and b.](image)

Depending on the type of distance chosen, the threshold for the distance filtering can be chosen in different ways. It can be a fixed value, or it can be a function of the distance to the farthest element, or of the distance to the destination for starting point distances.

Using a distance filter is needed to reduce the number of candidates. However these are not the only interesting filters. Some other filters are needed for the solution to work:

- Only select people that can be drivers when the user \(u\) can’t. In the opposite case, only select people that can be passengers when the user \(u\) can only be a driver.
- Only select people that are not yet matched with someone else.

Other optionnal filters can be used to improve the solution such as the two following ones:

- Filter people based on time windows if the users want to specify time intervals during which they can leave home.
- Allow users to review users they rode with before. Then use this compatibility value between users to only select users that like each other.
4.1.2 Sorting process

The sorting step seems to be very simple. We have a set of selected candidates and for each of these candidates we want to compute the distance to get to the destination. Depending on who the driver is we then compute the following:

- If $u$ is the driver: we compute the distance from $u$'s starting point to the destination with the candidate’s starting point as waypoint.

- If the candidate is the driver: we compute the distance from the candidate’s starting point to the destination with the $u$’s starting point as waypoint.

Then when we know this distance $\text{distanceCarpool}$, the detour length can be directly computed, assuming we also know the distance $\text{distanceAlone}$ to go from the driver’s starting point to the destination without passing by any waypoint:

$$\text{detour} = \text{distanceCarpool} - \text{distanceAlone}$$

However, computing these distances can be the most problematic part of the whole filtering and ordering process. As seen in section 3.1.4, using web services can be slow. Using a local solution to compute shortest paths with map files has a cost too since maps need to be maintained and computations would not be instantaneous either.

To address the problem of the computation time of the ordering step, two solutions were considered. Those two can also be combined.

- The first solution is to try to avoid computing the distances for each of the candidates selected.

  This can be considered for a fully automated solution to select the best candidate without worrying about the order of the others. It could also be used for a partially automated solution as a refinement of the previous filtering if we want to use it to compute the top best distances and discard the other ones.

  The idea is to use an admissible heuristic\(^1\) to guess the distance. An obvious choice for such a heuristic would be the flying distance. It never overestimates the real distance that will be covered by roads.

  Instead of simply computing all the distances by road, we apply the following algorithm:

  1. Compute the straight line distance for each candidate.
  2. Sort the candidates using these heuristic distances.
  3. For the best one only, compute the distance by road and use it to replace the flying distance.
  4. Re-sort the list if needed since one of the distances was modified.
  5. If the distance by road of the first candidate in the ordered list is already computed, select this candidate. Else repeat from step 3.

\(^1\)As a reminder, a heuristic function is admissible if it is optimistic. In other words if its value is never greater than the real value of what it approximates.
This guarantees that the selected candidate is the one with the smallest distance. Even if the worst case number of requests to a web service stays the same, it can avoid to have to compute all the distances by road.

Figure 4.5 represents the different steps of the execution of this algorithm on a theoretical example.

<table>
<thead>
<tr>
<th>user</th>
<th>flying distance</th>
<th>road distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 user1</td>
<td>11</td>
<td>?</td>
</tr>
<tr>
<td>2 user2</td>
<td>9</td>
<td>?</td>
</tr>
<tr>
<td>3 user3</td>
<td>7</td>
<td>?</td>
</tr>
<tr>
<td>4 user4</td>
<td>21</td>
<td>?</td>
</tr>
<tr>
<td>5 user5</td>
<td>15</td>
<td>?</td>
</tr>
</tbody>
</table>

(a) Step 1 : compute flying distances

<table>
<thead>
<tr>
<th>user</th>
<th>flying distance</th>
<th>road distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 user3</td>
<td>(7)</td>
<td>10</td>
</tr>
<tr>
<td>2 user2</td>
<td>9</td>
<td>?</td>
</tr>
<tr>
<td>3 user1</td>
<td>11</td>
<td>?</td>
</tr>
<tr>
<td>4 user5</td>
<td>15</td>
<td>?</td>
</tr>
<tr>
<td>5 user4</td>
<td>21</td>
<td>?</td>
</tr>
</tbody>
</table>

(b) Step 2 : sort

<table>
<thead>
<tr>
<th>user</th>
<th>flying distance</th>
<th>road distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 user2</td>
<td>9</td>
<td>?</td>
</tr>
<tr>
<td>2 user3</td>
<td>(7)</td>
<td>10</td>
</tr>
<tr>
<td>3 user1</td>
<td>11</td>
<td>?</td>
</tr>
<tr>
<td>4 user5</td>
<td>15</td>
<td>?</td>
</tr>
<tr>
<td>5 user4</td>
<td>21</td>
<td>?</td>
</tr>
</tbody>
</table>

(c) Step 3 : compute road distance

<table>
<thead>
<tr>
<th>user</th>
<th>flying distance</th>
<th>road distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 user2</td>
<td>(9)</td>
<td>12</td>
</tr>
<tr>
<td>2 user3</td>
<td>(7)</td>
<td>10</td>
</tr>
<tr>
<td>3 user1</td>
<td>11</td>
<td>?</td>
</tr>
<tr>
<td>4 user5</td>
<td>15</td>
<td>?</td>
</tr>
<tr>
<td>5 user4</td>
<td>21</td>
<td>?</td>
</tr>
</tbody>
</table>

(d) Step 4 : re-sort

<table>
<thead>
<tr>
<th>user</th>
<th>flying distance</th>
<th>road distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 user3</td>
<td>(7)</td>
<td>10</td>
</tr>
<tr>
<td>2 user1</td>
<td>(11)</td>
<td>?</td>
</tr>
<tr>
<td>3 user2</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>4 user5</td>
<td>15</td>
<td>?</td>
</tr>
<tr>
<td>5 user4</td>
<td>21</td>
<td>?</td>
</tr>
</tbody>
</table>

