Artificial Intelligence

Self-Driving Rides

Final report slides for AI Project 1
March 2020

Francisco Gonçalves
Luís Ramos
Martim Silva

Class no. 4 (3MIEIC04)

Introduction

- Millions of people commute by car every day; for example, to school or to their workplace. We'll be looking at how a fleet of self-driving vehicles can efficiently get commuters to their destinations in a simulated city using numeric notation in text files (**.in** and **.out**).

Project goals

- This project aims to solve an optimization problem of self driving rides, with the goal of efficiently getting commuters to their destinations on time, by assigning optimal rides to vehicles on the map.

Task

- There should be a list of pre-booked rides in a city as well as a fleet of self driving rides given as input and with this information the program should be able to determine an efficient solution.

Problem Description

Map: The city is represented by a rectangular grid of streets, with R horizontal streets (rows) and C vertical streets (columns). Street intersections are referenced by integer, 0-based coordinates of the horizontal and the vertical street. For example, [r, c] means the intersection of the r-th horizontal and the c-th vertical street ($0 \leq r < R$, $0 \leq c < C$).

Vehicles: There are F vehicles in the fleet. At the beginning of the simulation, all vehicles are in the intersection [0, 0]. There is no limit to how many vehicles can be in the same intersection.

Time and distance: The simulation proceeds in T steps, from 0 to $T - 1$. The distance between two intersections is defined as the minimum total number of city blocks (cells in the grid) that a vehicle has to pass in each direction to get from one intersection to the other. That is, the distance between intersection [a, b] and intersection [x, y] is equal to $|a - x| + |b - y|$.

Rides: There are N pre-booked rides. Each ride is characterized by the following information:

- **start intersection** – to begin the ride, the vehicle must be in this intersection.
- **finish intersection** – to end the ride, the vehicle must be in this intersection. Finish intersection is always different than start intersection.
- **earliest start** – the earliest step in which the ride can start. It can also start at any later step.
- **latest finish** – the latest step by which the ride must finish to get points for it

Note that the given “latest finish” step is the step in which the ride must already be over (and not the last step in which the vehicle moves) – see example below.

Simulation: Each vehicle makes the rides you assign to it in the order that you specify:

- first, the vehicle drives from its current intersection ([0,0] at the beginning of the simulation) to the start intersection of the next ride (unless the vehicle is already in this intersection)
- then, if the current step is earlier than the earliest start of the next ride, the vehicle waits until that step
- then, the vehicle drives to the finish intersection ○ the vehicle does this even if the arrival step is later than the latest finish; but no points are earned by such a ride
- then, the process repeats for the next assigned ride, until the vehicle handles all scheduled rides or the simulation reaches its final step T (whichever comes first)
- any remaining assigned rides are simply ignored

Problem Formulation

Evaluation Function

```python
def score_ride(car_to_score, ride_to_score, bonus_to_score):
    drive_distance = ride_to_score.start_position.distance(ride_to_score.destination_position)
    pick_distance = car_to_score.position.distance(ride_to_score.start_position)
    wait_time = max(0, ride_to_score.earliest - (car_to_score.current_t + pick_distance))
    on_time = pick_distance + car_to_score.current_t <= ride_to_score.earliest

    return drive_distance - pick_distance - wait_time + (bonus_to_score if on_time else 0)
```

Solution Representation

The solution is provided in an output file (.out).
Each **line** corresponds to a **vehicle**.
The first number (let's call it **n**) is the amount of rides that have been assigned to each vehicle. Following this number there another **n** numbers representing the index of each assigned ride.
On the right there is an extract of an output file (first vehicle - first line - has **3** rides [**85, 121, 51**]).

```
3 85 121 51
3 168 188 219
3 36 32 196
3 116 40 292
4 221 174 291 274
4 234 282 199 183
3 212 239 289
4 108 53 118 151
3 249 185 145
```

Implementation

The project will be developed mostly in **Python 3** with a possible visualization using JavaScript.

The development environment used by the group to develop the project is **PyCharm** by JetBrains.

Data structures that will be used are trees for the most part.

The project directory will follow this structure:

- **doc**: where all documents will be stored;
- **src**: where the project's source code is located;
  - **objects**: folder that contains all the classes, representing the entities involved: Car, Ride, Position, Rides, FIFO, Car Genetic Rides.
  - **solvers**: solver files for all implemented algorithms: **hill climbing**, **simulated annealing**, **genetic** (car), **genetic** (rides)
- **assets**: where code developed by others and the **input** is located (5 different files).

