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

Group 30: Jinbu Liu, Mengjie Zhao

October 11, 2016

1 Implementation

This section aims to introduces the implementation details of the reactive agent for pickup and delivery problem which mainly contains three part: the implementation of reward maps, the implementation of the state transition map and training method of the policy. Before that, we will illustrate the representations of states and actions which are important for the following illustration.

1.1 Representation Description

In this section, we mainly illustrate the representation of states and actions. In current pickup and delivery problem, when a vehicle reaches at a city, the environmental information it obtained includes the current location, and whether there is a task in current city. The current state of the vehicle is decided by these information, so we create a VState class to represent the state of the vehicle whose data domain contains three part: a City type value represents the current city, a boolean type value indicates whether there is a task in the current city, and a City type value represents the delivery city of the task (the value is null if there is no task).

There are two types of action which can be carried out by a vehicle to reach to the next state, move and pickup. We create a VAction class to represent these actions. The data domain of VAction includes a Enum type value indicating the action type (move or pickup) and a City type value indicating the destination city (for move action) or the delivery city (for pickup action).

1.2 Implementation Details of Reward Map

Because the cost per kilometer maybe different among different vehicles, we initialize a reward for every vehicle. When a vehicle carries out an action on its current state, it gets a reward. So the reward mapping is from (state, action) to reward. We create a Tuple class to represent the tuple consisting a state and an action, we use a double value to represent the reward, and we use HashMap to store these mappings.

Now we need to calculate these rewards. There are two different case according to the type of action. The first case is that a vehicle moves to a neighbor city. In this case, the reward is the cost of moving from current city to the neighbor city. The second case is that there is a task in the current city and a vehicle pickups this task, and the reward will be the reward of the task minus the cost from current city to delivery city. According to above methods, we compute rewards of all possible combination of states and actions, and we store all these mappings from the tuple (state, action) to the reward into the reward map to get the final reward map.

1.3 Implementation Details of State Transition Map

Because the state transition is the same for all the vehicles, we just need one state transition map shared by vehicles. After carrying out an action on the current state, a vehicle's next state is decided by the next city and whether there is a task in the next city which is uncertain. So we need to calculate the probability distribution of the next state, and map (currentState, action, nextState) to a double value which indicates the probability of occurrence of the next state. For convenient, we create a Triplet class to represent a triplet consisting the current state, an action and the next state, and we use a HashMap to store these mappings.

The probability of the occurrence of the next state is only determined by the next state. There are two case according to whether there is a task in next city. In the first case, no task is generated in the next state, now the probability is the probability of the current city of the next state generates no task. In the second case, there is a task in next state generates, so the probability of next state is the probability of the current city of next state generates a task to the delivery city. According to above methods, we can calculate probabilities of all possible transitions, and we store all these mappings from the triplet (currentState, action, nextState) to the probability into the transition map to get the final transition map.

1.4 Implementation Details of Policy training process

Till to this moment, we already get reward maps and the state transition map. In order to use the training method studied in the class, we still to initialize a policy table. The policy table contains an optimal action, which leads to maximum profit, for every state, and we use a HashMap, containing mappings from the state to the optimal action, to represent the policy table. At the beginning, the optimal action for a state is initialized to be null. Now we also have the policy table, so can use the method showed in the class to train the policy table. We use a threshold to control the stop of the training process. In every training iteration, we save the overall improvement of the policy which is the maximum absolute value of improvement of the profit of different states. if the overall improvement is smaller than the threshold, the training process stops.

