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

Group № 38: Andrej Janchevski 309143, Orazio Rillo 313423

October 5, 2020

1 Problem Representation

1.1 Representation Description

According to the documentation of the Logist platform and the exercise description, we assumed that when a vehicle decides to pick up a new task, it is immediately teleported to the location in which the task has to be delivered.

With this fundamental hypothesis stated, let us define our state representation. Let S be the set of all the possible states in our model and C be the set of all the cities on the map. So:

$$s \in S \iff s = (c_L, c_D) \text{ with } c_L \in C, c_D \in (C \cup \{null\}),$$

where c_L represents the current location of the vehicle and c_D represents the destination in which the task carried by the vehicle has to be delivered if any task is available in c_L, otherwise it is *null*. With this definition $|S| = |C|(|C| + 1) = O(|C|^2)$.

Let now A be the set of all possible actions among which a vehicle can choose. There are $|C|$ of *moveTo* actions (one action for each city), plus an extra *pickup* action. So, $A = \{0, 1, \ldots, |C|\} \subset \mathbb{Z}$ and in total, we have $|C| + 1 = O(|C|)$ actions.

Once we have state and action representations, we can illustrate the reward table's construction. Let us, in fact, define the (future) reward R obtained in state s by choosing action a as follows:

$$R(s,a) = R(c_L, c_D, a) = \begin{cases} \mathbb{E}[r(c_L, c_D)] - K \cdot distance(c_L, c_D), & \text{if } a \text{ is } pickup \wedge c_D \neq null \\ -K \cdot distance(c_L, C[i]), & \text{if } a \text{ is } moveTo(i) \wedge neighbors(c_L, C[i]) \\ -\infty, & \text{otherwise} \end{cases}$$

where K is a constant value that represents the cost per km we pay by moving a vehicle and $\mathbb{E}[r(c_L, c_D)]$ is the (expected) reward for a task between two cities (from c_L to c_D). We note that when a is a *pickup* action we require that $c_D \neq null$ since the fact that we chose to *pickup* implies that we should have in the vehicle's state the information about the city in which the picked up task has to be delivered. In the second case when a is a $moveTo(i)$ action we require that c_L and $C[i]$ (the destination city of the $moveTo(i)$ action) are neighbours, otherwise the movement is prohibited.

Let us finally define our transition table's representation:

$$\begin{aligned} T(s,a,s') &= T((c_L, c_D), a, (c'_L, c'_D)) = \mathbb{P}(\{c'_L, c'_D\} | \{c_L, c_D, a\}) \\ &= \begin{cases} \mathbb{P}(\{c'_L, c'_D\}), & \text{if } a = pickup \wedge c_D \neq null \wedge c'_L = c_D \wedge pathExists(c_L, c_D) \wedge c'_D \neq null \\ \mathbb{P}(\{\text{no task in } c'_L\}), & \text{if } a = pickup \wedge c_D \neq null \wedge c'_L = c_D \wedge pathExists(c_L, c_D) \wedge c'_D = null \\ \mathbb{P}(\{c'_L, c'_D\}), & \text{if } a = moveTo(i) \wedge c'_L = C[i] \wedge neighbors(c_L, C[i]) \wedge c'_D \neq null \\ \mathbb{P}(\{\text{no task in } c'_L\}), & \text{if } a = moveTo(i) \wedge c'_L = C[i]] \wedge neighbors(c_L, C[i]]) \wedge c'_D = null \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

In order to prove the correctness of our representation, we will show that these transition probabilities satisfy the unit sum property, as $\forall s, a$, following from the independence of the r.v.'s c_D and c'_D:

$$\sum_{s' \in S} T(s, a, s') = \sum_{s' \in S} \mathbb{P}(\{c'_L, c'_D\} | \{c_L, c_D, a\}) = \sum_{c'_D \in C} \mathbb{P}(\{c'_L, c'_D\}) + (1 - \sum_{c'_D \in C} \mathbb{P}(\{c'_L, c'_D\})) = 1,$$

where the last sum is given by two contributions: respectively they are the probability to find any task starting from c'_L (which is determined since if a is a *pickup* action $c'_L = c_D$ and if a is a *moveTo* action $c'_L = C[i]$) and the probability to find no task from c'_L.