(e) Step 5 : back to step 3 : compute road distance

<table>
<thead>
<tr>
<th>user</th>
<th>flying distance</th>
<th>road distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 user3</td>
<td>(7)</td>
<td>10</td>
</tr>
<tr>
<td>2 user1</td>
<td>11</td>
<td>?</td>
</tr>
<tr>
<td>3 user2</td>
<td>(9)</td>
<td>12</td>
</tr>
<tr>
<td>4 user5</td>
<td>15</td>
<td>?</td>
</tr>
<tr>
<td>5 user4</td>
<td>21</td>
<td>?</td>
</tr>
</tbody>
</table>

(f) Step 4 : re-sort

<table>
<thead>
<tr>
<th>user</th>
<th>flying distance</th>
<th>road distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 user3</td>
<td>(7)</td>
<td>10</td>
</tr>
<tr>
<td>2 user1</td>
<td>(11)</td>
<td>?</td>
</tr>
<tr>
<td>3 user2</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>4 user5</td>
<td>15</td>
<td>?</td>
</tr>
<tr>
<td>5 user4</td>
<td>21</td>
<td>?</td>
</tr>
</tbody>
</table>

(g) Step 5 : best element is selected

Figure 4.5: Best candidate selection algorithm example

- The second thing to do to speed up the access to web services is to use threads to access the web services. Optionally, proxies can also be used to bypass limits on the number of requests by IP address by second.

This seems obvious, the web request takes time so we want to use the time instead of having a single threaded application where nothing happens during the requests.

This was implemented using akka [7] actors that perform the web requests. Since akka also allows its actors to be distributed on several different computers, using multiple IP
addresses is the same as distributing the work over multiple threads from the application point of view.

4.2 Clustering method

The second idea to solve the carpooling problem is to see it as a clustering problem. What we basically want to do is to build clusters of users. Each of these clusters corresponding to a car.

A problem with this approach is that we need to limit the size of the clusters to accomodate the fact that cars have a limited capacity. We could also simply build clusters without such a limit and then consider that users within this cluster can use more than one car. A third possible use of clustering would be to use it as a preprocessing step to separate big sets of users into smaller sets. This would be a more efficient way to split users than doing it randomly.

These three different uses of clustering for the carpooling problem will now be discussed more in details.

4.2.1 Clustering as a complete solution

Using clustering as a complete solution means that each cluster must correspond to a car.

The problem here is to limit the size of the clusters, it is something that a basic k-means can't do. Two approaches were considered to do this :


As a reminder, the k-means algorithm is based on a repetition of two steps until the solution doesn’t improve anymore. After randomly generating initial centroids, those steps are :

1. **Assignment** of the elements to clusters : Assign each element to the cluster corresponding to the closest centroid.

2. **Update** of the centroids locations : Recompute the value of the centroid of each cluster, the new centroid must correspond to the center of its cluster.

The step that will be modified to accomodate size limits is the assignment step. We simply want to assign a user to a cluster only if this cluster size has not reached the limit. The modified k-means algorithm is :

1. **Assignment** of the elements to clusters. Assign each element to the cluster corresponding to the closest centroid that is not full yet. To do this :

   (a) Compute the distances to each centroid $c$ for each element $e$ to place in a cluster.

   (b) Sort the $(c, e)$ pairs with regards to the corresponding distance previously computed.
(c) Iterate on the sorted pairs. For each of those, if the cluster corresponding to \( c \) is not full : Place \( e \) in it and remove other occurrences of \( e \) from the sorted pairs. If it is full, do nothing.

2. **Update** of the centroids locations : Recompute the value of the centroid of each cluster, the new centroid must correspond to the center of its cluster.

To use this modified k-means on the carpooling problem, we still need to know how to compute the distances and the centroids. A simple way to do this is to use the starting points of the users. The used distances will be the geographical flying distance between those. The centroids can be placed at center of each cluster of points.

Figure 4.6 shows the clusters obtained by applying this modified k-means algorithm to a pool of 50 users with a maximum car capacity of 3, driver included.

![Figure 4.6](image.png)

**Figure 4.6:** Results of a modified k-means algorithm applied to 50 users, with a maximum cluster size of 3.

Once the clusters are found, the choice of who is the driver and the pick up order can be determined by applying a Travelling Salesman Problem to the small clusters. Another possibility is to simply compute all the detour lengths. It is easy since the number of users in a cluster will never be higher than the number of places in a car.

Note that since this modified version of the k-means algorithm has limited size clusters, the value of \( k \) must have a minimal value. We need \( k \geq \frac{\text{number of users}}{\text{maximum car capacity}} \). If we want all cars to be filled, we can choose \( k = \frac{\text{number of users}}{\text{maximum car capacity}} \).

- Use a hierarchical clustering algorithm, preferably one that results in a balanced tree. Then "cut" this tree so that the clusters under the cut are all under the limit size.

The hierarchical clustering algorithm that was implemented is a bottom-up algorithm. Basic clusters consist of only one user each. Then clusters are merged in pairs until all the clusters have been merged.
Ward’s method [12] is used to choose which clusters to merge, the algorithm uses a distance matrix containing the distances between each pair of clusters. The pair with the smallest associated distance is selected, merged and its distances to the other clusters are added to the matrix.

To limit the size of clusters, it is possible to cut the clusters tree horizontally. The level at which it is cut will contain clusters of limited size. This is shown on figure 4.7. Notice that in this figure the tree is roughly balanced. Having an heavily unbalanced tree, for example if at the root we have a cluster containing only one user to the left, and a cluster with 9 users to the right, reduces the effectivity of the cutting since it is more likely that the cut will generate clusters with few elements.

![Dendogram obtained for a hierarchical clustering on 10 users.](image)

The rough balancing of the tree is done by modifying the distance formula for new clusters. The basic distance is to use the average of the distances of the paired clusters. Adding a penalty to this average allows to prioritize clustering elements that are in smaller clusters.