Each input file contains 6 integers in each line. The first line contains [**grid rows, grid columns, vehicles, rides, bonus per ride, steps**]. All the other lines contain [**row1, column1, row2, column2, earliest start, latest finish**].

```
[0] from (0,0) to (1,3) via car 1
[2] from (2,0) to (2,2) via car 2
[1] from (1,2) to (1,0) via car 2
Score for file a_example -->        10
```

```
hill_climbing × rides_genetic × car_genetic × simulated_annealing ×
/usr/bin/python3.7 /mnt/1AAE98A8AE987DC7/IART/Projects/P1/src/solvers/car_genetic.py

GENETIC (CAR)
a_example           time 0.1681s with score 10
b_should_be_easy    time 59.7397s with score 176,877
c_no_hurry          time 6767.7762s with score 11,048,950
d_metropolis        time 2767.8569s with score 8,678,841
e_high_bonus        time 6332.6728s with score 21,243,463
Global score is 41,148,141
Global runtime is 15928.2139s

Process finished with exit code 0
```

```
hill_climbing × rides_genetic × car_genetic × simulated_annealing ×
/usr/bin/python3.7 /mnt/1AAE98A8AE987DC7/IART/Projects/P1/src/solvers/hill_climbing.py

HILL CLIMBING

a_example           time 0.0008s with score 4
b_should_be_easy    time 0.5055s with score 164,636
c_no_hurry          time 415.8198s with score 6,704,699
d_metropolis        time 502.1818s with score 4,189,112
e_high_bonus        time 1032.0356s with score 15,706,741
Global score is 26,765,192
Global runtime is 1950.5438s

Process finished with exit code 0
```

Approach

We started by creating objects for the project:

- **Position:** class containing information of a Position object (for the cars and ride starting position and destination position);
- **Ride:** class containing information regarding a ride;
- **Car:** class containing information regarding a car, including the score(fitness) and the rides assigned to this car.

Then we implemented a greedy algorithm based on already implemented algorithms found online (References slide) to give us a proximate value for the scores. After this we decided to implement a genetic algorithm based again on already implemented algorithms, having done this we realized that we had implemented a not very generic genetic algorithm since we were iterating on the cars and not on the rides, so we went back and implemented a genetic algorithm based on the rides vector by simply switch rides from one car to another.

After implementing this genetic algorithm we decided to implement hill climbing and simulated annealing and realized that we could adapt the approach done in the implementation of the genetic algorithm (based on the rides vector) to implement these ones. Our hill climbing algorithm was implemented in a mostly deterministic way, in the sense that choosing the best neighbor is done deterministically by comparing all neighbors but the first solution is obtained by randomly assigning cars to all rides. The simulated annealing initial solution is done in the same way but choosing the best neighbor is done by choosing the best random neighbor found or randomly choosing a worse solution.

The evaluation function used in all our implementations (except greedy) is the score that would be obtained by that current solution.

The operators used by the genetic algorithms are:

- **Reproduce**: crosses two solutions and returns two childs where the first child contains a part of a parent solution and a part of the other parent solution and vice-versa for the second child;
- **Mutate**: randomly (based on a percentage) changes the solution of a child by switching a ride from one car to another.

Algorithms

Greedy approach

Greedy search: this algorithm was implemented before the checkpoint delivery, it gives the best results of all algorithms tested just because the evaluation function was more specific to the problem given.

Iterative improvement

Individual based:

- **Hill climbing**: this algorithm usually finds the local maximum, depending on the initial state. In our situation the initial state is calculated randomly so there is the possibility of the solution getting stuck in a very low local maximum.
- **Simulated annealing**: this algorithm usually finds a good local maximum but also depends on the initial state. In theory this is method is better than the hill climbing algorithm, but given our random based implementation this algorithm sometimes produces worse results compared to the hill climbing output. This may be due to the fact that in our situation the initial solution is highly determinant of the maximum obtainable score, since to get a better one we would have to change multiple values that when looked at separately represent worse solutions. This has a very low chance of occuring.

Population based:

- **Genetic**: this algorithm starts with a randomly generated solution and tries to generate a population of new solutions based on the first solution using reproduce and mutate operators. This algorithm tries to do this until a maximum population size and then uses the individual with maximum fitness of the population to introduce into a FIFO of fixed size whose function is to record the maximum fitness of each population. When the FIFO has a set number of equal values the best fitness individual is considered to be best solution.