1.2 Implementation Details

It was decided to implement the vehicle state as a separate `State` class with two attributes: `location`, a `City` object representing c_L and `taskDestination`, a `City` or `null` object representing c_D. The state set is then generated and saved as an `ArrayList<State>`. The actions are represented through their integer codes in the set $\{0, 1, \ldots, |C|\}$ and as such they do not incur a memory cost.

Many entries in the reward and transition probability tables were expected to be set to values such as 0 or $-\infty$, insinuating an impossible action. It was decided to store only non-trivial values in memory for efficiency, by utilizing `HashMap` objects. Two new classes for the map keys, `RewardTableKey` and `TransitionProbabilityTableKey`, were created. These objects represent the (s, a) and (s, a, s') tuples and are formed appropriately from `State` and action integer objects. The rewards and transition probability tables were defined to map these keys to appropriate `Double` values. A trivial value is assumed if a key is not present in the tables. With this approach it can be shown that the memory complexity is reduced to $O(|C|^2 d)$ and $O(|C|^3 d)$ respectively, where d is the maximum node degree in the topology.

Finally, the $Q(s, a)$ table and $V(s)$ state value vector required during the algorithm execution were represented as an `ArrayList<ArrayList<Double»` and `ArrayList<Double>` respectively. The Q matrix is initialized with zeroes, while the V vector entries are set to a fixed arbitrary value. Convergence is verified using the condition $\forall s \in S, |V(s)_t - V(s)_{t-1}| \leq \varepsilon$, where the precision ε can be given as a parameter to the algorithm, however we use the default value of 10^{-6}. The final derived policy is a `HashMap` mapping `State` objects to integer action codes.

2 Results

2.1 Experiment 1: Discount factor

2.1.1 Setting

For this experiment we wished to investigate how three reactive agents, trained with typical or extreme values for the discount factor parameter (γ) of the algorithm, fare against one another when sent to solve the PDP on the same map. The three agents used are defined in the `agents.xml` configuration file, having the names `reactive-rla-tiny-discount`, `reactive-rla-medium-discount` and `reactive-rla-huge-discount`. They are configured as instances of the `ReactiveTemplate` class, with the γ values set to 0.01, 0.50 and 0.99 respectively.

2.1.2 Observations

The results from these experiments are presented through the collage in Figure 1. We observed a common pattern across multiple runs and all topologies. In some cases, initially the $\gamma = 0.01$ agent possesses the advantage, as it utilizes the most greedy approach (it puts almost no weight to future rewards). However, after a certain amount of time the superiority of the other two tactics becomes evident. The performance is directly proportional to the discount factor, which is to be expected, as a higher weight on future rewards can generate a more intelligent policy.

Possibly another point worthy of mention however is the difference in convergence time of the 3 cases of the algorithm. While for the 0.01 and 0.5 values of γ we receive the policy in only about 6 and 20 iterations respectively, it can take as many as 1200 iterations for the 0.99 case.