4.2.2 Clustering as a semi-automated solution

In this case the idea is to group users into clusters then let them organize themselves within those.

Doing this is easy, since there is no size limit to take into account for the cluster. We can use a simple k-means algorithm. The only thing is to choose a distance. The distances available to compare different journeys were already described in section 4.1.1.

Once the clustering is done, the users receive the list of candidates with which to travel and choose from there.

4.2.3 Clustering as a pre-processing step before using another method on the clusters separately

It is the same idea as the previous one, except that instead of letting users choose we then use an automated method to decide who rides with who. This automated method can be one from sections 4.1 and 4.3 or even another clustering but with size limits.
4.2.4 Conclusions about clustering

These three ways to use clustering methods on the carpooling problem show that it is an interesting tool for this kind of problems.

However it has a big drawback. Clustering algorithms are based on a binary comparison of elements. In the case of the carpooling problem, comparing elements two by two is not always good enough. For cases where a driver picks up more than one passengers, we might need to be able to use informations about the modified route to choose the second passenger.

Another problem when using clustering as a complete solution is that it is not flexible enough to be able to customize the solution. For example we can’t easily specify a maximum detour length, so we might have drivers that have to go far to pick up multiple passengers. The last method proposed to solve the clustering problem will allow more control over what is wanted.

4.3 Vehicle routing approach

The third method to solve the carpooling problem will use constraint programming. The idea is to build a constraint programming model with all the users and their constraints. Then this model can be solved to minimize the total duration of the journeys. Everything is computed at once, by opposition to computing the detours for one user only in the filtering and sorting method from section 4.1. This means that this method would not be efficient for public websites where people want fast results for their journey only.

To build this constraint programming model we can start from the vehicle routing problem which is well documented.

4.3.1 Vehicle routing problem definition

The vehicle routing problem consists in finding routes for a set of vehicles that must visit a set of locations. These vehicles start and finish their travels at a given location. It is an optimisation problem in which this must be done while minimizing the total distance. Figure 4.8 represents a solution to such a problem.
Figure 4.8: VRP : 4 vehicle and 16 customers example

The motivation for this problem is to serve customers with delivery vehicles based in a depot.

A precise formulation of the problem is given by the Handbook of Constraint Programming [13] for an integer linear programming approach. This formulation can also easily be extended to take time windows into account.

The set of $n$ customers is represented by $C = \{1...n\}$. To this we add the depot at route start 0 and the depot at route end $n + 1$. The set of locations obtained is $N = C \cup \{0, n + 1\}$. The set of $m$ vehicles is represented by $M = \{1...m\}$. A few constants are used:

- $c_{ij}$ The cost of travelling from $i$ to $j$ with $i,j \in N$.
- $\delta_{ij}$ The distance from $i$ to $j$ with $i,j \in N$.
- $r_i$ The demand of customer $i \in N$. ($r_0 = 0, r_{n+1} = 0$)
- $Q_k$ The capacity of vehicle $k \in M$.

The decision variables are $x_{ijk} \forall i,j \in N, k \in M$:

$$x_{ijk} = \begin{cases} 1 & \text{if vehicle } k \text{ travels from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$
The VRP problem formulation with these variables is:

\[
\text{minimize } \sum_{k \in M} \sum_{i \in N} \sum_{j \in N} c_{ij} x_{ijk} \tag{4.1}
\]

subject to

\[
\sum_{k \in M} \sum_{j \in N} x_{ijk} = 1 \quad \forall i \in C \tag{4.2}
\]

Exactly one vehicle leaves each customer toward only one next destination.

\[
\sum_{i \in C} r_i \sum_{j \in N} x_{ijk} \leq Q_k \quad \forall k \in M \tag{4.3}
\]

Total customers demand for a vehicle can’t exceed its capacity.

\[
\sum_{j \in N} x_{0jk} = 1 \quad \forall k \in M \tag{4.4}
\]

Every vehicle leaves the starting depot once.

\[
\sum_{i \in N} x_{ihk} - \sum_{j \in N} x_{hjk} = 0 \quad \forall h \in C, \forall k \in M \tag{4.5}
\]

Every vehicle arriving at a customer’s location leaves it.

\[
\sum_{i \in N} x_{i(n+1)k} = 1 \quad \forall k \in M \tag{4.6}
\]

Every vehicle arrives at destination depot once.

\[
\sum_{i,j \in S} x_{ijk} \leq |S| - 1 \quad \forall S \subseteq C \tag{4.7}
\]

For every subset of customers, the number of internal arcs is lower than the size of the subset. Avoids any cycle not including the depot.

**Constraint programming modelization of the VRP**

The previously described problem can be solved with constraint programming. We still have the sets of vehicles \( M = \{1...m\} \) and customers \( C = \{1...n\} \).

We will try to find an unique circuit in which each customer is visited once. For a single vehicle this could simply be done by making sure the circuit visits each customer and the depot. To take into account that there are multiple vehicles, we will replicate the depot in the model to have the same number of virtual depots as the number of vehicles. To accommodate the fact that a vehicle leaves and returns to the depot, these virtual depots are duplicated into two sets:

- the starting points, containing the first visit of each vehicle: \( F = \{n+1...n+m\} \)
- the destinations, containing the last visit of each vehicle: \( L = \{n+m+1...n+2m\} \)

An example of this modelization with duplicated depots can be seen on figure 4.9. A particular vehicle’s route is then made of a starting point from \( F \), some customers from \( C \) and a destination from \( L \). Let call the set of all possible visit locations \( V = C \cup F \cup L = \{1...n+2m\} \).
The constraint programming modelization of the VRP uses two arrays of variables to build the circuit, pred and succ:

- \( \text{pred}_i \in V \) with \( i \in V \) represents the predecessor of \( i \) in the circuit.
- \( \text{succ}_i \in V \) with \( i \in V \) represents the successor of \( i \) in the circuit.

