Project Artificial Intelligence 2018-2019

Jarrit Boons, Cédric Gullentops, Laurens Van de Walle

April 2019

1 Introduction

Some problems have complicated solutions where the calculations for the most optimized solution can take a very long time. In this project we will be creating an application for such a problem. In this problem people can request a rental car. It is our job to place these cars in a city that is divided in multiple zones to optimize the profit, and thus lower the cost. We have to account for the amount of available cars, their type, the location and timing of the requests. For example, a person can make a request for a vehicle at a certain time for which we have to provide that car in that zone or neighbouring zone. That same car can then be used in the same location for another request at a later time.

When we try to create a 'normal' solution for such a problem we can quickly notice the complexity that makes the calculation time very big. To find a suitable solution in a more practical way we will use a heuristic method, namely a meta-heuristic method called simulated annealing.

The goal of this project is to create an algorithm that can find a suitable and correct solution in a short timespan (5 minutes) in which we have to minimize the cost. To make this happen we have to make sure we spend enough time planning about optimization, the algorithm and efficient functions to be able to run as many iterations as possible. We will also have to find the most optimal settings in which to run this algorithm.

In the following sections we will talk about the approach we used, the used methods, the results and finally the conclusions we make with recommended improvements.

2 Approach

To start we had to make a choice for what programming language we would use. Since the search algorithm had to be as optimal as possible and we could divide data structures into classes we found that Java or C would be the most optimal choices. We chose Java since we were more familiar with it. After this we started to create multiple **Classes** for each element. For example, we made a **Controller**, **Head**, **Car**, **Request**, **Zone** and **Solution** class. The class **Controller** contains the main function which creates a **Head** to initialize everything. The **Head** class reads in the data, creates objects, creates a crossover matrix, then creates threads and finally starts all processes for a certain duration. The other classes contain basic data like id's and functions to change the data. The solutions are saved in an oject of the **Solution** class. These contain lists of other basic data classes and functions to generate a initial solution, generate mutations, calculate the cost and write results to a file. This initial solution is generated using a first fit technique in which every request is assigned the first possible car. We made this choice to avoid spending too much time calculating a initial solution when this time is better spend on doing more mutations and using the meta-heuristic method. These last two will have a bigger impact on the final result than the initial solution.

We chose for simulated annealing as our stochastic meta-heuristic method since it is able to efficiently search through a big search space and find a good solution. This method also allows us to set certain parameters to optimize the results even more. We start the temperature parameter on a big value and let it cool in geometric fashion to a lower value. (This temperature and cooling are a reference from the theorie.) This starting value has to be defined empirically. We were able to find a logical start and end value by doing the maths and came to the conclussion that our starting value should be around a value of 1000 and our ending value should be around a vaulue of 20. The performance of these setting will be evaluated in a later segment.

Furthermore we chose for simulated annealing to make sure that our algorithm doesn't get stuck on local maxima. This is because there is a small chance that based on the temperature a worse solution is accepted as a new solution. This forces the algorithm to a new neighbourhood in the search space. When the temperature gets lower the chance of a worse soltution being accepted is lowered aswell. This makes the algorithm search for better soltions in its current neighbourhood instead. Temporary solutions are always being compared to the best found soltion so far, if it is better it takes its place.

Because there are multiple settings that are codependent we wanted to make this codependance as small as possible. To do this we implemented a function that makes sure that the cooling value, the starting temperature and the end temperature can be constants. After some tests we decided that a starting temperature of 800 and ending temperature of 10 are good values. From the theorie we know that a cooling value should lie between 0.5 and 0.99. We chose a value of 0.99 because this makes sure that the colling value is lowered gradually. Our functin only changes the amount of iterations iterated per temperature by calculating the amount of mutations peformed in the passed time so far and predicting the amount of possible mutations left. This makes sure that our algorithm will always pass through all temperature values between 800 and 10. However, this function has one side-effect. Because our algorithm now is dependent on time and the amount of provided calculation time by the processor it is no longer pseudo-random. This is why we implemented it after all other tests with seed values were completed.

