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

Group №35: Aleksandar Hrusanov, Rastislav Kovac

October 6, 2020

1 Problem Representation

1.1 Representation Description

- **State Representation *s*:** Our state representation contains a pair of cities - *current* and *destination*. The former reflects the city the agent occupies at the given state. The latter either reflects that the agent has not picked up a task (e.g. has no value), or reflects the delivery city of the task the agent has picked up.

- **Possible Actions *a*:** Our action representation contains a city value, representing the city to which the action would take the agent and a type of action. We support two types of action - *move* and *pickup*. We do not have a separate *deliver* type since the agent has to deliver a task right away, once it picks one up (e.g. *deliver* is contained within *pickup*).

- **Reward Table *R(s,[a,r])*:** Our reward table has an entry for every possible state (e.g. every possible pair of *current* and *destination* cities, with *destination* value being optional). For a given state **s**, we have a set of *legal* action **a** - reward pairs **r** (see implementation details). The reward for a *move* action is negative and represents the cost for traveling the distance between two cities. The reward for a *pickup* action is the reward of delivering the corresponding task minus the cost for traveling the distance between pickup and delivery city.

- **Transition Probability Table *T(s,a,s')*:** We do not store the transition probabilities in a table, but rather use a function which is used during the offline learning. It computes the transition probability from a start state **s** to and end state **s'**, given a certain action **a** which can *legally* take the agent from **s** to **s'**. The transition probability for a *move* action to a given neighboring city is equal to the maximum probability that there exists a task from that neighboring city to any other city in the topology. In this way, we send the agent to the most *promising* neighboring city. The transition probability for a *pickup* action is equal to the probability given by the task distribution that a task exists such that its pickup city is the agent's current city and its delivery city is where the action would take the agent to.

1.2 Implementation Details

- **Modular structure of the code:** We split the code into 3 main classes. *RoadAction* contains the action representation (see above), *State* contains the state representation (see above)This algorithm optimizes the value of a state which is the optimal recursive sum of rewards starting from that state. Both classes have automatically generated *hashCode* and *equals* functions needed for correct functionality of HashMaps. *Reactive* class contains *Setup* and *Act* Logist functions needed for the simulation. Q-table, V-table and Reward tables (implemented as hash maps) are all populated

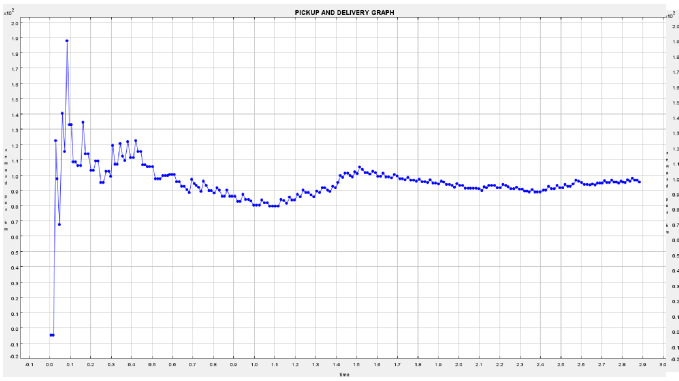

Figure 1: Discount factor of 0.1

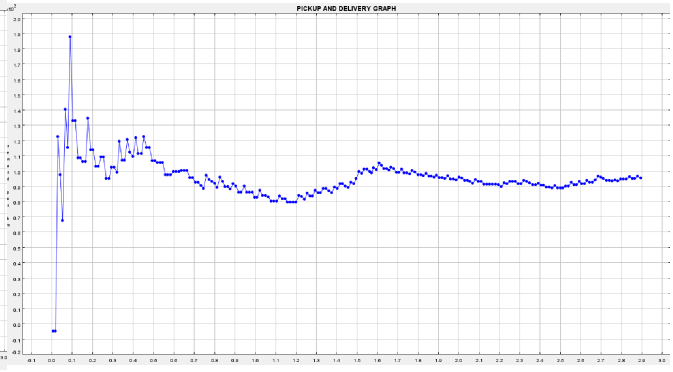

Figure 2: Discount factor of 0.6

based on our representation description in the *Setup* function. At the end of this function we run the RL algorithm.

- **Legal Action**: we define a *move* action as legal if the agent's state has no destination city (e.g. the agent has not already picked up a task) and it the action's intended city is neighboring to the one the agent is in at the moment. A pickup action is defined as legal if the agent's state has a destination city which matches the task's delivery city and the next state's current city.
- **Reinforcement Learning Algorithm** goes through all the states and all possible actions within given state to update the Q-table as follows, until **convergence**:

$$Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s')$$

- **Convergence** functionality goes through all elements in 2 consecutive V-tables, previous and updated, and returns *true* if the maximum difference between any two corresponding V-table entries is smaller than 0.001.

2 Results

2.1 Experiment 1: Discount factor

This experiment shows the importance of the discount factor γ used in the RL algorithm.

2.1.1 Setting

We used one RL agent in the France topology, using the seed of 3590420242192152425 for the task distribution.