One of these would be sufficient to model the VRP problem, but using both helps to prune the search space because of the redundancy in the constraints that will apply to them. A third array of vehicles is needed to track the vehicle servicing each customer, vehicle:

- \( \text{vehicle}_i \in M \) with \( i \in V \) represents the vehicle visiting \( i \).

A fourth variables array can be used to track vehicles load if the capacity of vehicles is limited. It is used along with the maximum capacities and quantities of goods picked up (or dropped if negative) at a location. \( Q_v \) represents the maximum capacity of vehicle \( v \) and \( r_i \) represents the quantity of goods picked up at location \( i \).

- \( \text{load}_i \) with \( i \in V \) represents the vehicle load after visiting \( i \). This variable as a large domain only using bounds consistency on it is recommenced.
The objective function that will be minimized is the total duration (the drive duration is proportionnal to the distance in most cases and is more relevant when not) driven by the vehicles. Let assume that the duration between locations \(i\) and \(j\) are stored in \(\delta_{i,j}\) \(\forall i, j \in V\). The objective function is:

\[
TotalDuration = \sum_{i \in V \setminus F} \delta_{\text{pred}_i, i}
\]

The last thing needed is the set of constraints applied on these variables:

1. \(\text{pred}_i \neq \text{pred}_j\) \(\forall i, j \in V, i < j\)
   Makes sure that each location is visited and every vehicle arriving at a location leaves it, corresponds to the equations 4.2, 4.4, 4.5 and 4.6 from the linear integer formulation.

2. \(\text{NoCycle}(\text{pred}_i\ \forall i \in V \setminus F)\)
   Specialized constraint, avoids cycles without any depot location in the path.
   It operates by storing two values for each location: \(b_i\) and \(e_i\). These represent the locations corresponding to the beginning and the end of the chain of locations in which \(i\) is. These two variables are only kept up to date for the first and last elements of the chains since elements in the middle are already bound anyway. When a value is bound to \(\text{pred}_i\) \(b_B\) is updated for \(B = b_{\text{pred}_i}\) and \(e_E\) is updated for \(E = e_i\). The constraint to avoid loops is that, after that, \(\text{pred}_B \neq E\).

3. \(\text{succ}_{\text{pred}_i} \neq i\) \(\forall i \in V \setminus F\)
   To ensure consistency between \(\text{succ}\) and \(\text{pred}\) variables.
   To apply this, an element constraint is needed. The element constraint works by using a new variable: \(\text{succ}_{\text{pred}_i} \neq i\) is applied by using an implicit variable \(z: z = \text{succ}[\text{pred}_i] \land z = i\).

4. \(\text{vehicle}_i = \text{vehicle}_{\text{pred}_i}\) \(\forall i \in V \setminus F\)
   Assigns vehicles by making sure that the vehicle visiting \(i\) is the same as the one visiting its predecessor, except for starting points.

5. \(\text{load}_i = \text{load}_{\text{pred}_i} + r_i\) \(\forall i \in V \setminus F\)
   Updates vehicle load for each visit using predecessors.

6. \(\text{load}_i = \text{load}_{\text{succ}_i} - r_{\text{succ}_i}\) \(\forall i \in V \setminus F\)
   Updates vehicle load for each visit using successors.

7. \(\text{load}_i \leq Q_{\text{vehicle}_i}\) \(\forall i \in V\)
   Makes sure maximum capacities are not exceeded.

This set of constraints corresponds to a VRP model with limited vehicles capacity. Time windows can easily be added with new variables to track the arrival time at each customer.

In practice, when implementing this model in OscaR [8], the specialized circuit constraint will be used on \(\text{pred}\) and \(\text{succ}\) instead of the two first constraints. This constraint forces the path made by these variables to be an hamiltonian cycle.

Vehicle routing problems are NP-hard, it is important to find efficient heuristics if we want interesting results. Thankfully the vehicle routing problem is well documented and efficient.
search heuristics exist. The one that will be used when trying to solve the carpooling problem as a VRP will be presented in section 4.3.2.

4.3.2 Adaptation of the carpooling problem as a vehicle routing problem

The model presented for the vehicle routing problem is similar to the type of carpooling problem that we try to solve. The depot in the VRP reminds of common destination of the users in the carpooling problem. This similarity between the two allows to extend the VRP to solve carpooling problems. Let’s call such extended models carpool vehicle routing problems (CVRP).

The main difference between the vehicle routing problem and the carpooling problem is that vehicles are the cars of some customers. This means that they do not start at the depot but at the location of the first customer of a chain.

The trick that will be used to transform the carpooling problem into a vehicle routing problem is to set all the distances from starting locations to users to 0. This makes the first trip of each vehicle from the VRP free. These free trips correspond to trips that simply don’t happen in the case of a carpooling problem. An example of such a transformed problem can be seen on figure 4.10. The complete solution, including the greyed free trips, corresponds to the solution of VRP. But if we remove the free trips, we get the solution to a carpooling problem.

![Figure 4.10: Carpooling problem as a VRP, free trips.](image)

The second thing to take into account for the carpooling problem is the number of places in a car. This is the exact same thing as what is modelized by the maximum capacity of vehicles and the customers demands for the VRP. In the case of the carpooling problem, the maximum capacity of vehicles is the number of places in the car of the driver. A simplification of the
model is to use a fixed value for this for all the drivers. For example we could assume that all of them have at least 4 places in the car and can pick up 3 passengers. Once the maximum capacity of vehicles is fixed, the demand of each user can be set to −1 since one user is picked up each time.

The third thing to set is the number of vehicles. Its is set to the number of users. By doing this, the existence of a solution is assured and unused vehicles will simply correspond to vehicles going straight from a starting point to a destination point.

**Constraint programming modelization for carpooling**

Since the carpooling problem can easily be transformed into a vehicle routing problem, the constraint programming model will be very close to the one describe for the VRP in section 4.3.1.

The only two differences are the distances that must be zero for the first trips of a vehicle and the loads in the vehicles that must reflect the fact that passengers are picked up instead of delivering goods.