Results

Individual based algorithms:

- **Hill climbing**: This algorithm usually finds the local maximum, although it can get stuck on a local minimum, depending on the initial state. In our situation we got the expected results. We implemented hill climbing random. It does not guarantee an optimal solution but produces an efficient one. We also found that the final result (total runtime and final score) is affected by the initial guess or initial state. After running the algorithm for the five input files altogether we got a global score of 26,654,680 and the total runtime was 1914.3 seconds (just under 32 minutes). The second test produced score of 26,765,192 and the total runtime was around 32.5 minutes. This confirms that the solution may vary, depending on the initial guess.
- **Simulated annealing**: After running the algorithm for the five input files altogether we got a global score of 26,543,199 and the total runtime was 6.07 seconds. Because the value of the temperature for each iteration is the value of the last temperature multiplied by 0 the temperature converges rapidly to 0 the run time of this algorithm is much faster than the other ones.

Population based algorithms:

- **Genetic**: As shown by the tables to the right, genetic rides don't seem to vary in result too much but runtime gets a lot bigger when we increase population. Genetic cars runtime gets somewhat proportionally bigger when we increase population and it also impacts the score. The best score is achieved with genetic cars, despite the heavy processing.

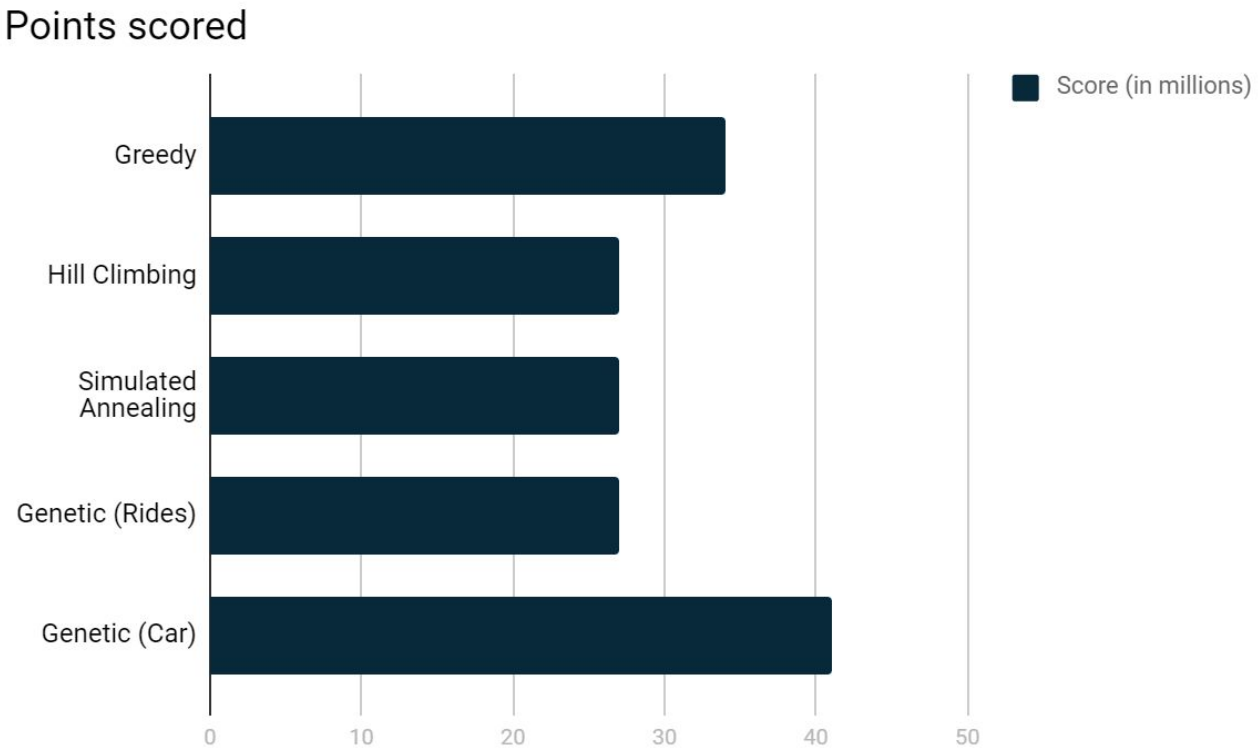
Table with values for genetic rides

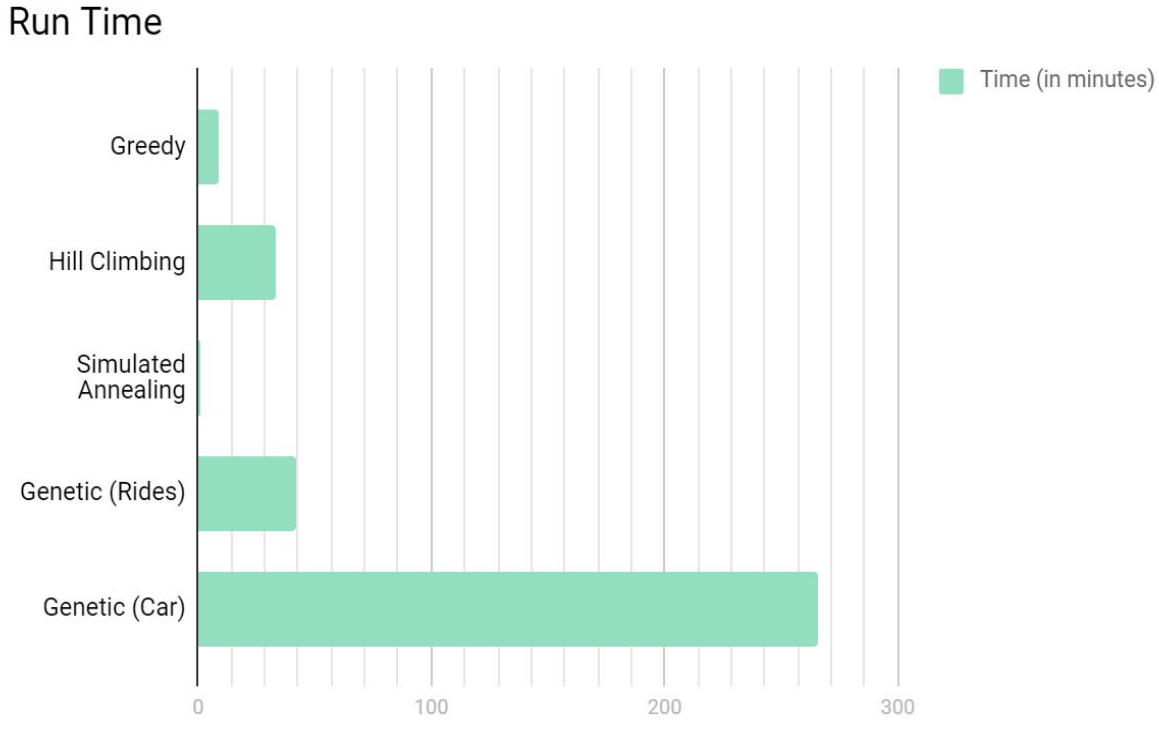
Population	Time (s)	Score (M)
5	4.8201	26.699229
10	5.7682	26.943173
20	14.0637	26.960164
40	30.4523	27.079018
80	58.7926	27.065403
160	90.4901	27.119698
320	330.9667	27.949754
640	487.7629	27.213853
1280	960.0337	27.214665

Table with values for genetic cars

Population	Time (s)	Score (M)
5	12.8279	29.674080
10	14.7317	31.245341
20	21.6146	32.811505
40	41.0956	34.642448
80	104.2826	36.268674
160	286.6843	37.545675
320	711.6655	38.610558
640	1643.1174	39.531738
1280	3428.2849	41.638023

Results analysis





