Exercise 2: A Reactive Agent for the Pickup and Delivery Problem

Group №22: Dubuis Samuel, Facklam Olivér

October 6, 2020

1 Problem Representation

1.1 Representation Description

We denote V the set of vertices (cities). The chosen set of states is $S = V \times (V \cup \{nil\})$. A state $s = (loc, dst)$ is a couple where:

- the first element loc is the current city the agent is in,
- the second element dst is the destination of the proposed task, or nil if no task is proposed.

The set of actions is $A = \{Accept\} \cup \{Move(c) \text{ for } c \in V\}$. So an action a can be either

- picking up and delivering the proposed task ($Accept$): this action transports the agent directly to the destination (over the shortest path) and gives the reward corresponding to the task
- ignoring the task and moving to a certain city c ($Move(c)$)

The reward function can be represented in the following table, with $w(c, c')$ denoting the weight of the edge between c and c' (or ∞ if no edge exists) and $d(c, c')$ denoting the shortest distance between c and c'. We note α the cost per km of the vehicle.

$R(s,a)$	$a = Accept$	$a = Move(c)$
$s = (loc, nil)$	$-\infty$ (impossible)	$-\alpha w(loc, c)$
$s = (loc, dst)$	$r(loc, dst) - \alpha d(loc, dst)$	$-\alpha w(loc, c)$

Table 1: Reward function representation

The transition function can be represented using two tables depending on the chosen action a:

$a = Accept$		
$T(s, a, s')$	$s' = (loc', dst')$ with $loc' = dst$	$s' = (loc', dst')$ with $loc' \neq dst$
$s = (loc, nil)$	0	0
$s = (loc, dst)$	$p(loc', dst')$	0

$a = Move(c)$		
$T(s, a, s')$	$s' = (loc', dst')$ with $loc' = c$	$s' = (loc', dst')$ with $loc' \neq c$
$s = (loc, dst)$ no edge (loc, c)	0	0
$s = (loc, dst)$ with edge (loc, c)	$p(loc', dst')$	0

Table 2: Transition function representation

1.2 Implementation Details

We decided to create three auxiliary classes to help implement our `Reactive` agent.

We first defined a `State` class which is a simple implementation of the states we defined in the previous subsection. A `State(City loc, City dst)` contains public functions returning task details. In addition, the `equals` and `hashCode` functions were implemented to allow a `State` to be used as key in a `HashMap`.

Next class was the `Response` one, which simply represents an agent's potential action: the Accept/Pickup act, or the Move act. It only contains a `city` variable, which is the move destination or `null` for a pickup.

Finally we coded a much bigger class called `Learning`, which implements the actual Reinforcement Learning Algorithm (RLA), taking care of the Q table optimization until convergence. Its constructor is `Learning(TaskDistribution td, int costPerKm, double discount)` and it also implements a couple of necessary functions like `reward(State s, Response a)` or `transition(State init, Response a, State next)` which return respectively the expected profit from an action in a certain state, and the probability of transition to next potential states (as defined in the tables from the previous subsection).

Finally, the `Reactive` class implements our RLA agent. During setup, after creating all possible states and actions, we run our optimizer from our `Learning` class to generate the agent's policy. We also implemented a quick `CsvWriter` class to help us easily export data points in order to output better plots.

2 Results

2.1 Experiment 1: Discount factor

For this first experiment, we analyze the discount factor's impact on our reactive agent's performance. We modify the discount value and compare the corresponding profits.

2.1.1 Setting

We run the experiment for multiple discount factor values (`discount-factor=[0.01, 0.3, 0.6, 0.85, 0.99]`) and for 10,000 ticks, on the same simulation so that all have same cities and probabilities.

2.1.2 Observations

`discount-factor`	Average profit
0.01	37860
0.3	37618
0.6	39509
0.85	38878
0.99	38923

Table 3: Average profit depending on discount factor

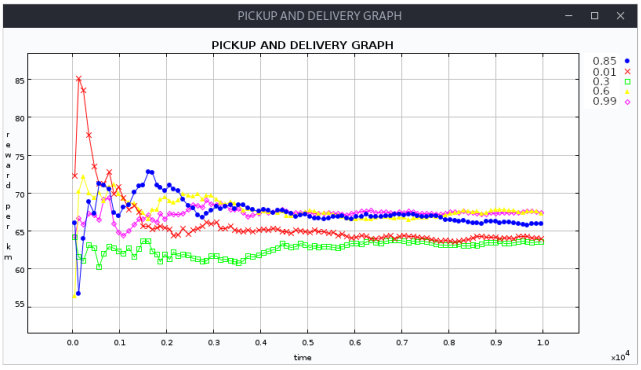


Figure 1: Reward per vehicle, the best three results are values closer to 1.

