

## CS440 MP2 – Due 12/8 (Last day of class)

### Overview

The objective of this homework is to implement and test a reinforcement-learning agent. The language java should be used for this assignment. Since you will work in groups of 4-5, you should make sure someone in your group is familiar with java.

The submission should be made with a zip or tar.gz file to [ta440@cs.uiuc.edu](mailto:ta440@cs.uiuc.edu) containing your java source code, the class file for your code and your report in PDF or doc (or docx).

The assignment is divided in two parts:

#### Part I

- Implement a Q-learning agent
- Test your program

#### Part II

- Run various experiments
- Report and explain the results

### The Problem

Your friend John has the very questionable habit of dating two girls at the same time. So far, this has worked well for him as long as the two girls never meet. However, recently both girls moved to John's neighborhood and he is afraid his girlfriends will eventually learn about each other.

To avoid complications in case this happens, your friend has decided to implement an aggressive over-compensation strategy to make both girls happy. Your friend plans to deliver gifts regularly to both girlfriends. However, since your friend cannot do the delivery himself, he decided to create a robot to do so. Although you strongly disagree with your friend's conduct you lost a bet and now you have to help him program the delivery robot using reinforcement learning.

	X	G1		X
X			T	
			X	G2
		J		

The 5x5 grid above describes John's neighborhood. G1 and G2 represent the houses of each of John's girlfriends and J represent's John's house. The robot cannot occupy the positions marked with X in the grid. Also, since the robot is far from perfect all

actions have a random failure probability, in which case the robot finds itself in any valid adjacent position.

Your robot should also be aware of the presence of thief in the neighborhood. The thief is marked as T in the grid but he may move. If he spots the robot he will decide to follow it and try to steal the gifts. However, the Thief is rather incompetent and although it always behaves according to the same policy (no hidden states) we cannot assume that he behaves optimally.

The robot can carry up to two gifts at a given time. It should deliver one gift for each girlfriend and return to John's house to reload.

Your task is to use reinforcement learning to learn an optimal policy for this problem. The complete state description consists of the position of the robot, the position of the thief, the delivery status for each girlfriend and the number of gifts the robot is currently carrying. The total number of states is therefore 7500. (25 possible positions for the robot, 25 possible positions for the Thief, 3 possible number of gifts, 2 possible delivery statuses for G1 and for G2 ->  $25 * 25 * 3 * 2 * 2 = 7500$ ). Note that not all of these states will be observed during simulations (For example, the robot may not occupy some of the positions).

In every state the robot can execute 4 actions: moving North (0), South (1), East (2) or West (3). If the action leads to an impossible location (such as the surrounding walls or those marked X) then you will remain in the same location in the next state.

## Simulation

We provide most of the necessary code to run the simulation. Your main task is to implement a Java class that implements the Agent interface:

```
interface Agent {  
    public boolean reset();  
    public int getAction(double reward, State s);  
}
```

The `getAction` method will be invoked by the simulator at each step of the simulation. The first argument, `reward` is the reward received after executing the previous action, and `s` is the current state (i.e. state reached after executing the previous action). The method `getAction` should return an integer, which corresponds to the next action to take (in state `s`).

The State object has one important method, `getID()`. It returns a non-negative integer, which is unique for each state. You may retrieve the individual state variables by instantiating a `GiftState` object with the state ID. `GiftState` is defined below:

```

class GiftState {

    public int getID()
    public int getAgent_X()
    public int getAgent_Y()
    public int getThief_X()
    public int getThief_Y()
    public int getGift1_Status()
    public int getGift2_Status()
    public int getNumberOfGifts()

}

```

Note that not all of these variables are defined in all scenarios (see below). The maximum number of possible states is given by the constant **GiftWorld.NUM\_STATE** (all state IDs are guaranteed to be smaller than this number). The number of possible actions is given by the constant **GiftWorld.NUM\_ACTION**, and the actions are numbered **0, 1, . . . , GiftWorld.NUM\_ACTION - 1**.

At the end of each training episode (i.e. after the maximum number of steps in the episode is reached), the method `reset` will be invoked. The return value of `reset` should be true to start a new episode (restart from an initial state), otherwise the simulation will end.

We provide 3 different world scenarios, described below:

- **GiftWorld1** There is no Thief in the neighborhood
- **GiftWorld2** The Thief is there and you know his location
- **GiftWorld3** The Thief is “invisible” to you (but will affect your actions)

Each world has 2 possible reward functions:

- **R1** There is a positive reward when you make each delivery
- **R2** In addition to the positive reward at each delivery, there is a positive reward when you get the set of Gifts from the campaign headquarters, and there is a negative reward when the Thief steals a gift.

Each of this combination will be named **GiftWorld1\_R1, GiftWorld1\_R2**, etc.

