Excercise 2.
A Reactive Agent for the Pickup and Delivery Problem.

Group 21: Kristoffer Landsnes, Martin Vold

October 9, 2018

1 Problem Representation

1.1 Representation Description

A reactive agent has been implemented on the basis of the travelling salesman problem. Through reinforcement learning, specifically the value iteration algorithm, it's desired to create a reactive agent which behaves optimally for picking up tasks. Thus, a large part of the problem-solving is to define the state of world and different actions a reactive agent execute in the network of cities. After training the agent offline it will be launched in the Logist platform acting with the optimal trained strategy.

The state contains a variable for the city the agent is currently in, $fromCity$, and the city the task is to be delivered to, $deliveryTo$, if there is any task here. It was decided on representing a state with the $fromCity$ and $deliveryTo$ variables. If $deliveryTo$ is null there is no task available in the city the agent currently is in. From a state with $fromCity$ and $deliveryTo$ only some actions can be executed. Pick up the task and deliver it to the $deliveryTo$ city, which demands that $deliveryTo$ is not null, or move to one of the neighboring cities of $fromCity$. As for the reward table it is the value of the delivery, if $deliveryTo$ is not null, minus the distance between distance travelled from one city to another times the cost per km for the agent. For the probability transition table we used the function $TaskDistribution.probability(from, to)$ function. Here the probability for a city to have no task is 1 minus the sum of probabilities from the city the agent is in to all other cities in the topology.

1.2 Implementation Details

1.2.1 Representation

The state class is represented as shown below:

```
public class State {
     public City fromCity; // the city we are in
     public City deliveryTo; // city the task is to be delivered to
     public HashMap<City, Double> qValues = new HashMap<City, Double>(); // Q(s,a)
     public HashMap<City, Double> rewards = new HashMap<City, Double>(); //R(s,a)
```

The State class initially sets the current city for the state (fromCity), the destination city for any possibly picked up task (deliveryTo), and a Q-table formulated as $publicHashMap < City, Double > qValues = newHashMap < City, Double > ();$ (initially all values set to null) and a reward-table, $publicHashMap < City, Double > rewards = newHashMap < City, Double > ();$. The reward-table is calculated as described above.
To store all these states we used a HashMap of HashMaps. It looked like this $states = newHashMap < City, HashMap < City, State >> ();$. Each of the first cites, $fromCity$, has it's own HashMap where

the next city, *deliveryTo* is one of all possible cities the task could have as it's delivery city, including the *null* task.
The best value and best action for each state is stored in their own public HashMaps, where the key is the state and the value is either a Double (best value) or City object (best action).

1.2.2 Implementation of Value Iteration

The value iteration algorithm is implemented as it was described in the task description. The stages of the algorithm is:

- Firstly, the value iteration implementation iterates over all the states.
- For each state we iterate over all the different actions the agent could do in this state and uses the bellman update equation to update the q-value corresponding to this state-action pair.
- After the q-values are updated we update the best value and the best action for this state.
- After this is done for all states we check if the highest absolute value of the new best value minus the old best value of all the states is above or below a set limit, we used 1/1000000000 as this limit. If it's below we are done with the value iteration algorithm and if it's above we continue.

For us not to consider an action that will make us move to the same city that we are currently in, we have made the q-values HashMap to not contain this action, by adding move actions to only neighboring cities and the delivery city of the task, if it's not *null*.

1.3 Choosing an action

To choose a action to do we fist check to see if there is an available task. If there isn't we move to the best neighboring city, best action for this state. If there is a task, we must decide if we want to take it. We take the task if the best action for the state is the same as the delivery city of the task. If it's not we go to the city that is the best action of the state.

2 Results

2.1 Experiment 1 - Discount factor

2.1.1 Setting

For this experiment we have added some more agents to the agents.xml file in the config folder. This is so we can run the simulation with multiple reactive-rla agents. For this experiment the program arguments where as follows: $config/reactive.xml reactive-rla-0.99 reactive-rla-0.8 reactive-rla-0.5 reactive-rla-0.01$.

2.1.2 Observations

We let the simulation run for a long time. Each agent got to do approximately 1500 actions and the result we ended up with was:

Discount-factor	**Iterations**	**Average profit**
0.99	21	38375.69
0.8	19	37254.79
0.5	14	37854.91
0.01	6	37629.64

Firstly, when running the value iteration with different discount factor's we observe that all of the agents converges to same value after around 1500 actions. The interesting part is how little the performance of the reactive agent differs depending on the discount-factor. It's a clear difference when using $\gamma = 0.01$ and the remaining ones with $\gamma \geq 0.5$. After reaching the "good enough" area, the discount factor doesn't really affect the performance remarkably. Also, when the discount-factor increases we will need more iterations to converge. As the algorithm weights the future more than necessary.

2.2 Experiment 2: Comparisons with dummy agents

The experiment used the random agent given in the starter file, another dummy agent always picking up a task and our reactive agent.

2.2.1 Setting

The agent added, called the always pick-up agent is doing exactly what its name implies, picking up every task. For this experiment the program arguments where as follows: $config/reactive.xmlreactive-rla-0.99reactive-randomreactive-allwaysPickUp$.

2.2.2 Observations

- **Reactive agent with a discount factor of 0.99**: Average profit of 36906,37.
- **Random agent with a 0.85 chance of picking up a task**: Average profit of 32241,72.
- **Random agent with a 1.0 chance of picking up a task** Average profit of 35495,75.

The performance of the different agent's can be observed in figure 1, with the different agents corresponding to the label. It is observed the agent always picking up tasks is outperforming the random agent form this experiment. Furthermore, it is obvious that the reactive agent is far better over time, which was expected.

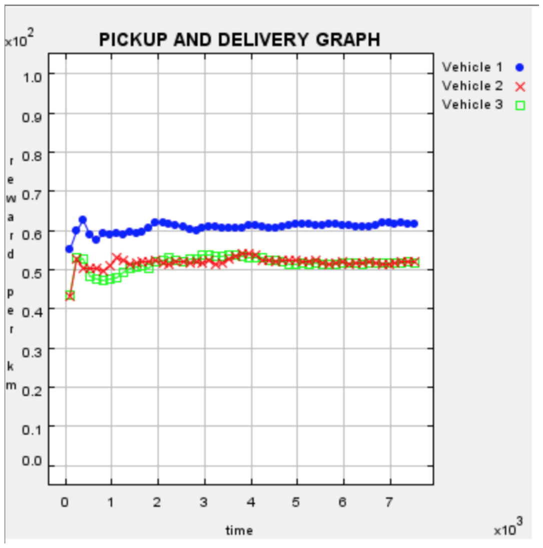

Figure 1: Multiagents plot. Blue = reactive. Red = random. Green = pick-up all.