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

- The **evaluation function** follows the score calculation function:
- Fitness / score = distance between starting position and destination position of the ride should the car get to the destination position on time plus a bonus if the car is at the starting point at the earliest possible time to start the ride.

Neighbourhood functions:

- A neighbour solution is a solution where the car associated with a ride is changed to another car, has such, each solution will have a number of neighbours equal to the number of cars times the number of rides minus one (current solution).

Mutation and Crossover functions

- **Selection**: ordering the population by score and using the highest scoring individuals (with a fixed polling size) to generate a new population.
- **Reproduction**: crosses two solutions and returns two children 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;
- **Mutation**: randomly (based on a percentage) changes the solution of a child by switching a ride from one car to another.

Representation of the solution

- Each element of the array of Rides is a ride and when the algorithm finishes each of these rides has a car associated with it. This represents the best available car that was selected for that particular ride.

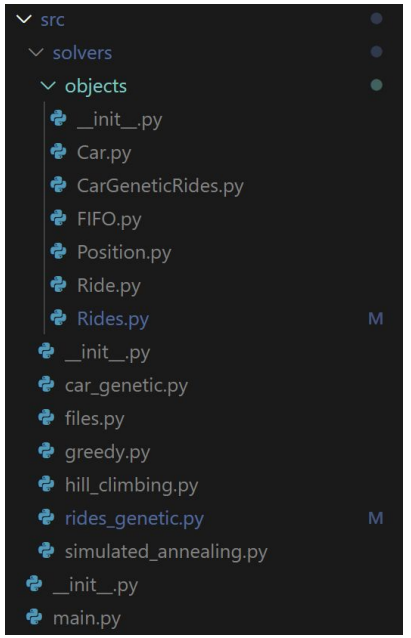
Implementation

The project will be developed mostly in **Python 3**

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

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) and **greedy**
- **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**] regarding the overall information. All the other lines contain [**row1, column1, row2, column2, earliest start, latest finish**] regarding each ride.

```
3 4 2 3 2 10
0 0 1 3 2 9
1 2 1 0 0 9
2 0 2 2 0 9
```

Input example

Each line of the output file corresponds to a vehicle. The first number is the number 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]).

```
0
3 1 2 0
```

Output example

Approach

- The **evaluation function** used was implemented in the Car class, it is used to calculate all overall score of the rides assigned to it, and is used by the Rides class to calculate the fitness/score of all the cars it has.
- The **heuristic used** was just the global score(fitness) obtained by the solution. As such the choice was usually made by choosing the solution that would obtain a higher score.

```
def calculate_fitness(self):
    self.position = Position(0, 0)
    self.current_t = 0
    self.fitness = 0
    self.sort_rides()

    for ride in self.rides:
        self.current_t += self.position.distance(ride.start_position)
        self.position = ride.start_position
        ride.score = 0

        time = max(ride.earliest, self.current_t)

        if time == ride.earliest:
            self.fitness += int(self.BONUS)

        self.current_t = time + ride.distance

        if self.current_t <= ride.latest:
            self.fitness += ride.distance

        self.position = ride.destination_position
    return self.fitness
```

Operators:

```
population.sort(key=lambda rides_elem: rides_elem.fitness, reverse=True)

if population[0].fitness > max_fitness_rides.fitness:
    max_fitness_rides = population[0]
fitness_pile.put(max_fitness_rides.fitness)

new_population = []
while len(new_population) < POPULATION_SIZE:
    children = population[random.randrange(0, POOLING_SIZE)].reproduce(
        population[random.randrange(0, POOLING_SIZE)])
    children[0].mutate()
    children[1].mutate()
    new_population.append(children[0])
    new_population.append(children[1])
```

Heuristic being applied

Reproduction

Selection

Mutation

```
def reproduce(self, parent):
    children = []
    i = random.randrange(Rides.N_RIDES)
    children.append(Rides(self.rides[0:i] + parent.rides[i:len(self.rides)]))
    children.append(Rides(parent.rides[0:i] + self.rides[i:len(self.rides)]))
    return children

def mutate(self):
    for ride in self.rides:
        if random.random() < Rides.MUTATION_RATE:
            car = random.randrange(Rides.N_CARS)
            self.cars[ride.car].remove_ride(ride)
            self.cars[car].add_ride(ride)
            ride.car = car
```

Algorithms

Greedy approach

Greedy search: starting with an empty solution for every ride we assigned it to every car and registered what car gets a higher score, the one that does, is assigned the ride. When all the rides were assigned the solution was completed.