## Running the Simulation

To test your agent (after successfully compiling your code), run the following command:

```
java Sim <world> <agent> <steps> <episodes> <display> <output> [<params> ...]
```

where `<world>` is the class name of the world (the world classes are named GiftWorld1\_R1, GiftWorld3\_R2, etc.), `<agent>` is the class name of the agent, `<steps>` is the maximum number of steps to take in each episode, and `<episodes>` is the maximum number of training episodes to run (0 for infinity—only stops when your agent returns false during reset). `<display>` is either 0 or 1. When it is set to 1, each state visited will be printed to the standard output (screen), for testing purpose. `<output>` should be the name of a file where the output is sent. The output will be a 4-column text file (separated by a space), where the columns contain the episode number, the number of deliveries made in the episode, the number of Gifts stolen in the episode, and the number of Gift set refills made, respectively. An episode will terminate when `<steps>` steps are performed in the simulation.

[`<params> ...`] is an optional list of parameters. The entire list of parameters (including the first 6) is stored in a static String array `Sim.ARGs`. So, `Sim.ARGs[6]` will be the first parameter in [`<params> ...`]. You can use this to set any input parameters that your agent might need.

Example: To run GiftWorld1 R1 with an agent named QAgent1 for 50000 episodes, with 1000 steps in each episode, no display, and output file out.txt,

```
java Sim GiftWorld1 R1 QAgent1 1000 50000 0 out.txt
```

## Saving and Testing a Policy

We have provided a special `FileAgent` to help you evaluate your learned policies. To use it, you need to first save a `Policy` object to a file. We have defined a `Policy` class as follows:

```
class Policy {
    public Policy(int [] actions, double [] utilities);
    public boolean save(String filename);
}
```

To save a policy, you need to instantiate a `Policy` object by providing two arrays, where the first array contains the action to perform in each state ID (e.g. `actions[34]` will be 0 if action 0 is to be performed in state 34). The learned expected utility of each state has to be specified with the utilities array.

In this MP the reward is allowed to depend on the current state, the action performed, as well as the next state reached. Note that this is more general than

either the treatment in the text or in class. We use the following definition for the optimal expected utility:

$$U^*(s) = \max_a Q^*(s, a),$$

where:

$$Q^*(s, a) = E_{\text{Pr}(s'|s,a)} \left[ R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right]$$

where E denotes taking an expectation of (i.e., averaging) the angle-bracketed argument over the subscripted distribution.

The save method of the Policy object will save the Policy to the specified file. You can then run the simulation using the FileAgent, and when asked, enter the name of the saved Policy file.

There is also a KeyAgent where you may manually control the actions of the agent. This allows you to explore the world dynamics directly.

## Test and Report

You need to write code to collect data and answer the following questions:

### PART I

1. Implement a Q-learning agent with the following properties:

(a) Use a constant learning step size of 0.02.

(b) Use a discount factor of 0.99.

(c) Initialize all Q-values to zero.

(d) By “greedy” we mean always choose the action with the highest Q-value, and if there is more than one such action, choose randomly among them.

2. Test your agent:

(a) On GiftWorld1 R1 with a greedy strategy.

(b) On GiftWorld1 R2 with  $\epsilon$ -greedy strategy with exploration using  $\epsilon = 0.05$ .

3. Save the learned policy (as described in the previous section) in each scenario.

4. Hand in your Java code and the learned policies. Include sufficient information about your implementation so that we can easily understand what you did.

## PART II

5. Run each of these experiments on the specified world with the specified exploration strategy. Simulate 1000 steps per episode.

(a) GiftWorld2 R1 with a greedy strategy.

(b) GiftWorld2 R1 with an  $\epsilon$ -greedy exploration strategy using  $\epsilon = 0.05$ .

(c) GiftWorld2 R2 with a greedy strategy.

(d) GiftWorld2 R2 with an  $\epsilon$ -greedy exploration strategy using  $\epsilon = 0.05$ .

6. Present the policies learned in problem 5 in an appropriate graphical format (as in Figure 17.2 of the textbook). Note that depending on the world, you may need one such grid for each position of the Thief, each delivery status, and each number of Gifts you have.

You do not have to show the policies for all possible Thief positions; instead, pick a few representative Thief positions and show the policies (you should pick at least one position near a girlfriend's house, one position near Johns house, and one other position). You also need to include the saved policy files in your submission.

7. Plot graphs that show the number of deliveries made in each episode. What trend do you see from the graphs?

8. Compare experiments in problem 5. Which method converges faster? How do the learned policies differ between the methods? Explain your findings.

9. GiftWorld3 is an example of a partially observable environment. Run GiftWorld3\_R2 with an  $\epsilon$ -greedy exploration strategy using  $\epsilon = 0.05$ . Compare the results with the fully-observable case (GiftWorld2\_R2). Explain the difference (if any).

10. We want to expand the size of the map to  $25 \times 25$  and include 10 girlfriends and 5 Thieves in the neighborhood, expanding the capacity of the robot to 10 gifts at a time. The combined state space will be extremely large. How large would it be and how would you address this problem? (You do not need to implement or run this).