In other words the model from section 4.3.1 is reused with the following modifications:

- The distances from the starting points to the users are equal to zero: \( \delta_{i,j} = 0 \ \forall i \in F \).
- Constraints 5 and 6 are simplified since the constants \( r \) are not needed:
  
  5. \( \text{load}_i = \text{load}_{\text{pred}_i} + 1 \ \forall i \in V \setminus F \)

  6. \( \text{load}_i = \text{load}_{\text{succ}_i} - 1 \ \forall i \in V \setminus F \)

- If the problem simplification is used, the seventh constraint is also modified to use a global \( Q \) value instead of a different one for each vehicle:

  7. \( \text{load}_i \leq Q \ \forall i \in V \)

Apart from this, the constraints and objective function are the same as for a basic VRP. The complete carpooling model using constraint programming and based on the VRP implementation is available in the appendices at section 7.2. This model uses OscaR as constraint programming solver and is written in scala. Note that this model can’t be run alone, it must be used along with the search heuristic. Only the variables and constraints are attached, the rest of the implementation is available online.

As it was mentioned for the VRP, this problem is NP-hard. Two things were considered to accelerate the computation: a search heuristic adapted to this particular problem and using large neighborhood search to reduce the searched space. These two optimizations will be described now.

**Search heuristic**

The search heuristic that is used to solve the carpooling problem is a well-known heuristic used for vehicle routing problems. It uses the concept of regret, which is the difference between the cost of assigning a variable’s its best value and the cost of assigning it its second-best value.
The idea is to use a variable ordering heuristic that chooses the variable with the maximum regret. As value ordering we chose to assign the best value first. This heuristic is applied to bind the values of the successor variables from $succ$.

This search heuristic can be split into several steps:

1. Find the location $x$ with the highest regret value.

   The regret is computed for each $i$ such that $succ_i$ is not bound yet. For each of these regrets, the cost of assigning all values $j$ that are in the domain of $succ_i$ is computed. The cost of assigning the value $j$ to $succ_i$ is equal to the distance between $i$ and $j$.

   For each $i$ the cost of assigning the best and second-best values respectively are the followings:

   $$best(i) = \min_{j \in \text{dom}(succ_i)} \delta_i, j$$

   $$secondBest(i) = \min_{j \in \text{dom}(succ_i) \setminus best(i)} \delta_i, j$$

   These costs can be used to compute the regrets for each value of $i$ and the selection of variable $x$ is made by maximizing the regret:

   $$x = \arg \max_{i: succ_i \text{ is unbound}} (secondBest(i) - best(i))$$

2. Select the best value $v$ for this $succ_x$:

   $$v = \arg \min_{j \in \text{dom}(succ_i)} \delta_i, j$$

3. Branch the search tree with $succ_x = v$ on the left and $succ_x \neq v$ on the right.

### 4.3.3 Large neighborhood search

Large neighborhood search (LNS) can be used to accelerate the optimization problem. The principle of LNS is to restart the search from a previously found solution. LNS consist in keeping part of this previous solution while relaxing some variables and running the search again.

The process of relaxing a solution consist in fixing part of the decision variables to their value in the solution. The other variables are relaxed, their values from the previous solution are not fixed.

After relaxing a solution, the problem can be restarted. Restarting the search consists in running it on the relaxed solution with a limited number of failures. Once this limited number of failures is reached, the current best solution found is relaxed and the problem is restarted again. This is done until a fixed maximum number of restarts is reached.

Figure 4.11 shows the exploration of a search tree using LNS.
The key to have an efficient LNS is the choice of relaxation heuristic. The relaxation defines where the search will begin after the next restart. A good relaxation can direct the search toward a better solution.

Relaxations used for the carpool vehicle routing problem

Several types of relaxation strategies were implemented to use on the constraint programming carpooling model. As a general rule, the points that we will want to relax are the locations corresponding to users. The destinations and starting points distribution don’t matter. Also note that in the following when relaxing the values of $succ_i$ is mentionned, this means that the corresponding values of the $pred$ variables are also relaxed. Otherwise the relaxation would have no effect.

1. **Basic random points relaxation** :

   This relaxation heuristic consists in relaxing some of the $succ_i$ variables based on a completely random decision. Given a percentage of desired relaxed values $p$, for each $i \in C$, a random value $v$ is generated between 0 and 100 and $succ_i$ is relaxed if $v < p$.

   This heuristic is not very efficient because the randomly relaxed points have good chances to be far from each other which means that no new interesting path can be build. Even in the best case, the chances to find a better solution than the previous one are random.

2. **Random area relaxation** :

   The idea is to select a random element $e \in C$. Then the $n$ closest points to $e$ are relaxed.
The value of \( n \) depends on the percentage of points that have to be relaxed. This heuristic is already way more efficient than the previous one. Relaxing entire areas ensures that the relaxed points are close to each other and that interesting permutations are possible.

3. **Random path relaxation**: 
   The basic idea is similar to the previous relaxation heuristic. Except that for each point in the relaxed area, all of the locations serviced by the vehicle associated with it are also relaxed.

   This gives slightly better results than the previous heuristic.

4. **Closest paths relaxation**: 
   This is the first heuristic not involving a random choice. In this case, a distance was defined to compare two vehicles set of users. This distance is equal to the distance between the two closest users serviced by each of the vehicles. The two closest vehicles according to this distance have all of their users relaxed.

   This relaxation targets vehicles that picked up passengers close to each others. Those are interesting choices since their paths are likely to be similar, which means that switching passengers between the two vehicles might be interesting.

5. **Crossed paths relaxation**: 
   This is another heuristic that doesn’t involve a random choice. The points relaxed once again correspond to passengers that are picked up by two chosen vehicles. This time the choice of those vehicles is made by selecting those who have paths that cross each other.

   This is a very efficient heuristic. In most cases when two paths cross each other, permutations can be made to uncross them and these permutations will improve the solution.