Iterative improvement

Individual based:

- **Hill climbing (random)**: starting with a randomly generated solution the algorithm calculates the score of a set number of random neighbours solution and the one with the highest score is considered the best and is used for the next round of calculating the best random neighbour. When the best neighbour has the same score has the current solution the solution is considered to be maximum.
- **Simulated annealing**: starting with a randomly generated solution the algorithm and a temperature equal to the number of rides the algorithm calculates the score of a random neighbour, if the score is higher than the current solution's score it is carries on to the next cycle with that solution, otherwise it has a probability of carrying on to the next cycle equal to: $e^{\Delta_{fitness}/T}$. After each cycle the temperature is decreased at a random rate (between 0.8 and 0.99). The algorithm stops calculating the next solution when the temperature reaches 0 and the current solution is not improved.

```
while previous_score < solution.fitness:
    previous_score = solution.fitness
    solution = solution.hill_climbing_random()
```

```
def hill_climbing_random(self):
    population = []

    for rideIndex in range(0, self.N_RIDES):
        ride = random.choice(self.rides)
        car = random.randrange(Rides.N_CARS)

        swap = ride.car
        ride.car = car

        population.append(Rides(self.rides))

        ride.car = swap
```

Hill climbing

```
temperature = len(rides)

while temperature > 0.1 or previous_score != solution.fitness:
    previous_score = solution.fitness
    solution.simulated_annealing(temperature)
    temperature = random.uniform(0.8, 0.99) * temperature
```

```
def simulated_annealing(self, temperature):
    fitness = self.fitness
    Mk = (self.N_RIDES * math.sqrt(temperature + 1))

    for m in range(0, int(Mk)):
        ride = random.choice(self.rides)
        car = random.randrange(Rides.N_CARS)

        swap = ride.car
        ride.car = car

        if self.calculate_fitness() > fitness:
            return

        elif random.random() <= math.exp((self.fitness - fitness) / temperature):
            return

        self.fitness = fitness
        ride.car = swap
```

Simulated annealing

Algorithms (continued)

Population based:

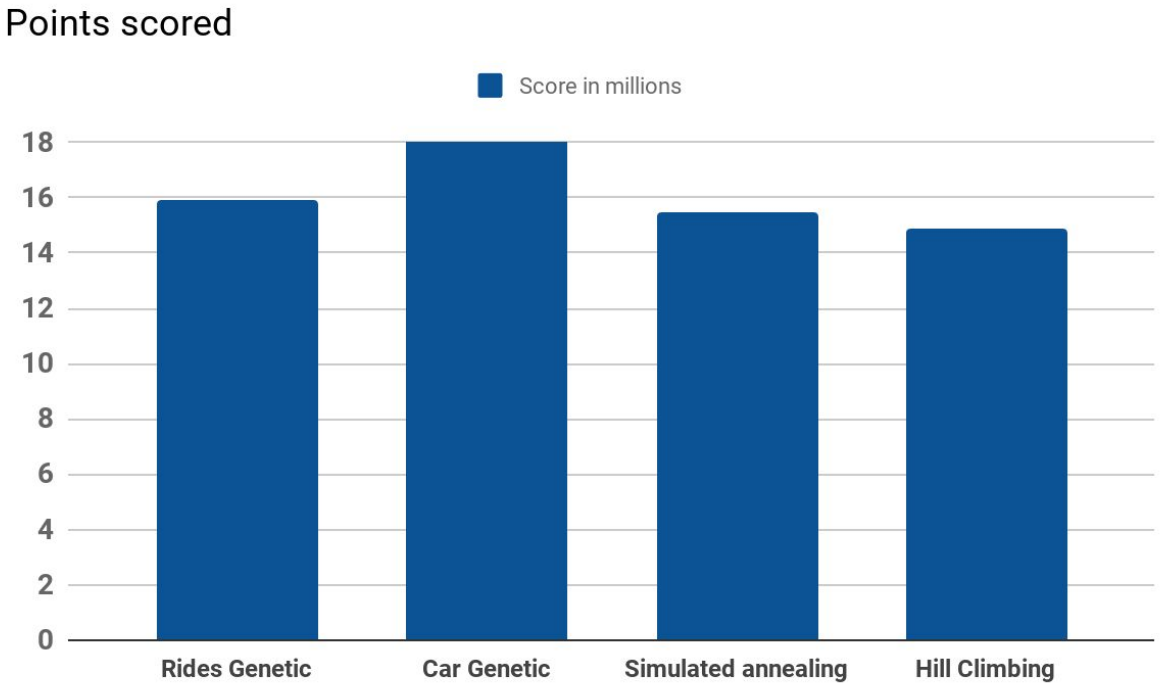
- **Genetic (rides)**: the algorithm starts by calculating a generation with individuals with the rides randomly assigned to each car. Every generation the algorithm builds a population by reproducing the last population's best fitness individuals and then calls the mutation function for each child generated to generate even more different solutions. The algorithm keeps track of the maximum fitness in a FIFO with fixed size so that it can see if the generation was better or worse, if the generations fail to produce a better individual a set number of times(equal to the size of the FIFO) the maximum fitness individual in the FIFO is considered the optimal solution.

- **Genetic (cars)**: the algorithm starts by calculating a generation of Cars with a fixed number (the number of rides divided by the number of cars) of randomly assigned rides. It then chooses the Car with the best fitness and uses it to generate a new population with the same logic as the genetic rides algorithm. After arriving to the optimal individual, the rides assigned to this Car are removed from the available rides pool. It does this for all the Cars available except for the last one. To the last Car are assigned the remaining rides in the pool, meaning those that have no Car assigned to them.

- **Genetic (rides) vs Genetic (cars)**: We initially started by making the Genetic algorithm for the cars, but after a online meeting with the professor, it was decided to do a more generic approach. Genetic rides is more generic because in genetic cars we are dividing the problem into many parts (individual Cars).

```
while not fitness_pile.is_constant():
    for rides in population:
        rides.calculate_fitness()

    population.sort(key=lambda rides_elem: rides_elem.fitness, reverse=True)

    if population[0].fitness > max_fitness_rides.fitness:
        max_fitness_rides = population[0]
    fitness_pile.put(max_fitness_rides.fitness)

    new_population = []
    while len(new_population) < POPULATION_SIZE:
        children = population[random.randrange(0, POOLING_SIZE)].reproduce(
            population[random.randrange(0, POOLING_SIZE)])
        children[0].mutate()
        children[1].mutate()
        new_population.append(children[0])
        new_population.append(children[1])

    population = new_population
    generation += 1
```

Results analysis



- We measured the score for each algorithm by making a few tweaks on each method (reducing **Rides.NRIDES** for **hill climbing**, **slow temperature decrease** for **simulated annealing**, **pooling size**, for example), so that, the runtime of each procedure would be around the same value (it would not make sense to compare algorithms that take different time and made different number of iterations).
- This chart was calculated based on the average of 2 tests for each algorithm using always the same file (e_high_bonus.in). This way we can better control the differences that could appear on this chart, if we used different files.
- With these results, we can see that both genetic algorithms managed to outperform the individual based ones.

The tweaks we made were only experimental and therefore temporary.
As it is, simulated annealing provides good results with very few runtime/iterations (because we used **temperature = random.uniform(0.8, 0.99) * temperature** and this lowers the temperature very rapidly).
Hill climbing random produces similar results but is using up a lot more runtime/iterations.
The genetic algorithms are set to have population of 80 and pooling size has been set to 40% of that. Mutation rate is 0.01 and generation number is 6.

