Declarative Programming project: Taxi Company

Jeroen Heymans

January 13, 2012

Contents

1 Introduction

This is the report for the project for the course Declarative Programming, a master-course teached at the Vrije Universiteit Brussel. In this report we describe the design and the functionality that has been implemented.

The general idea of the project was to build a program that would calculate the routes for the taxis of the taxi company El Cabinero. This company works in a given city that is described in a graph. This graph was given to us, alongside with a list of 500 customers who wanted to be transported.

The program can be run on SWI-prolog. What you need to do, is load the main.pl file and execute the main function like this:

```
?- reconsult(main).
true.
?- main.
```

The code for this project can be found in the attached ZIP-archive or on https://github.com/jeroenheymans/PrologProjectCabDriver

2 The implemented program

2.1 Path calculation

Since we needed to implement the discovery of routes, we obviously needed an algorithm that could calculate a short path. Multiple algorithms exist that can calculate the shortest (or almost shortest) path in a graph. We have firstly implemented a version of Dijkstra's wellknown algorithm. With this, we could get a good indication of what the shortest path would be. Unfortunately, Dijkstra is quite heavy in the use of memory. This slowed the program down significantly. As a benchmark, we took the calculation time that was necessary to calculate the path between node # 0 and node # 2499, two of the utmost nodes in the graph.

With our implementation of Dijkstra, it took half an hour to calculate this path. In order to get the execution time down, we choose to add a heuristic to the algorithm. The problem with the naïve Dijkstra implementation was that we could have situations where the shortest path goes from left to right but that the algorithm would first see if there was any shortest path to the left. By adding a heuristic, we could guide the algorithm more to the correct direction.

We found a heuristic that decreased the time of calculation to a maximum of 2 seconds. With this heuristic, our implementation of a shortest path search algorithm could be used in the final program. The heuristic only has one flaw: it does not find the absolute shortest path. For example, if the absolute shortest path has a distance of 100, our algorithm will vary between a result of 100 and 105. Though this is a small flaw, we continued to search for a better heuristic but we had to conclude that this was the best one we could find without losing to much speed.

2.2 Datastructures

Our own implemented datastructures are:

- customerAvailable/3
- taxiJob/4

We will explain them more in detail what their purpose is.

2.2.1 Keeping track of the customers with customerAvailable/3

As the program runs, we must keep track of the customers that are already picked up. For this, we created the *customerAvailable/3* dynamic fact. When we start the program, we give each known customer a customerAvailable fact by asserting them. With this, we could easily retract the customers that we already picked up without losing all the original data about the customers. It also enabled us to add extra information that could be preprocessed like the path from the dropofflocation of a customer to the startpoint of the taxi's.

2.2.2 The taxi jobs with taxiJob/4

The functor taxiJob/4 is asserted whenever the loop for a taxi is completed. It holds the ID of the taxi, the customers that will be transported by the taxi, the total path that the taxi must follow and the return time of the taxi to his original startingpoint.

2.3 The main structure

The overall program is a sequence of the following parts: preprocess, loop over taxis, output. We will go deeper in this structure, so we can explain more in detail what we have implemented.

2.3.1 Preprocess

In order to increase performance of the program, we try at first to decrease the size of the dataset on which we operate. In the dataset, we have customers and their preferred range of time of picking up. There are however customers where the range of time for picking up is not good enough. For example, a customer that wants to be picked up between 1200 and 1300 can not be transported if the distance to his destination is 250. It is impossible for a taxi to pick the customer up on time and drop him off at such a time that the taxi can return home before midnight (1440).

A similar problem occurs when a taxi can pick up and drop off a customer before midnight but can not return home before midnight. In these cases, it is unnecessary to take such customers in account as it is already known that is impossible to transport them within the time limits. We therefore preprocess all this information in order to decrease the search sapce of customers to transport.

2.3.2 Loop over taxis

In order to fill the taxis, we take a list of all the taxis and loop over them one by one. When taking a taxi, we immediately assign one customer to it. After this, we try to find out if we can add more customers to the same taxi. Each time, we look for the best possible customer to add. This means a customer that we can pick up within his timerange or where we only have to wait some time so we can pick him up within the timerange. Every customer that we consider generates a lot of calculating to see if we could stay within the timeframe of a complete day. If we can find a customer that can be picked up within his timerange but where we can't stay within the timeframe of one day, we have to discard this customer and try again with other customers.

We continue to fill the taxi until there are 4 customers assigned to the taxi or if there is no valid customer left to pick up. When we get to one of these two endstates, we save the total calculated path that the taxi must drive to pick up and drop off his customers. This path is saved within the functor taxiJob.

2.3.3 Output

When all the taxis are filled or when all the customers are assigned to a taxi, we start outputting the results. What we do is loop over all the generated taxiJobs and output the result of every taxiJob. We loop over the path of a taxiJob and every time we can pick up or drop off a customer, we show this. This generates an output like this:

```
Route for taxi 0 (arrival: 1343):
Picking up customer 462
Picking up customer 6
Picking up customer 394
Picking up customer 81
Dropping off customer 81
Dropping off customer 394
Dropping off customer 462
Dropping off customer 6
```

2.4 The final result

The program almost generates a result where all the customers are transported from the original dataset. With preprocessing, only 262 of the original 500 customers are considered to be transportable within the timelimits. From this 262, the program can make sure that a total of 255 customers are transported correctly.