Project 3 (CSCI 360, Spring 2018)

10 Points

Due date: April 30th, 2018, 11:59PM PST

1 Introduction

In this project, you will use the tools of probability theory, decision theory, and reinforcement learning to compute optimal *policies* that Wheelbot can follow to reach target locations in its environment (while avoiding obstacles) in the presence of actuation noise. To simplify this project, sensor noise will not be considered, and Wheelbot will know the locations of all the obstacles in its environment as well as the target location. Environment boundaries and obstacles will again act as barriers that prevent Wheelbot's movement.

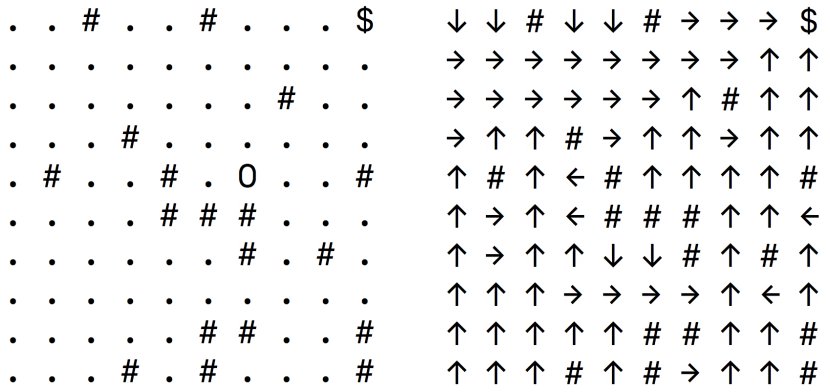

Figure 1: A screenshot of the text-based simulator for project 3 (left) and a visualization of the optimal policy for navigating to the goal location ($) in this environment, computed using the value iteration algorithm (right). Note how the actions at each state lead Wheelbot away from obstacles and toward the goal location.

2 Simulator

2.1 Wheelbot

For this assignment, Wheelbot has the following actions and sensors:

2.1.1 Wheelbot Actions

1. **MOVE_UP**: Moves Wheelbot one grid space up (North)
2. **MOVE_DOWN**: Moves Wheelbot one grid space down (South)
3. **MOVE_LEFT**: Moves Wheelbot one grid space left (West)
4. **MOVE_RIGHT**: Moves Wheelbot one grid space right (East)

2.1.2 Wheelbot Sensors

1. **Perfect obstacle sensor**: this→simulator.getObstacleLocations() returns a vector of obstacle locations in the environment as Point2Ds.
2. **Perfect target sensor**: this→simulator.getTarget() returns a Point2D representing the current target location in the environment.

2.1.3 Other functionality

The following important functionality is required for this project:

1. In the MDP class, the dimension n of the $n \times n$ environment grid can be accessed using the *gridDimension* member variable.
2. In the MDP class, the number of grid cells, $gridDimension * gridDimension$, can be accessed using the member variable N.
3. In the MDP class, the member variable *actions* provides a list of the actions available to Wheelbot.
4. In the MDP class, the member variable *policy*, which maps from Point2D instances to RobotActions is a container in which to store the optimal policy computed by the value iteration algorithm.
5. In the MDP class, the member variable Q, which maps from Point2D-RobotAction pairs to double precision floating point numbers, is a container in which to store the Q function learned by Q-learning.

6. In the MDP class, the function getRewardForTransition(RobotAction a, Point2D current, Point2D next) provides the (double precision) reward associated with taking an action a and transitioning from state *current* to state *next*.

7. In the MDP class, the function getProbabilityOfTransition(RobotAction a, Point2D current, Point2D next) provides the probability of transitioning from *current* to *next* under action a.

8. In the MDP class, the function printVIPolicy() prints a visualization of the policy computed by value iteration (see Figure 1).

9. In the MDP class, the function printQPolicy() prints a visualization of the policy computed by Q-learning.

10. In main.cpp, *alpha* controls the probability with which Wheelbot executes the correct action (transition noise parameter); *numEpisodes* controls the number of episodes used to learn the Q function in Q-learning; *numEvalEpisodes* controls the number of episodes used to evaluate value iteration or Q-learning (after the optimal policy has been learned); *gamma* is the discount factor; *learningRate* is the Q-learning learning rate; *valItErrorThreshold* controls the allowable error in the value iteration algorithm (convergence threshold).

11. In main.cpp, *epsilon* controls the probability with which Wheelbot executes random actions during Q-learning. **Note that this definition of ϵ is different from Project 2**.

