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

Group №15: Christian Sciuto, Lorenzo Tarantino

October 10, 2017

## 1 Problem Representation

### 1.1 Representation Description

In order to understand how to represent the world of the problem, we started by considering the "act()" method of the Logist platform, where it is said that the method is called when the agent arrives in a city and is not carrying a task (it has just delivered it, or it had no task when it arrived in the city).

#### 1.1.1 State representation

Once the agent has arrived in the city X, it can only see whether a task is available or not, so its current state is described by (currentCity, destination city of the task). Obviously, the possible destination city of the task can actually be Null if there is no task available.

#### 1.1.2 Action representation

From the current state, the agent can change state through the "go to town Y" action. Each state is tied to a set of valid actions, this set contains the actions with the neighbour cities of the currentCity of the current state and the action with the Destination city of the task if it is different from Null.

#### 1.1.3 Transition function

Considering the current state s, the action 'a', the next state s' and that P (x, y) is the probability that there is a task from city x to city y, our transition function is such:

$$T(s, a, s') = P(a.nextCity, s'.taskDestination) \tag{1}$$

This is also true if the taskDestination of the state s' is null, since with P(x, null) is calculated the probability that there is no task in the city x.

#### 1.1.4 Reward function

Considering the current state s, the action 'a' and that reward(x, y) indicates the reward function that returns the average reward obtained by handing a task from city x to city y, the reward function is such:

$$R(s,a) = \begin{cases} reward(s.currentCity, a.nextCity) - s.currentCity.distanceTo(a.nextCity) * costPerKm(), & \text{if } s.taskDestination = a.nextCity \\ -s.currentCity.distanceTo(a.nextCity) * costPerKm(), & \text{if } s.taskDestination \neq a.nextCity \end{cases} \tag{2}$$

### 1.2 Implementation Details

We created the State class that contains the current city and the city of the task, and in addition there is a list of actions that are valid for the specific state.

The Action class simply contains a City attribute, the nextCity.

The section regarding the Reinforcement has been implemented in the Reinforcement class. Inside, in addition to the required attributes of the logist platform as the discount factor, the topology, the taskDistribution and agent, we also find two maps

```
private Map<State, Double> accumulatedValuesMap;
private Map<State, Action> bestActionPerStateMap;
```

The "accumulatedValuesMap" stores for each state the expected reward calculated with the V(s) function; the "bestActionPerStateMap" stores for each state the best action to choose in that specific state. Then, we implemented the methods to calculate the Reward function, the transition function and the Q function. The code is very simple and follows the specifications explained at 1.1 and what it has been said during the lecture. Let us therefore consider the valueIteration() method: the method follows the algorithm presented in the class, there is a first while-loop considering a stopping criterion, then a for-loop on the state list where two local variables are initialized

```
double currentV = accumulatedValuesMap.get(state);
double maxReward = Double.NEGATIVE_INFINITY;
```

The maxReward variable is used within the for-loop on the action list, if maxReward is less than the value returned by functionQ(state, action), then it is updated to that value and the "accumulatedValuesMap" and "bestActionPerStateMap" maps are also updated. At the end of each cycle iteration on the state list, a map named "tolerateErrorMap" is updated, which for each state 's' contains:

$$|V'(s) - V(s)|$$

At the end of the loop on the state list, the "stoppingCriterionIsVerified" method is called, which verifies whether the "valueIteration" method can terminate.

In the ReactiveAgent class we put a Reinforcement type attribute, then we called the valueIteration() method in the setup() method, and finally in the act() method we took the best action for the current state (the current state was obtained from arguments assigned to the act method).

## 2 Results

### 2.1 Experiment 1: Discount factor

The purpose of this experiment is to understand how the discount factor influences the result.

#### 2.1.1 Setting

We used the default configuration files, in the "agents.xml" file we changed the class-name in order to use our agent with the reinforcement algorithm and the discount factor. These are the discount factor that we used: 0.000001, 0.50, 0.99. the simSpeed is equal to 100.

#### 2.1.2 Observations

Discount factor = 0.000001: after 1500 ticks the reward per Km is around 0.6, then after 50 thousand ticks it stabilizes around 0.64.

Discount factor = 0.50: after 1500 ticks reward per Km touches is maximum at 0.67, then it is stable around 0.65.

Discount factor = 0.99: after 1500 ticks the reward per Km touches is maximum at 0.7, then after 50 thousand ticks it is around 0.68

#### 2.1.3 Conclusion

Through these experiments, we can observe that as we increase the discount factor, the asymptotic reward increases as well.

### 2.2 Experiment 2: Comparisons with dummy agents

In this experiment we have compared our Reactive Agent with other two agents: the first one is the agent given in the starter files that chooses an action randomly, the second one is an agent that chooses the worst action possible (the action that will give the agent the lowest reward in that state)

#### 2.2.1 Setting

For each agent we used the default configuration files with a discount factor of 0.85, the simSpeed is equal to 100.

#### 2.2.2 Observations

1) Reactive Agent: the reward per Km is at its maximum of 0.68 around 1500 ticks, then after 50 thousand ticks it stabilizes at 0.675.

2) Random Agent: the reward per Km is at its maximum of 0.55 around 1000 ticks. After 1500 ticks the reward per Km is around 0.52, then after 50 thousand ticks it stabilizes at 0.48.

3) WorstAction Agent: the reward per Km is at its maximum of 0.36 around 1500 ticks, then after 50 thousand ticks it stabilizes at 0.38.

#### 2.2.3 Conclusion

As we expected, we saw the best performances with the smart reactive agent. We can observe that the random agent reaches the average reward between the smart agent and the worst case agent. Clearly, the worst agent has the worst performances since he always chooses the action with the minimum reward.