Assignment 5
Markov Decision Processes
Part 1: Value Iteration
CS 3420 – Spring, 2016
Due: Wednesday, 4 May 2016

In this three-part assignment, you will solve a somewhat more complicated version of the simple gridworld problem I presented in class using three different solution techniques:

Part 1: Value Iteration,

Part 2: Policy Iteration, and

Part 3: Q-Learning

I highly recommend that you work in groups of two or three.

The Problem

In this gridworld, the agent always starts from the square marked `Start` (please see the figure on the last page). There are four available actions: `Go-North`, `Go-East`, `Go-South`, `Go-West`. They are probabilistic in the same way as the actions in the maze problem in the book. With probability 0.8, the agent goes in the intended direction, with probability 0.1 it goes left instead, and with probability 0.1 it goes right instead. In all cases, if it hits a wall it remains in the square it started in. Once it enters the square marked `+1` or either of the two squares marked `-1`, no further movements are possible. This can be modeled by making all the transition probabilities from these squares 0.0 (in spite of the fact that this violates the constraint that transition probabilities from a state should sum to 1.0).

Rewards are specified as a function of the state the agent is in. There is a reward (step cost) of -0.04 associated with every state with the following exceptions. The reward associated with a state in which the agent occupies the square marked `+1` *and* has a `Key` (see below), is +1. The reward associated with a state in which the agent occupies the square marked `-1` is -1. All rewards should be changeable via command line arguments (more on this below).

As noted in the previous paragraph, the agent must have a `Key` to receive the +1 reward (the reward is behind a locked door). Note that the agent need only enter the `+1` square with a `Key` in its possession to obtain the reward—no "unlock" action is necessary. To obtain a `Key`, the agent must visit the square marked `Key`. As soon as the agent makes a transition that places it in the `Key` square, it obtains a `Key`—no explicit "get-key" action is necessary. You should assume that there is a never-ending supply of `Key`s in this square so that the agent can go back and get another one if it loses the

Key in its possession, which will happen with probability 0.5 (also changeable on the command line) if it enters the Lose-Key? square.

Intuitively (and correctly in most instances) the optimal policy will have the agent obtain a Key by visiting the Key space and obtain the positive reward by visiting the +1 space with the Key. Notice that the possible paths to the Key square (and from the Key square to the +1 square) fall into three main categories:

1. The most direct path takes the agent between the two -1 squares, both to and from the Key square, even though there is a risk of being trapped in one of the -1 squares.

2. On the most indirect paths, the agent completely avoids the -1 squares, but it must go through the Lose-Key? square, where it will lose any Key it has with probability 0.5.

3. There are "in-between" paths that pass through the square between the Lose-Key? square and the left-most -1 square. Taking a path of this type lessens the agent's risk of losing a Key, but exposes it to one of the -1 squares.

Provided Code

I have provided an MDP class that implements a number of things for you, so you can focus on implementing the solution techniques. The code declares some constants and global variables, some of which I will discuss below. All of them are documented. I have also written:

- a printUtilitiesAndPolicy method that prints out the utilities of each state and the policy, and
- an initializeMDP method that initializes T, a 3-dimensional array that contains the state transition probabilities, and R, a 1-dimensional array containing the rewards for each state.

Feel feel to discard any pieces of (or all of) this code, as long as your code operates in the way I have specified. In particular, your code must print out the utilities and policy in exactly the same way I have done.

I have described the characteristics of a state (location and key possession). Note that the state of the agent must specify both its location *and* whether or not it has a Key. Thus, with the exception of the Key square, there are two states associated with each square—a state in which the agent is at that square and has the Key and a state in which the agent is at that square and does not have the Key. Of course, the agent cannot be in the Key square without having the Key, so there are 65 states.

I have specified a numbering scheme for the states (0-64) that I would like you to use, since the method I have given you that prints out the utilities and policy depends

on this numbering. To see the numbering, compile and execute the code I have given you. For each square in the grid you will see something that looks like the following (this is the left-most square in the top row):

```
(58) 0.0 (N)
(59) 0.0 (N)
```

This means that state 58 is the state in which the agent is at that location and *has* a `Key` and state 59 is the state in which the agent is at that location and *does not have* a `Key`. The numbers are the utilities associated with those states and the letter indicates the action specified by the policy.

I have specified the action set (`Go-North`, `Go-East`, `Go-South`, `Go-West`) and declared named constants for the actions.