12. In main.cpp, *mode* controls whether to run the $VALUE\_ITERATION$ or $QLEARNING$ validation procedure.

3 Programming Portion

3.1 Theory

A *policy* is a mapping from environment states to actions. It tells the agent what action to perform in each possible state it could end up in. In general, policies may specify probability distributions over possible actions at each time step, but, for the purposes of this project, we will consider a policy to be a function $\pi : X \rightarrow A$ that maps grid cell states $x \in X$ to Wheelbot actions $a \in A$. *Reinforcement learning* (RL) is an area of machine learning concerned with computing *optimal policies*, where optimality is defined

as the *expected discounted sum of future rewards*. *Rewards* are (generally human-designed) scalar signals given to the agent throughout its interactions with its environment that punish bad behavior and reward good behavior such that the agent learns to behave optimally over time. (Many RL techniques are also concerned with learning the transition and observation probability distributions you computed in the previous project in unknown environments, but these techniques are beyond the scope of this project.)

In this project, Wheelbot's environment can be defined as a *Markov Decision Process* (MDP). An MDP is a 5-tuple (X, A, T, R, γ), where X is a set of states (grid cells, in this case), A is a set of agent actions ({MOVE_UP, MOVE_DOWN, MOVE_LEFT, MOVE_RIGHT}, in this case), T is the set of transition distributions $P(x'|x, a)$ for each $x, a, x' \in X \times A \times X$, $R : X \times A \times X \to \mathbb{R}$ is a reward function mapping transition triples x, a, x' to scalar rewards, and γ is the reinforcement learning discount factor (i.e., how much future rewards should be discounted to ensure the expected discounted sum is finite over an infinite horizon, see AIMA Chapter 17 for more details on this technical condition).

In this project, Wheelbot knows X, A, T, R and γ, but must *learn* an optimal policy $\pi : X \to A$ that maps each grid cell $x \in X$ to an *optimal* action relative to reward function R, which is designed, in this project, to navigate Wheelbot to a goal location while avoiding obstacles. The reward function R has the following simple definition:

1. Transitions into the target (goal) grid cell are given a reward of $+100$.
2. Transitions from a grid cell back to itself are punished with a reward of -10 (because this indicates we hit an obstacle or grid boundary).
3. All other transitions are punished with a reward of -0.04, such that Wheelbot wants to escape to the goal as quickly as possible.

This reward function has been implemented for you in the function getRewardForTransition() in the MDP class. Your task will be to *learn* an optimal policy in this MDP environment (using both *Value Iteration* and *Q-learning*) that enables Wheelbot to optimally navigate itself to a goal location while avoiding obstacles, even in the face of transition noise.

3.1.1 Value Iteration

Value Iteration is an iterative method for learning an optimal value (or utility) function $V : X \to \mathbb{R}$, which maps each state in the agent's state space to the optimal utility of that state (i.e., the expected discounted sum of

future rewards attainable from that state by following an optimal policy). This value function can then be used to select the actions that are optimal in each state of the environment. The algorithm operates by iteratively solving a set of *Bellman* equations (one for each state) until convergence, meaning that the maximum change in the utility of any state in some algorithm iteration is below some user-defined threshold. See the pseudocode in Figure 2.

```
1: Procedure Value_Iteration(S,A,P,R,θ)
2:         Inputs
3:                 S is the set of all states
4:                 A is the set of all actions
5:                 P is state transition function specifying P(s'|s,a)
6:                 R is a reward function R(s,a,s')
7:                 θ a threshold, θ>0
8:         Output
9:                 π[S] approximately optimal policy
10:                V[S] value function
11:        Local
12:                real array V_k[S] is a sequence of value functions
13:                action array π[S]
14:        assign V_0[S] arbitrarily
15:        k ←0
16:        repeat
17:                k ←k+1
18:                for each state s do
19:                        V_k[s] = max_a Σ_s' P(s'|s,a) (R(s,a,s')+ γV_{k-1}[s'])
20:        until ∀s |V_k[s]−V_{k-1}[s]| < θ
21:        for each state s do
22:                π[s] = argmax_a Σ_s' P(s'|s,a) (R(s,a,s')+ γV_k[s'])
23:        return π,V_k
```

Figure 2: Pseudocode for value iteration.