Results analysis (continuation)

With about the **same** number of iterations/runtime (somewhere around 100 iterations) we ran the 2 individual based algorithms 10 times and noticed that they average a very **similar** score. The average and standard deviation for **hill climbing** were 15.663251 and 0.08340, respectively. For the **simulated annealing** algorithm these values were 15.693856 and 0.06689.

The picture to the right represents the values for each iteration of both algorithms. The **orange** line represents the **average**. The difference between the 2 average values (0.0306052) is **smaller than the both the standard deviations**, meaning that the 2 algorithms are not very different in result. To get more precise information about the behavior of the two algorithms we would have to do this analysis many times increasing the number of iterations for each test.

We also compared the 2 genetic algorithms that we implemented and found that the **genetic cars** grow significantly more in **score** upon increasing the **population** when compared to the **genetic rides**.

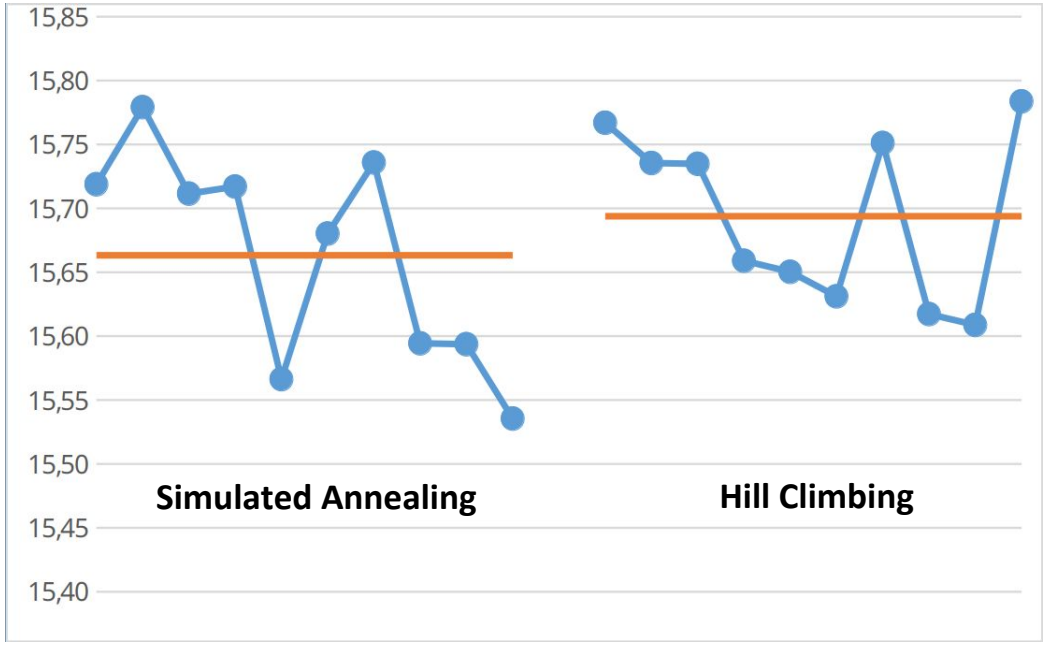


Table with values for genetic rides

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

Conclusion & References

- 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 analysing the results, by changing and tweaking some parameters, we started to notice the influence of certain attributes and the consequences of changing them. All in all, the assignment was successfully completed and we believe the project goals were fulfilled.

Here we define all references and articles used during our research process

- **Greedy**: https://github.com/n1try/hashcode-2018/blob/master/qualification/python/
- **Genetic**: https://github.com/devspaceship/google-hash-code-2018
- **Simulated Annealing**: https://www.sciencedirect.com/topics/computer-science/cooling-schedule
- **Others**: https://medium.com/plapadoo/lessons-learned-from-google-hash-code-9982e5e207d