   In the end the most effective heuristic was found to be the latest. However the crossed paths relaxations has two downsides. The first one is that it can’t be applied if there are no crossed paths. The second one is that it is not random in any way. This lack of randomization means that it could get stuck and repeat itself without finding a new best solution.

   To remedy to these two problems the crossed paths relaxation is not used alone. It is used until there are no more crossed paths, it is also made so it can’t try to uncross the same paths twice if the best solution didn’t change. Then if the maximum number of restart is not reached when there are no more paths to uncross, another relaxation heuristic involving random choices is applied.

   Note that these relaxations are either random, or compare paths two by two to choose which ones to relax. This means that there is no guarantee that each path will be locally optimized if the points were never selected by the relaxations. Because of this, after applying the usual restarts, another number of restarts equal to the number of vehicles in the problem is applied. Each of these restarts sees one of the vehicles’ passengers locations relaxed. This allows to make sure that each vehicle at least follows the optimal path to pick up its passengers.
4.3.4 Comparison with other solutions

The goal when using constraint based optimization on the carpooling problem is to have some guarantee that the solution found is good with regards to the objective function. Both of the two other solutions, filtering and sorting and clustering are such that the solution should be good because of the way groups are made. But none can prove that its solution is good.

Two things will be used to compare the different algorithms. The first one is the computation time of each. The second one is the total driving duration for the solutions obtained. Having a lower total driving duration means that in average the customers spend less time on the route.

All the tests were done on previously generated data sets of various sizes. This means that when two separated tests are run for the same number of users, the data sets used with the locations of those users are the same. These data sets are made of semi-randomly generated geographical locations in Belgium. To make the simulation more realistic, the distribution of the locations is not entirely random. Users locations are generated depending on the distribution of the population. To do this a list of the population in the major cities\(^2\) is used.

About the filtering and sorting method, the two proposed distances give similar results both in terms of computation time and durations obtained. The starting point distance being generally slightly better, it is the only one that will be used to compare with the other solutions.

The clustering method that is used is the modified k-means, it is better than the hierarchical solution and faster to compute. Since the random location of the initial centroids can change the results for the k-means algorithm, this algorithm was ran 20 times on each instance and the average results were kept. This was not necessary for the two other methods since filtering and sorting is completely deterministic and the solutions from the routing method don’t vary much either.

The durations used for these comparisons are all flying distances divided by a speed of 100km/h. More realistic durations from the online maps services might have been used but the results would not change and since those tests were ran multiple times the access limits and slow access to those services were impractical. Moreover, the modified k-means clustering works with flying distances, using the same distances for the other solutions is a more interesting way to compare them.

The first set of tests were made with only one passenger by car. This is the only type of problem that can be directly solved by the filtering and sorting method.

\(^2\)The list of the populations in the largest belgian cities that was used comes from \textit{http://www.world-gazetteer.com/wg.php?x=E\&men=geis\&lng=en\&des=wg\&srt=npan\&col=abcdefghinoq\&msz=1500\&geo=-30}
The results are presented in figure 4.12 for the durations. As expected the CVRP method gives the best results. It is not easy to say which one of the two other methods is the second best. The clustering method takes the second place in 3 of the 5 cases, it seems to consistently do better for higher numbers of users.

<table>
<thead>
<tr>
<th>number of users</th>
<th>CVRP</th>
<th>filtering and sorting</th>
<th>clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>15851</td>
<td>16882</td>
<td>17555</td>
</tr>
<tr>
<td>20</td>
<td>28273</td>
<td>32418</td>
<td>30846</td>
</tr>
<tr>
<td>50</td>
<td>62425</td>
<td>72084</td>
<td>66541</td>
</tr>
<tr>
<td>100</td>
<td>118113</td>
<td>129774</td>
<td>121052</td>
</tr>
<tr>
<td>200</td>
<td>230037</td>
<td>248109</td>
<td>235585</td>
</tr>
</tbody>
</table>

Figure 4.12: Total duration results of tests, maximum 1 passenger. Travel times in seconds. Lower is better.

Figure 4.13 gives the time needed for each algorithm to find a solution. Once again the results are as expected. The CVRP method is way slower than the other two, especially for high number of users. The k-means clustering is very fast. It is interesting since one of its disadvantages is the randomness of the solution depending on the initial centroids. The fact that it finds a solution so quickly allows to run it multiple times and keep the best solution to solve this randomness problem.

Some other tests were executed with a different vehicle capacity. Since the filtering and sorting method only works with a unique passenger it will not be used this time. The results are showed in figure 4.14. Only the total driving durations is presented, the computation times are nearly the same as for the problem with one passenger.

<table>
<thead>
<tr>
<th>number of users</th>
<th>CVRP</th>
<th>clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>373</td>
<td>9</td>
</tr>
<tr>
<td>20</td>
<td>1669</td>
<td>22</td>
</tr>
<tr>
<td>50</td>
<td>1034</td>
<td>22</td>
</tr>
<tr>
<td>100</td>
<td>8463</td>
<td>91</td>
</tr>
<tr>
<td>200</td>
<td>149573</td>
<td>578</td>
</tr>
</tbody>
</table>

Figure 4.13: Time (in s) needed to computer the solution. Lower is better.

Figure 4.14 gives the time needed for each algorithm to find a solution. Once again the results are as expected. The CVRP method is way slower than the other two, especially for high number of users. The k-means clustering is very fast. It is interesting since one of its disadvantages is the randomness of the solution depending on the initial centroids. The fact that it finds a solution so quickly allows to run it multiple times and keep the best solution to solve this randomness problem.

Some other tests were executed with a different vehicle capacity. Since the filtering and sorting method only works with a unique passenger it will not be used this time. The results are showed in figure 4.14. Only the total driving durations is presented, the computation times are nearly the same as for the problem with one passenger.