There are 2 mutations that can be done. The first is a mutation based on a number of cars. This amount is another parameteren will also be evaluated in a later segment. This mutation makes the biggest amount of changes and is mainly used to make bigger steps in the search space. The second mutation is based on a amount of requests. This mutation makes less changes and is mainly used to further optimize a temporary solution. We chose to work in steps when choosing which mutation to use. One such step is the change of 1 temperature value and takes a number of iterations. During these iterations we will start by mutating a lot of cars and lower this value in exhange for more request mutations.

Furthermore we chose for the ability to use multiple threads in which every threads is of the **AnnealLoop** class. Using multiple threads makes sure that we can search with multiple solution in the search space which gives even more

variation in the algorithm. This increases the chance of finding a better solution. We chose to compare the solution of each thread solely in the end since we can't be sure that comparing solutions earlier will result in a better final solution.

Lastly we made a class **PlotData** to plot a graph and a class **Clock** to manage timing.

3 Testing and results

As stated earlier, there are number of parameters which can be tweaked to change te outcome of the program. We determined the optimal value of most of these in an experimental way. We found that for the step size of the mutations we got the best results for a value of three. And that a start and end temperature of respectively 800 and 10 score the highest (i.e. lowest cost). Note, however, that these values not necesarrely render the most optimal score possible, since there always is a random factor which we can not always take into account. The scores we obtain with these values are overall quite good and will not be researched more thoroughly.

We did not create a table for the test values we got while testing the above mentioned parameters since they were determined in a more trial-and-error fashion, rather than in a predetermined way. And with the randomness and the small difference in result in mind we did not consider it a very valuable time investment.

While testing the program we altered the values of three of the parameters we consideren most impactful: the random seed, the time the program was allowed to run and the number of created threads. After a few tests where we increased the number of threads we found that a higher number of threads consistently resulted in a lower cost. To limit the duration of the testing we decided to always work with 4 threads. The results of these tests can be seen in the table below (Figure 1).

Seed:	**Tijd:**	**100_5_14_25**	**100_5_19_25**	**210_5_33_25**	**210_5_44_25**	**360_5_71_25**
Initial		14800	13005	24915	13660	25225
1	1min	8905	6085	10950	5770	11470
	5min	8845	5935	10955	5055	9145
	10min	8890	5760	10615	4755	8860
10	1min	8755	5900	10595	5000	9655
	5min	8690	5745	10540	5005	8690
	10min	8825	5715	10495	4380	8270
100	1min	8940	5940	11070	5630	10810
	5min	8785	5685	10545	4785	9280
	10min	8765	5750	10435	4905	8730
AVG	1min	8866,7	5975,0	10871,7	5466,7	10645,0
	5min	**8773,3**	5788,3	10680,0	4948,3	9038,3
	10min	8826,7	**5741,7**	**10515,0**	**4680,0**	**8620,0**
Target	**/**	**10015**	**6700**	**14325**	**5890**	**12955**

Figure 1: Table with the experimental values for different seeds en duration.

From the table we can deduce that we overall get better results when the program runs for a longer duration. There are some exceptions though, this is prabably due to the fact that we were not able to completely avoid randomness in the program which led to results differing from those from a previous execuction.

It is evident that we get different results when changing the value of the random seed, since the randomness with wich we do mutations and accept worse results is seen as the power of simulated annealing. We opted for using three different seeds as sample for the experiment, and did not see any large differences between the results. We can safely assume that our program consistently renders good results.

4 Conclusion

From the results we can make several conclussions. Namely from Figure **??** we can conclude that our algorithm achieves better results than our our predetermined goals. This can be attributed to the succes of simulated annealing, this method is very effective and efficient for this kind of problem. Also our function that makes sure the temperature always runs through each value between the set start and end value gives a good result even when given only a small period.

Generally we can see that the longer the program runs, the lower the cost becomes. However, this is not always the case. These edge-cases mainly happen with smaller search spaces (less cars and less zones) since it is harder to jump to another solution from its current maxima.

A possible expansion of the algorithm would be to give each thread a different initial solution. This would make each thread start in a different part of the search space and avoid local maxima. Another potential expansion would be to restart a thread after it was unable to find a better value after a while and/or the temperature reaches its end. The randmomness of the algorithm would also make that thread find different solutions the second time.