Sharing cars to get to work: A Local Search approach

Artificial Intelligence

Carlos Bergillos, Roger Vilaseca, Adrià Cabeza

March 30, 2019

Contents

1 Introduction

The objective of this assignment is to learn different problem-solving techniques based on Local Search using the AIMA library in Java. In particular, the Hill Climbing and Simulated Annealing.

After implementing states and operators for the algorithms, we have to do the second objective, which is comparing the different results obtained with both algorithms. Making different experiments and extracting results.

2 Description of the problem

For this assignment we are assuming a car sharing system where all the users are sharing the car. We have N people and M drivers. which is a subset of N.

Our city is a 10×10 km square and each street is disposed every 100 m (horizontally and vertically). This disposition creates a grid of 100×100 *blocs* with each bloc of 100×100 m.

TODO: PUT GRID IMAGE

The calculation of the distance between two points of the city will be made using the Manhattan function:

$$d(i,j) =\mid i_x - j_x \mid + \mid i_y - j_y \mid$$

where i_x and i_y are the coordinates x and y of the i point in the grid.

Each user of the service will leave home at 7am and has to arrive work before 8am and the highest speed in the city is 30 Km/h. With the previous restrictions, supposing that the speed always is 30 Km/h and supposing that picking up and dropping off people don't have penalization time, we can do the following reasoning.

$$dist = speed * time = 30\ Km/h * 1\ h = 30Km$$

So each driver at most can drive 30 Km.

Also, in each car can be at most three people at the same time (including the driver.

We will have two criteria in order to evaluate the quality of the solution:

- Minimize the distance traveled for each driver.
- Minimize the distance traveled for each driver and minimize the number of drivers.

3 Representation of a problem state

We need a way to represent a state of the problem that is small in memory, but at the same time easy and quick to work with. We want to avoid storing unuseful or redundant information, unless we clearly think that that information

will help us reduce the computation time. Also, we want to be able to represent all the possible states that could exist.

Our states contain the following information:

- Which users act as drivers and are providing their cars.
- For each driver, which users (passengers) the driver needs to pick up and drop off, if any.
- The order in which a driver needs to pick up and drop off its assigned passengers.
- The total distance each car will be travelling for its route.

All users (both drivers and non drivers) have a unique identifying number (id), taken from the position they occupy in the users list. So in our state representation we will be using these identifiers to refer to users.

We consider that the order in which a car picks up and drops its passengers is important. For example, if driver A is assigned passengers B and C, there are four possible routes after leaving its home, and before arriving at work:

- Pick up B, pick up C, drop off B, drop off C.
- Pick up B, drop off B, pick up C, drop off C.
- Pick up C, pick up B, drop off C, drop off B.
- Pick up C, drop off C, pick up B, drop off B.

These four options will result in different routes with potentially different total distances.

For this reason, we decided to use an ordered list to describe which passengers a driver needs to take care of, and the specific route they will use in the process.

Specifically, for each driver, we use a Java ArrayList listing all the sequential pick ups and drop offs of users that have been assigned, describing unambiguosuly a route that the driver needs to follow.

Continuing with the previous example, if driver A has id 1, and passengers B and C have ids 2 and 3 respectively, the first route proposed before will be stored like this:

$$[1, 2, 3, 2, 3, 1]$$

In a different state, maybe the second proposed route is used, in that case the list would look like this:

$$[1, 2, 2, 3, 3, 1]$$

Note how each id in the list always needs to appear twice, implicitly the first time it appears indicates that that user is being picked up, and the second one indicates that the user is being droped off.

Also note how the driver of the car is always the first and last element of the list, to represent the fact that the route must start at the driver's home, and end at the driver's workplace.

Because a state can describe the presence of many different cars (each one with its own driver, passengers, and route) we use an ArrayList to contain all the cars lists described before. Thus, we end up with the following data structure:

```
ArrayList<ArrayList<Integer>> assignments;
```

Apart from this, we also decided to use an extra data structure that contains the total route distances for each car:

```
ArrayList<Integer> distances;
```

Where the distance for the route in `assignments[i]` is stotred in `distances[i]`.
Although not strictly necessary (because this could be computed at runtime from other existing data), this `distances` variable will help us calculate the heuristic value of the state quicker. We just make sure to update these values any time `assignments` is changed.

4 Generating an initial solution

We have to make a representation fo the initial state, which at the same time is a solution state. We have implemented four different ways to generate our initial state:

1.
2.
3. Taking all the people that are not drivers inside the first car's queue.
4. Taking all the people that are not drivers inside the first car's queue.

5 Analysis of our operators

Once we have defined the structure we will work with, we have to decide which operators will modify our structure to move from one state to another. To do this, we must take into account several factors, so that when executing our algorithm, the best possible solution is found in a fairly reasonable execution time.
Our operators indicate all the possible paths that can be taken given any state. Then we will use all these possibilities with the objective that this becomes a state with some favorable characteristics. This is called a branch factor and it changes the way it is applied depending on which algorithm we are using. In the **Hill Climbing**, we generate all the successors and the heuristic decides if it is good enough to stay with him, in the **Simulated**

Annealing we generate a successor status in a random way and the heuristic decides if it is good enough to stay with him.