<table>
<thead>
<tr>
<th>number of users</th>
<th>CVRP</th>
<th>clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12999</td>
<td>14052</td>
</tr>
<tr>
<td>20</td>
<td>19745</td>
<td>22111</td>
</tr>
<tr>
<td>50</td>
<td>41918</td>
<td>45657</td>
</tr>
<tr>
<td>100</td>
<td>74166</td>
<td>77918</td>
</tr>
<tr>
<td>200</td>
<td>146645</td>
<td>144737</td>
</tr>
</tbody>
</table>

Figure 4.14: Total duration results of tests, up to 3 passengers. Travel times in seconds. Lower is better.

The results with up to 3 passengers are interesting. First of all, a quick comparison with the results for the model with only one passenger from figure 4.12 confirms that the total driven duration goes down when drivers are able to pick up more passengers. The second thing that can be noticed is that while the results are comparable to the previous ones for
numbers of users inferior to 200, the k-means solution for 200 users is better than the CVRP one. This can be explained by the fact that the number of restarts and maximum number of failures didn’t change for the CVRP. Increasing these parameters would allow the large neighborhood search to explore more of the search tree. Running the CVRP algorithm with up to 100 restarts and 2000 maximum failures by restarts confirms this. The solution found has a total driven duration of 139880s.

This reminds us that the CVRP method only finds an approximated best solution if the computation time is limited. With enough time the model can be set to find the absolute best solution.

4.3.5 Extended CVRP model with additional constraints

Giving better results in terms of optimization is not the only strength of the CVRP method. Another one of the advantages of using constraint programming is that the model is very flexible to adding other parameters. It means that it can be tailored following to the needs of customers.

This section will describe a few ideas of what can be added to the model. The extended model was not included in the previous performances comparison because it doesn’t find solutions to the same problem as the other methods. It would make no sense to compare the total driven duration found for a basic model with one found for a model that includes time windows.

The elements that are added to the extended CVRP model are the following ones:

- Replace the simplified global car capacity and use the capacities of the cars of the drivers.
  To do this a new array of constants is added to the model: \( capacity_i \in [1, 15] \), defined \( \forall i \in M \). These variable store the capacity of the different vehicles.

  The seventh constraint of the CVRP model must be modified to take this into account, another option is to add the new constraint and set the value of \( Q \) to \( Q = \max_{i \in M} capacity_i \). The new constraint is:

\[
7. \quad load_i \leq capacity_{vehicle_i} \quad \forall i \in V
\]

- Add time windows that allow users to choose when they can be picked up from their home. The choices of availability time windows from the users are stored in two arrays of constants: \( twStart_i \) and \( twEnd_i \) are defined \( \forall i \in C \) and give respectively the earliest time at which user \( i \) can leave his home and the latest. Another constant \( tArrival_i \) stores the latest time at which users can arrive to the destination.

  For this a new decision variable is needed:

  \(- departureTime_i \in V \) with \( i \in V \) represents the time at which \( vehicle_i \) leaves \( i \).

  The time needed by the passenger to get in the car is assumed to be close to zero.

  New sets of constraints are also needed:

\[
8. \quad departureTime_i < twEnd_i \quad \forall i \in C
\]

  Each user leaves his home before the end of his available time window.
9. \(\text{departureTime}_i \geq \text{twStart}_i\) \(\forall i \in C\)
   Each user leaves his home after the start of his available time window.

10. \(\text{departureTime}_{\text{succ}_i} \geq \text{departureTime}_i + \delta_{i,\text{succ}_i}\) \(\forall i \in C\)
    Departure times must take the travel time between locations into account.

11. \(\text{departureTime}_i < t\text{Arrival}\) \(\forall i \in L\)
    Each user gets to work in time.

- Add a maximum detour time that users can specify, this is for the car drivers. For example if user \(u\) is the driver and specified a maximum detour duration of 20min, the algorithm will make sure he doesn’t spend more than 20 more minutes on the road than if he was riding alone.

These maximum detour durations are stored in constants: \(\text{maxDetour}_i\) corresponding to the value chosen by user \(i\).

New variables are needed to keep track of the detour durations of each vehicle:

- \(\text{detour}_i\) with \(i \in V\) represents the detour made by \(\text{vehicle}_i\) when it gets to \(i\).

The constraints needed are the following ones:

12. \(\text{detour}_i = 0\) \(\forall i \in F\)
    Fixes the value of the detour to zero for starting points.

13. \(\text{detour}_i \leq \text{maxDetour}_{\text{vehicle}_i}\) \(\forall i \in C\)
    Makes sure that the maximum detour duration is not exceeded.

14. \(\text{detour}_{\text{pred}_i} + \delta_{i,\text{succ}_i} = \text{detour}_{\text{succ}_i}\) \(\forall i \in C\)
    Keeps track of the detours values.

\[\text{Figure 4.15: Extended model with 10 users. The labels format is the following :}\]
\[\text{UserId} - \text{peopleInCar/CarCapacity(UserCarCapacity)[TimeWindow]}\]

These few examples illustrate the possibility to add features to the model by adding new constraints. However adding new decision variables increases the size of the search space so it
slows down the computations.

### 4.4 Summary of the solutions

Three methods were proposed to solve the carpooling problem:

1. **Filtering and sorting (section 4.1)**
   
   This method is clearly more adapted to websites, the methods works well with a low number of passengers and is easy to apply. However it was found to be less efficient than the other two with regards to finding an optimized solution for a big group of users.

2. **Clustering (section 4.2)**
   
   This method is very efficient in terms of computation time. It also gives a pretty good solution. However it can’t go farther than that, it lacks a proper optimization process and can’t easily be improved by adding features.

3. **Carpool Vehicle Routing Problem (section 4.3)**
   
   This is the slowest solution in terms of computation time. But it gives the best results and features can easily be added.

In conclusion, the choice of the method to use depends on the application purpose. Filtering and sorting is more suited to a mobile or web based ”find a ride” type of application. Clustering can be used to compute solutions quickly for sets of users. The vehicle routing based solution is the one to be used if time is not a problem.