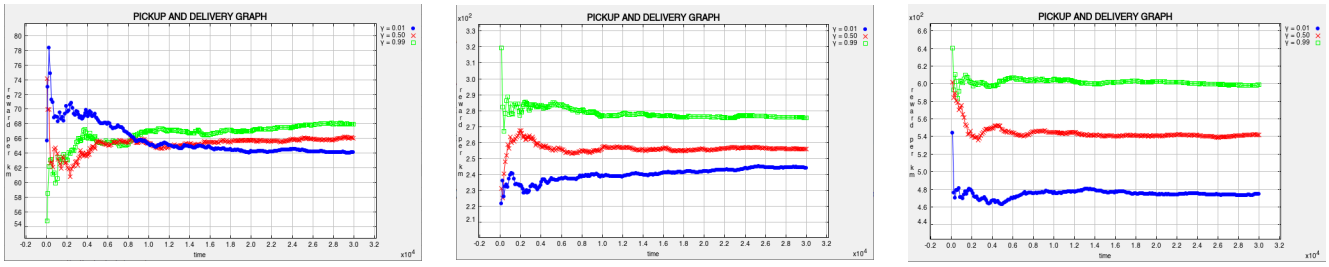

Figure 1: Plots comparing the average reward per km for three versions of our reactive agent across three topologies. From left to right: France, England, The Netherlands.

2.2 Experiment 2: Comparisons with dummy agents

2.2.1 Setting

For this experiment, we wish to validate whether our discovered policy will outperform random chance and other simplistic reactive behaviour.

For this manner, we first included a new class implementing `ReactiveBehaviour` - the `ReactiveRandom` class, which just contains the default template code provided for the exercise. The two dummy agents were configured in `agents.xml` as instances of `ReactiveRandom`. The names of these agents are `reactive-random` and `reactive-always-pickup` respectively. As a second dummy agent we wished to implement an "always pickup the task if possible" policy, and this was achieved by passing the value 1 for the `discount-factor` property, as this value is used in `ReactiveRandom` as a threshold. The third agent is our default `reactive-rla`, which utilizes the γ value of 0.85.

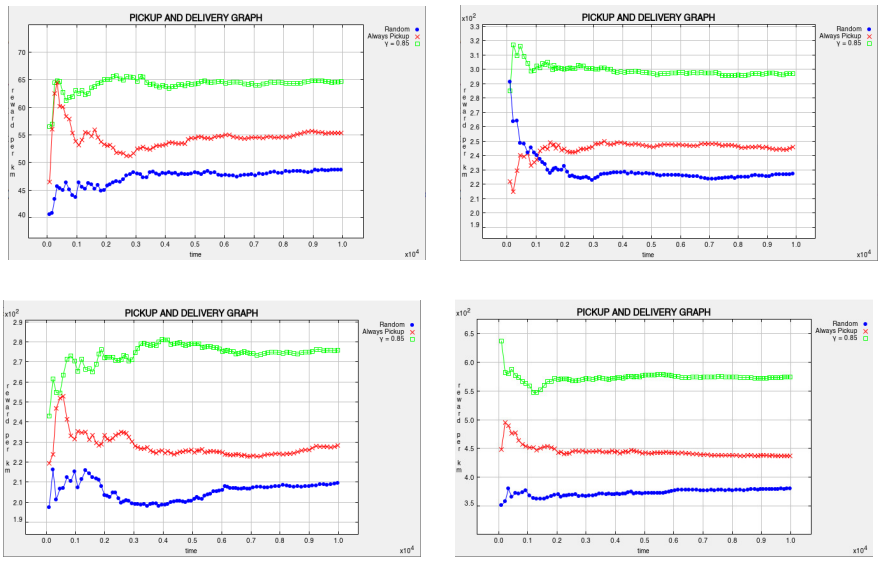

Figure 2: Plots comparing the average reward per km between our reactive agent and two dummy agents across the four topologies. From top to bottom and left to right: France, Switzerland, England, The Netherlands.

2.2.2 Observations

The results are presented in Figure 2. We observe that regardless of the underlying topology, our reactive policy outperforms the other two halfwit strategies. We also experimented with two new constructed topology: one with a circular structure and the other containing one unconnected city. However in the first case, we did not observe any new peculiar behaviours of the agents, while is was not possible to test the second experiment because the platform does not allow to have disconnected topologies. Interestingly, always choosing to deliver every task presented also outperforms random chance in all cases. We believe this is due to the relatively compact diameter of the city graphs.