The main Bellman update is performed in line 19 for each state until the maximum absolute value change of any state at two consecutive iterations is lower than some threshold θ (denoted $valItThreshold$ in MDP.cpp). After convergence, lines $21-22$ compute the optimal action at each state and store it in π (the member variable *policy* in MDP.cpp). Since the environment is known, Wheelbot need only perform value iteration once at the beginning of the simulation. Wheelbot can then use the learned policy to perform optimally in its environment regardless of its start state. Once the environment or the target location changes, Wheelbot will need to perform value iteration again to learn a new optimal policy. Chapter 17 of AIMA provides more information about value iteration.

3.1.2 Q Learning

How could Wheelbot hope to learn an optimal policy π in an environment with unknown dynamics (T)? Q-learning can be used to solve this problem. Q-learning is a *model-free* RL technique, which means that it can be used to learn an optimal policy without requiring the environment dynamics T to be known or even learned by Wheelbot. It does this by actively exploring its environment to estimate a function $Q : X \times A \to \mathbb{R}$ that gives the optimal utility of executing action $a \in A$ in state $x \in X$ for all state-action pairs in $X \times A$. At each time step t, the agent makes a transition from state x_t to x_{t+1} using action a_t. The agent receives a reward r_t associated with that transition. The agent then updates its Q table as follows:

$$Q(x_t, a_t) \leftarrow (1 - \alpha)Q(x_t, a_t) + \alpha(r_t + \gamma \max_a Q(x_{t+1}, a)) \tag{1}$$

α is called the learning rate (denoted *learningRate* in main.cpp and MDP.cpp). Given enough learning *episodes*, each of which ends when Wheelbot reaches the target grid cell, this Q function will converge to the true estimates of the optimal utilities of executing each action a in each state x. In practice, we simply allow Wheelbot to experience many episodes, and extract the policy after this training period as follows:

$$\pi(x) = \arg\max_a Q(x, a) \tag{2}$$

Q-learning is called an *off-policy* learning algorithm because it learns the optimal policy relative to the given reward function by following a different policy during learning, such as an *ϵ-greedy policy*. An ϵ-greedy policy executes the optimal action relative to the current estimated Q function (equation 2) with probability $1 - \epsilon$. With (some usually small) probability ϵ, the agent executes a random action. This allows the agent to *explore* its environment (with probability ϵ), hopefully to eventually find more reward, while simultaneously *exploiting* its environment (with probability $1 - \epsilon$) to get as much reward as possible given what it currently knows. Such a policy is already programmed for you in main.cpp, where $epsilon = 0.08$. For more information on Q-learning, see chapter 21 of AIMA.

3.2 Implementation

Your task, in this project, is to implement **Value Iteration** and **Q-learning** in the file MDP.cpp (which defines the MDP class). Specifically, you should complete the following functions:

1. *void valueIteration(double valItThreshold, double gamma)*: This function takes in $valItThreshold$ (equivalent to θ in Figure 2) and *gamma* (equivalent to discount factor γ in Figure 2). It should perform the Value Iteration procedure of Figure 2 and populate the *policy* member variable of the MDP class with the optimal action a corresponding to each grid cell x in Wheelbot's environment, relative to the reward function defined in MDP.cpp by the function getRewardForTransition(), which is already filled in for you.

2. *void QLearningUpdate(RobotAction a, Point2D current, Point2D next, double gamma, double learningRate)*: This function takes in an observed transition in the environment from *current* to *next* under the action a. The discount factor *gamma* and the learning rate $learningRate$ are also passed in. This function should update $Q[current][a]$ according to the Q-value update of equation 1.

3. *RobotAction getOptimalVIAction(Point2D state)*: This function receives a grid cell *state* as input and should compute the optimal action to take in that grid cell according to the policy computed by Value Iteration.

4. *RobotAction getOptimalQLearningAction(Point2D state)*: This function receives a grid cell *state* as input and should compute the optimal action to take in that grid cell according to the policy computed by Q Learning (equation 2).

3.3 Validation and Submission

To run the value iteration validation procedure, set mode = VALUE_ITERATION in main.cpp. This will invoke your valueIteration() procedure to compute an optimal policy and then execute that policy $numEvalEpisodes$ times from different random starting positions in the environment. To run the Q-learning validation procedure, set $mode = QLEARNING$ in main.cpp. This will train the Q-learning agent on $numEpisodes$ episodes of the same environment (from randomized starting positions). After this training period, the optimal policy will be extracted from the estimated Q-function and evaluated $numEvalEpisodes$ times. One key way to evaluate your RL learning procedures is to set *alpha* to 1.0 and ensure Wheelbot follows a shortest path to the goal location. For this project, please submit a zip archive of your modified code via Blackboard – MDP.h/MDP.cpp and any files you add – by the date and time listed at the top of this document.