Artificial Intelligence

Self Driving Rides

Checkpoint slides for AI Project 1
March 2020

Francisco Gonçalves
Luís Ramos
Martim Silva

Class no. 4 (3MIEIC04)

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.
- 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 ($0 \le r < R$, $0 \le c < C$).

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

References

During the research stage of the project, the group came across several implementations of algorithms that can be adapted and used on our own project, such as Python implementations of greedy algorithms, genetic algorithms and web articles explaining different approaches.

Greedy Approaches:

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

Genetic Algorithm:

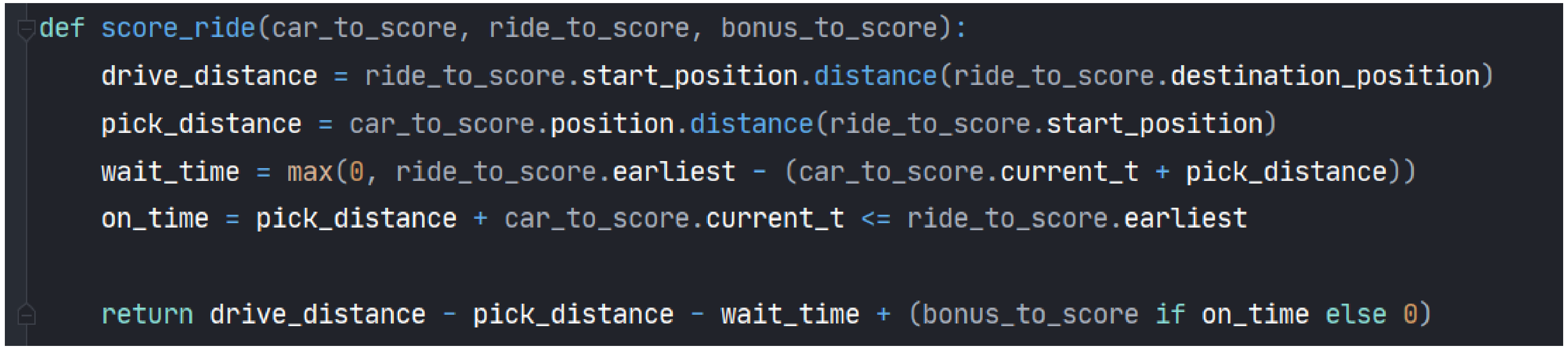
- https://github.com/schesa/hashcode2018
- https://github.com/devspaceship/google-hash-code-2018
- https://albertherd.com/2018/03/03/google-hash-code-2018-solution-and-source-code-1st-in-malta-and-top-20-worldwide/

Article:

- https://medium.com/plapadoo/lessons-learned-from-google-hash-code-9982e5e207d

Problem Formulation

Evaluation Function

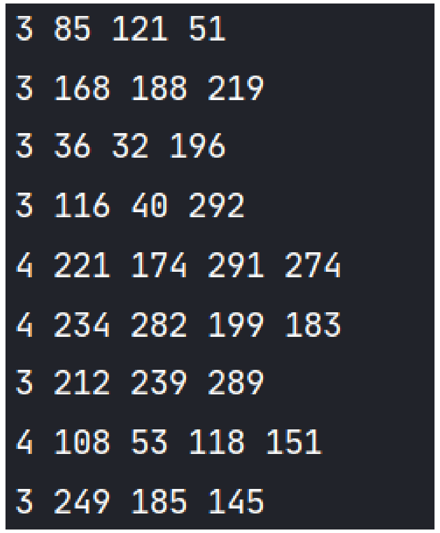


```python
def score_ride(car_to_score, ride_to_score, bonus_to_score):
    drive_distance = ride_to_score.start_position.distance(ride_to_score.destination_position)
    pick_distance = car_to_score.position.distance(ride_to_score.start_position)
    wait_time = max(0, ride_to_score.earliest - (car_to_score.current_t + 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;
- **assets**: where code developed by others and the **input** is located (5 different files).

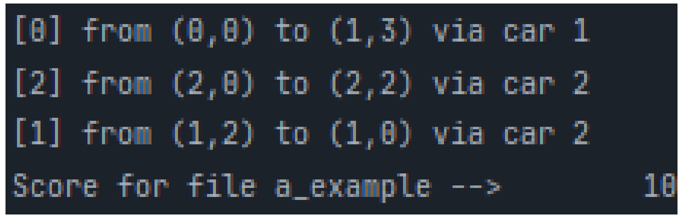
Each input file contains 6 integers in each line. The first line contains [**grid rows, grid columns, vehicles, rides, bonus per ride, steps**].
All the other lines contain [**row1, column1, row2, column2, earliest start, latest finish**].

The **greedy** approach has already been implemented by the group.

As it is, the program can be run with or without arguments. If the program is executed with an argument it should be the title of an input (.in) file. If not, the program will test all 5 input files determining the sum of all 5 scores. Both ways of running will print the details of all rides for each tested input file.

1st and 2nd files are almost instantly processed and as for the other 3 they all take around 4 and half minutes each (almost 14 minutes in total).

The console will show the details of every ride and when it's done it will show the score for the input file (using the evaluation function)



```
[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
```