Another option is to combine the methods. For exemple the clustering method and the vehicule routing method can easily be combined. The clustering method can be used to split the users set into smaller sets. Then the CVRP can be applied separately to each of these subsets. This would give a better solution than a clustering method alone would have found. It would also run faster since the work of the vehicle routing method is simplified by the preprocessing from the clustering method.
Chapter 5

Conclusions

Several ways to solve the carpooling problem described in the section 2.3 were found. As explained in section 4.4 these can all be used depending on what the user’s priorities are.

However if the solution doesn’t have to be computed instantly, the vehicle routing approach has no major disadvantage. Combined with some preprocessing if the number of users is too high, it gives the best results and can be customized at will. It is the best solution to the problem definition chosen in section 4.4 because it gives the best optimization of the objective function. However this problem definition assumed that the only important thing is the distance driven. This would be true if the only reason for carpooling was environmental concerns. Since it is not the case, solving this problem doesn’t mean that the vehicle routing solution is the absolute best.

Different approaches to the problem, such as using a different objective function, might be interesting to improve the solutions that were found here.

Since the needs can vary from one customer to another, there is not really a way to determine which of the solutions would be the best in general. The different methods described in this work can all be used as basis for building customized solutions.
Chapter 6

Bibliography


Chapter 7

Appendices

7.1 Source code

The complete source code produced for this thesis is available online. This includes the source code of the different solutions as well as the source code used to compare web services.

The adress of the repository is: https://bitbucket.org/cmulders/carpoolingthesis

The source code is organized in the following packages:

- **maps** contains all the implementation linked to accessing map services and comparing Google Maps with Mapquest.
- **filtering** contains the implementation of the filtering and sorting method from section 4.1.
- **clustering** contains the implementation of the clustering methods from section 4.2.
- **routing** contains the implementation of the vehicule routing based method from section 4.3.
- **benchmarks** contains classes used to apply the different solutions to previously generated data sets.
- **utils** contains the classes used to store points and journeys.
- **tests** contains some unit tests, these don’t cover the whole implementation but only the web services access classes.

7.2 Carpooling constraint programming model

```java
package routing

import oscar.cp.modeling._
import oscar.cp.core._
import oscar.util._
import utils.Point
```
class CarpoolRoutingModel(cl: List[Point], dest: Point, maxPassengers: Int) {

  /*
   * constants : index ranges for elements of the problem
   * ******************************************
   */
  // vehicles
  val nVehicules = cl.length
  val vehicules = 0 until nVehicules

  // people to pick up
  val nClients = cl.length
  val clients = 0 until nClients

  // for each vehicle one duplicated destinations as departures and
  // arrival points : arrivals
  val destinations = nClients until nClients + nVehicules

  // duplicated destinations as departures and arrival points : departures
  val starts = nClients + nVehicules until nClients + 2 * nVehicules

  // all locations used including destinations, starts and clients
  val nLocations = nClients + 2 * nVehicules
  val locations = 0 until nLocations

  /*
   * distances between locations
   * ******************************************
   */
  val durations = Array.ofDim[Int](nLocations, nLocations)

  // dist from departures is 0 dist between arrival and departure is 0 too
  for (i <- destinations)
    for (j <- starts) {
      durations(i)(j) = 0
      durations(j)(i) = 0
    }

  // dist to go from departure to first client (conductor) is 0
  for (i <- starts)
    for (j <- clients) {  
      durations(i)(j) = 0
    }

  // distances from client to client
  for (i <- clients)
    for (j <- clients) {
      if (i != j) durations(i)(j) = cl(i).duration(cl(j))
      else 0
    }
}
// distances from clients to arrivals
for (i <- clients)
    for (j <- destinations) {
        durations(i)(j) = cl(i).duration(dest)
    }

val cp = new CPSolver()

/*
 * variables
 * ******************************************
*/
// Successors
val succ = Array.fill(nLocations)(CPVarInt(cp, locations))
// Predecessors
val pred = Array.fill(nLocations)(CPVarInt(cp, locations))
// loads of vehicles at each point
val load = Array.fill(nLocations)(CPVarInt(cp, 0 to nClients))
// Vehicule used
val vehicule = Array.fill(nLocations)(CPVarInt(cp, vehicules))
// Total distance
val totDuration = CPVarInt(cp, 0 to durations.flatten.sum.toInt)

// successors of starting points are always either a destination (unused vehicule) or the associated client
for (i <- starts) {
    succ(i) = CPVarInt(cp, (i - 2 * nClients) +: destinations)
}

/*
 * constraints
 * ******************************************
*/
def carpoolConstraints() = {
    // departures predecessors are corresponding arrivals
    for (i <- starts) {
        cp.add(pred(i) == i - nVehicules)
    }

    // arrival successors are corresponding departures
    for (i <- destinations) {
        cp.add(succ(i) == i + nVehicules)
    }

    // Channeling between predecessors and successors
    for (i <- locations) {
        cp.add(pred(succ(i)) == i)
        cp.add(succ(pred(i)) == i)
    }

    // Consistency of the circuit with Strong filtering
}
cp.add(circuit(succ), Strong)
cp.add(circuit(pred), Strong)

// Total distance
cp.add(sum(locations)(i => durations(i)(succ(i))) == totDuration)

// start capacities
for (i <- starts ++ destinations) {
  cp.add(load(i) == 0)
}

// check capacities of the vehicles
for (i <- clients) {
  cp.add(load(i) <= maxPassengers)
  cp.add(load(pred(i)) + 1 == load(i))
}

// fix departure and arrival routes, arrival determines the vehicle used
for (i <- destinations) {
  cp.add(vehicule(i) == vehicule(pred(i))) // i-nClients-nVehicles)
}
for (i <- starts) {
  cp.add(vehicule(i) == i - 2 * nClients)
}

// keep track of the vehicle used for every client by matching route with destination number
for (i <- clients) {
  cp.add(vehicule(i) == vehicule(pred(i)))
}
}