Part 1: Value Iteration

Solve this problem using synchronous, in-place value iteration. In synchronous value iteration, you calculate new utilities for all the states on every iteration. "In-place" means that you calculate the utilities at iteration $(i+1)$ using both the utilities from iteration i and the new utilities being calculated, as they become available. Here is Value Iteration pseudocode, which you can use as a guide:

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
    rewards $R(s)$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum change in the utility of any state in an iteration

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']$
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return U

Your implementation should allow the user to specify a discount factor γ and a maximum allowable error ϵ in the state utilities. You should also allow the user to specify the key-loss probability, the positive terminal state reward, the negative terminal state reward, the step cost, and the solution technique (`v` for value iteration, `p` for policy iteration,

and `q` for Q-learning) **from the command line. Please write your program so that the arguments are given in that order.** For example:

```
java MDP 0.99 1e-6 0.5 1 -1 -0.04 v
```

specifies a discount rate of 0.99, a maximum state error of 0.000001, a key loss probability of 0.5, a positive terminal state reward of +1, a negative terminal state reward of −1, a step cost of −0.04, and value iteration. You don't have to use Java, but **your program must allow me to provide the specified arguments in the specified order on the command line**. Your program should stop iterating when the error target is met.

Your program should print out:

1. the solution technique used,
2. all of the parameters used for that run,
3. the number of iterations in that run,
4. the time taken,
5. the final state utilities, and
6. the policy implied by these utilities.

The last two items are provided by the `printUtilitiesAndPolicy` method I have supplied.

Questions to Answer

I will be asking you to conduct experiments comparing the different solution techiques. You will also need to write a report describing your results. I will describe the experiments and format of the report in subsequent handouts.

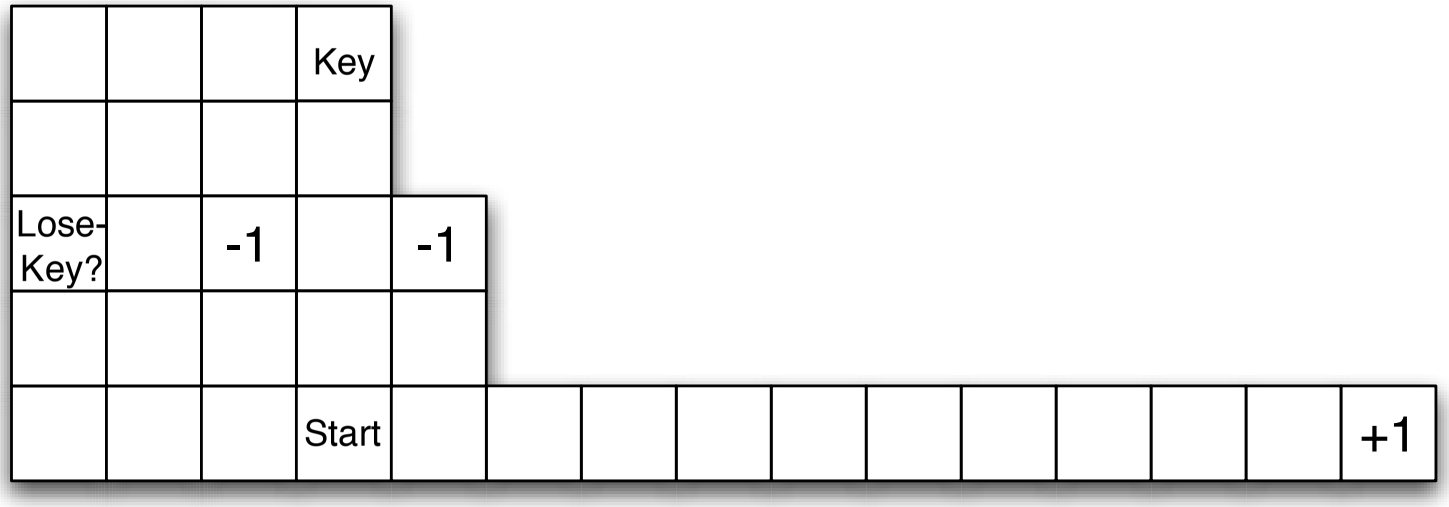
Grading

Your assignment will be graded approximately as follows:

60%: your code (correctness, clarity, etc.),

20%: your experiments, and

20%: your report.


Key
Lose-Key?
-1
-1
Start
+1