2 Results

2.1 Discount factor

Generally speaking, the discount factor reflects to what extent an agent taking care of rewards from future states. Smaller discount factor means that the agent focuses on rewards from a few upcoming states while a larger discount factor means the agent prefer to other policies that can generate high long-term rewards. The first effect of larger discount factor is that the training time for the policy will increase, since we need to consider long policies:

Table 1: γ and training time of a policy

γ	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
TrainTime (ms)	268	313	395	476	534	655	832	1280	2416	NotConverge

When evaluating the effects of discount factor, we created three agents with different discount factors: $\gamma = 0.1(blue), 0.6(read), 0.95(green)$ and controlling three vehicles receptively. The experimental results show that agents have higher discount factor obtains more rewards. We also noticed that the cost for moving of a vehicles results in different level of improvements. Following Figures 1 show our results when

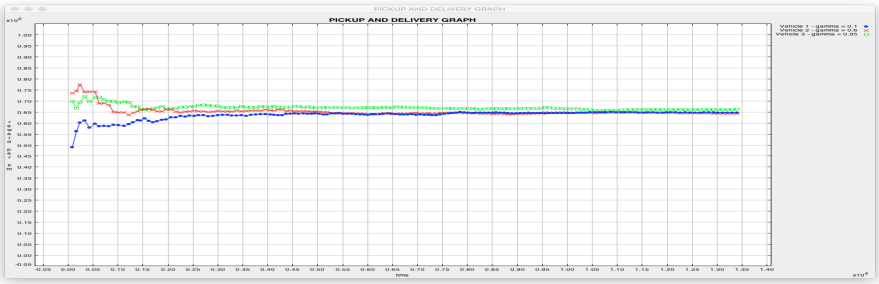

Figure 1: Rewards per km (cost = 5)

cost = 5: We can observe that larger discount factor results in higher reward per km. However, when *cost* = 5, the differences between the rewards are not significant (approximately 3). Following Figure 3 shows the rewards when *cost* = 30, We can observe that the effects of γ here is more significant (approximately 8). The reason is that higher the cost, higher the importance of good policies. In a higher cost scenario, the "greedy" policy trained with small γ shows its shortage more significantly.

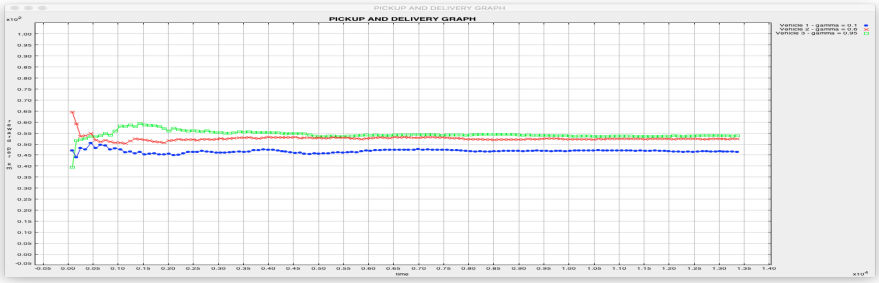

Figure 2: Rewards per km (cost = 30)

2.2 Comparisons with dummy agents

In this section, we compare our agent ($\gamma = 0.95$) with two dummy agents, which is trained with $\gamma = 0$ while the other agent has randomly movements. It can be easily observed that the agent with $\gamma = 0.95$

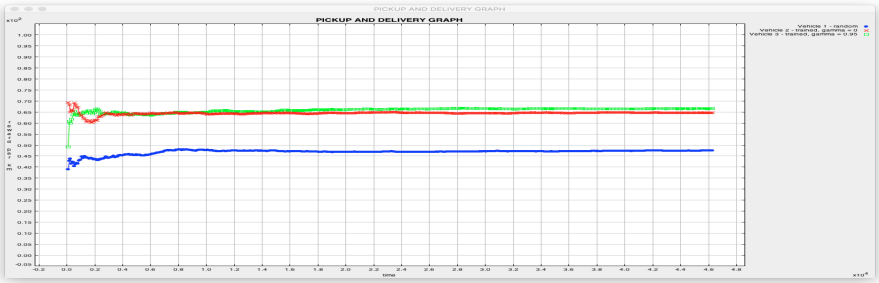

Figure 3: Rewards per km, 2 dummy agents and 1 trained agent (cost = 5)

(green) gives the highest rewards. The second highest reward is given by the agent with $\gamma = 0$ (red), which means that the agent always choose the first state to be the state that has highest reward and ignore future rewards. Hence the reward is still higher than the agent that has random behaviors (blue). Here the improvements are not significant, as the cost is relatively small.