As we can see in both Table 3 and Figure 1, the closer the discount factor is to the value 1, the better the results are on average, considering the reward per kilometer. In figure 1, pink is discount equal to 0.99, yellow is 0.6, blue is 0.85, red is 0.01 and finally green 0.3. However, to nuance this fact we see that the highest discount factor might not give the very best average profit on a simulation.

2.2 Experiment 2: Comparisons with dummy agents

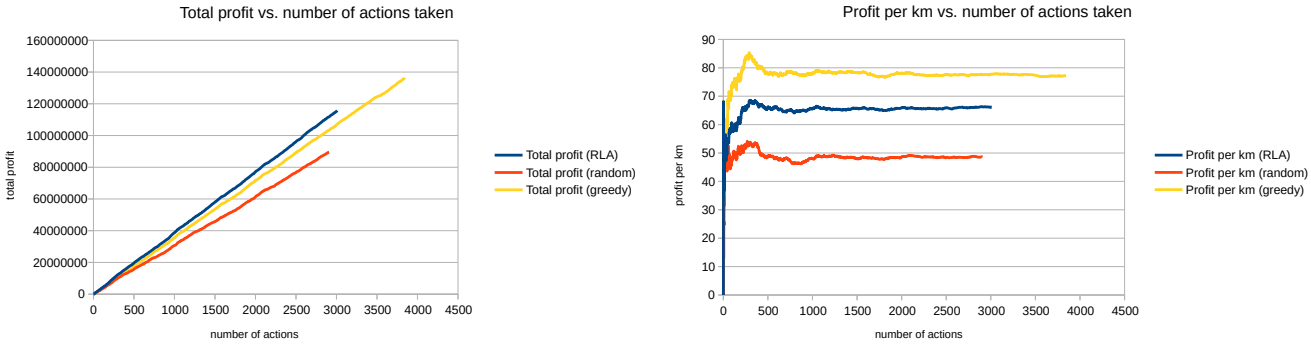
In this second experiment, we compare our reactive agent to two other different ones.

- The first one is the random dummy agent given in the template, that picks up a task at its location if there's one (with a certain probability $p = 0.85$), otherwise randomly moves to another city.
- The second dummy agent is actually a greedy one. It only looks one step ahead: "Is it better to take the current task or hope the next city has a better one?" It differs from our reactive agent which looks infinitely far.

2.2.1 Setting

We implemented here a new class `ReactiveGreedy` that sets up the greedy agent. It checks if a task is available at the current city, then checks neighbors and their probability to have a better task. We also let this experiment run for 10,000 ticks.

2.2.2 Observations



(a) Total profit

(b) Profit per km

Figure 2: Profit measures for 3 different agents (RLA, random, greedy), as a function of number of actions

Agent	Average profit
RLA	38453
Greedy	35459
Random	30769

Table 4: Average profit of each agent based on its type.

The simulation results are displayed on figure 2. The greedy agent logically has the best profit per kilometer as it always tries to get some immediate rewards (figure 2b). It also does more actions in 10,000 ticks. However, in figure 2a and table 4, we see that the RLA reactive agent gets the highest profit per action, which is the metric it was designed to optimize. The random agent has the worst performance in all metrics.

2.3 Conclusion

Both experiments show us the power of a simple reactive agent that learns a policy to optimize the Q table. The second experiment especially shows the potential of having a smarter agent that learns its state, possibilities and optimal actions before even running the simulation, compared to a greedy agent. Even if our greedy agent was planning one step ahead, it wasn't enough to beat the RLA agent which optimizes the (discounted) total value over infinite steps.

It's interesting to compare the ratio of the number of actions done to the total profit. This is where we really understand the capacity of our reactive agent to be very efficient.