2.1.2 Observations

We observe as the Discount factor gets closer to 1, we usually get better results. The reason behind this is the number of iterations to calculate the most optimal decision increases, because more weight is put on the value of the future state, not just the immediate reward. When the discount factor is close to 0, the agents main motivation is to go to choose an action with the highest immediate reward. The negative effect of this is of course the increased time needed to calculate the values.

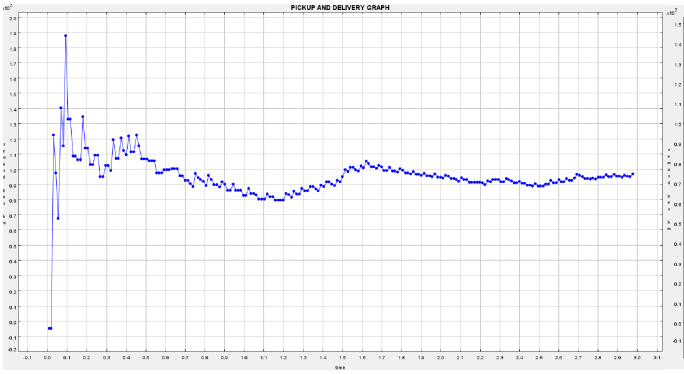

Figure 3: Discount factor of 0.85

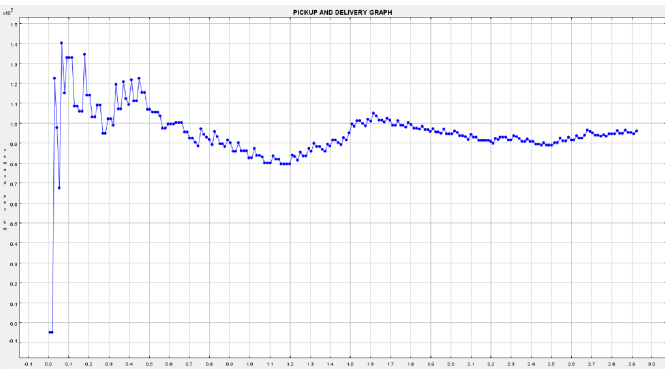

Figure 4: Discount factor of 0.99

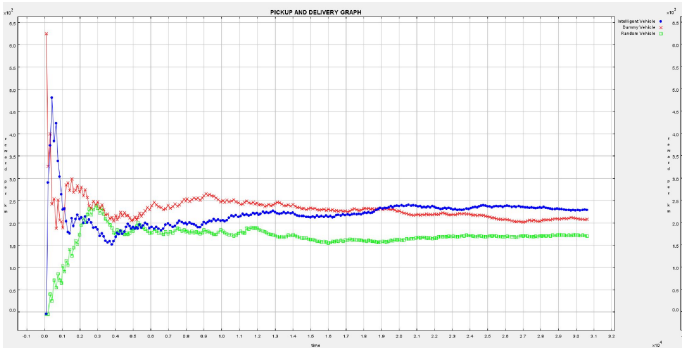

Figure 5: England

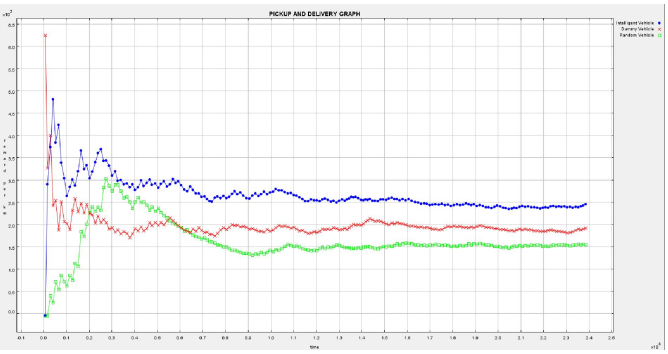

Figure 6: The Netherlands

2.2 Experiment 2: Comparisons with dummy agents

2.2.1 Setting

In addition to the provided random agent (green) we created a greedy agent (red) to compare against intelligent agent's (blue) performance. It checks if there is an available task for pickup the city it is currently located at and, if so, to pick it up. Otherwise, it moves to a random neighboring city.
The results shown in Figure 5 are from a simulation based on the provided topology of *England*. The intelligent agent started in London, the greedy agent in Leeds, and the random agent in Cardiff.
The results shown in Figure 6 are from a simulation based on the provided topology of *The Netherlands*. This time, all three agents started from the same city, Utrecht.
For both experiments, all three vehicles had the same cost per km (5) in order to not introduce an unfairness factor.

2.2.2 Observations

In both graphs we see some initial fluctuations in the vehicles' reward per km but those decrease with time and become more or less steady. The random agent tends to have the worst performance. During some simulations the intelligent agent takes the performance lead from the start, like in Figure 6. In those cases the tendency would persist. During other simulations we could see that the greedy agent started off better than the intelligent agent, such is the case with Figure 5. What we observed is that in, eventually, the intelligent agent would overtake the greedy one in terms of reward per km. This can be explained by the fact that while the greedy agent prioritizes immediate reward (if a task is available, pick it up and deliver it), the intelligent agent optimizes the value of a state which is the optimal recursive sum of rewards starting from that state. This implies that, even if the intelligent agent falls behind at the beginning, it makes up for it in the future and would perform better as time goes to infinity.