It's not very fair to compare genetic algorithms with the others because these take into account variables like population size, pooling size, etc. These values were obtained with population size at 8000, pooling size at 2000, generation number 5 and mutation rate of 0.01. Also, this score and runtime refer to the global values (all 5 files were taken as input).

From the results we can see that the genetic algorithms are heavy in processing but very powerful, whilst the simulated annealing is very lightweight per say and still provides an efficient solution in comparison to the rest of them. Hill climbing gives decent results but also takes quite some time processing.

Conclusion

- The fact that this project's topic combines important matters of Artificial Intelligence and a very relevant issue that is adequately assigning rides for commuters to arrive at their destinations in an efficient way, made the implementation and research process a lot more engaging. The project also helped with getting a more in-depth knowledge of the contents discussed in both lectures and practical classes.
- After some tryouts we were able to understand how and what parameters influence the results for each algorithm and gain insight on how these methods actually behave for our specific situation of self driving rides.
- To sum up, the assignment was successfully completed and we believe the project goals were fulfilled. It was a good first contact with the field of AI.

References

During the research stage of the project, the group came across several implementations of algorithms that could be adapted and used them our project, such as Python implementations of greedy algorithms, genetic algorithms and web articles explaining different approaches. We also used the slides from IART lectures.

Greedy:

- https://github.com/n1try/hashcode-2018/blob/master/qualification/python/
- https://github.com/papachristoumarios/HashCode-Team111/blob/master/selfdriving.py

Genetic:

- https://github.com/schesa/hashcode2018
- https://github.com/devspaceship/google-hash-code-2018

Simulated Annealing:

- https://www.sciencedirect.com/topics/computer-science/cooling-schedule

Hill climbing:

- https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/

Others:

- https://medium.com/plapadoo/lessons-learned-from-google-hash-code-9982e5e207d
- https://albertherd.com/2018/03/03/google-hash-code-2018-solution-and-source-code-1st-in-malta-and-top-20-worldwide