It is really important to cover all the space of solutions with our operators because if we are not doing it, there may exist solutions that will be lost, which could prevent us from reaching an optimal solution. Also we be cautious about creating repeated solutions because our execution time would be affected.
At the beginning of everything we made a brainstorming session with all the operators we could think of, which we believed that could be useful and serve for something: swap the order of the people inside a car, swap outside people between cars, deleting cars, moving a person into another car, etc...
Finally we decided to implement these 3 operators, which we think they would reach to all the possible solutions:

1. **Move**: this operator lets us move any person that is not a driver from the car i to another car j, in a pickup place k and a drop off place l.
2. **Swap inside**: this operator lets us swap the order of the people inside a car i.
3. **Delete car**: this operator lets us delete a car whenever the car is only occuped by the driver. When the deletion is made the driver is inserted into another car.

6 Generating successor states

The way we generate all the successor states really varies depending on the algorithm:

- **Hill Climbing**: we create a list with all the possible states that can be found from the given one. To do it, we create all the possible states that can be created using our operators, previously explained in section above. Then Hill Climbing will use our Heuristic Function to choose the best successor.
- **Simulated Annealing**: we return a random state that is a succesor of the given one. To do it, we choose a random operator with random values. Then we use it in order to change the initial state.

7 Representation of the Goal State

8 Heuristics Function

Once the representation of the solution state has been defined, the generation of the initial solution state and the operators on which we are going to work, we proceed into analysing the heuristic function.

In order to solve the two solution criteria given in the statement, we must perform two different heuristics, because the final result that must be returned has different priorities. The heuristic function that will solve the first criterion will be called Heuristic Function 1, and the one that resolves the second, Heuristic Function 2.

8.1 Heuristic Function 1

The criterion that this function must follow is quite simple, the objective is to minimize the sum of all the distances that each car has to do.
To follow this criterion, we...

8.2 Heuristic Function 2

For the second heuristic we have added another criterion in order to minimize also the number of cars that are driving.

9 Experiment 1

In this experiment we will decide which is the best operator of the different ones we have created using the *Heuristic Function 1*.

This experiment is going to be made using **200 people** (N), and **100 drivers** (M) and only the **Hill Climbing algorithm**. In order to perform it, firstly, we have generated an initial solution. From all the possible initial solution generators that we have implemented, we have chosen the first one, which BLABLABLA since we believe that is the one that gives us the best results.
As we mentioned in the previous section *Analysis of our operators*, we have three different operators. In order to try out which set of operators is the best one, we have automatized a process where all different operators are applied to the scenery using different seeds to see which one gives us better results and better time.
Our **hypothesis** is that a combination of all of them will gives us the best results.

10 Experiment 2

We did this experiment in order to know which was the most suitable way to generate our initial solution. In this experiment we will use the best set of operators available found in the previous experiment.
We will use the same scenery that was used in the first experiment ($N = 200$ and $M = 100$). In order to try all the different initial solution generators we have created a process where the same scenary is applied to different initial solution methods to see which one gives us better results and better time.
Our **hypothesis** is that...

11 Experiment 3

A really important part about this project is to be able to determine the parameters that are going to be used for the **Simulated Annealing algorithm**, so that its results are the best possible, that is, the minimum number of cars driving the shortest amount of distance and everybody getting to their workplace in a reasonable execution time.

The algorithm works with 4 parameters:

1. **Steps**: The total number of iterations that the algorithm will perform.
2. **Number of iterations for each temperature change**: Whenever there is a temperature change, a constant number of iterations will be made in which the probability of choosing a worse successor is kept. Each time this number of iterations is executed, the probability of choosing a worse state decreases.
3. **k**: parameter of the state acceptance function that affects the probability of choosing a worse succesor state. We know that the higher the value of the parameter, the longer it takes for the probability of staying with a worse successor state to decrease.
4. λ: Another parameter of the state acceptance function that affects the probability of choosing a worse successor. The higher the value of λ, the less it will take to decrease the probability of accepting a worse successor state.

12 Experiment 4

13 Experiment 5

14 Experiment 6

15 Experiment 7

16 Especial Experiment

At this point we have degined the best initial state, its operators, the heuristic algorithm and Hill Climbing and Simulated Annealing with the most appropiate parameters, so we can proceed to compare algorithms.

16.1 Hill Climbing

The experiment was made configuring our environment with 200 people, 100 drivers and seed 1234.

We would like to add that these results are a valid solution because no car drives more than 30 km and there is not any moment where a car carries more than 2 people.

Time	**Nodes expanded**	**Cars**	**Distance**
6054 ms	17	92	1834.3 km

Table 1: Results of our experiments with Hill Climbing

16.2 Simulated annealing

The experiment was made configuring our environment with 200 people, 100 drivers and seed 1234.

	Time	**Cars**	**Distance**	**Solution**
#1	2596 ms	62	1671.3 km	Yes
#2	1218 ms	64	1678.2 km	Yes
#3	1044 ms	67	1753.3 km	Yes
#4	1088 ms	65	1734.8 km	Yes
#5	1058 ms	63	1731.1 km	No
#6	1131 ms	65	1685.2 km	Yes
#7	1029 ms	65	1720.5 km	Yes
#8	1067 ms	66	1777.1 km	Yes
#9	1088 ms	64	1657.6 km	Yes
#10	2596 ms	65	1688.5 km	Yes

Table 2: Results of our experiments with Simulated Annealing

The mean time of our executions using Simulated Annealing is 1709.76 ms.