Programming Assignment #1
Due: Beginning of class, Thursday, July 19th

In this programming assignment, you will be implementing a robot simulation that makes use of some of the concepts of search and delayed-reinforcement learning described in Chapter 10 of the Nilsson textbook. This will give you a chance to better understand some of these techniques, as well as experiment with some of the properties of these algorithms.

You should read this entire assignment thoroughly before beginning to do any programming. This will help you to understand all the modifications that you will be making to your program to solve the various problems below. By knowing the changes that you may be required to make, you can write more general code from the outset and thereby have minimal coding modifications to make to answer all the questions.

The World
For this assignment, you will be dealing with a robot (denoted by an R) trying to reach a goal (denoted by a G) in a 10 by 10 grid world. Two examples of such grid worlds are shown below. In the first world there are no walls, while in the second world there are.

Note that the robot always begins in the initial cell (0, 0) and the goal is always in the cell (9, 9).

Actions
The robot can take four possible actions: moving Up, Down, Left or Right. Each such action moves the robot to the adjacent grid cell in the given direction. In some situations,
however, not all four actions are valid. For example, the robot can only move into grid cells that are not blocked by a wall. Also, the robot is not allowed to take an action that would move it off the edge of the grid (e.g., the robot cannot move Down from its initial position).

**Rewards**
The robot’s goal is to get to cell (9, 9). Recall that in delayed-reinforcement learning, rewards are not associated with a particular node. Rather, there is a reward associated with taking a particular action in a given node. So, we will give rewards to the robot when it takes an action that moves it into the goal cell.

The robot should get a reward of 100 when performing action Up in cell (9, 8), or action Right in cell (8, 9). That is, when the robot takes an action that results in the robot moving into cell (9, 9), it is given the reward of 100.

**Writing the Simulation**
You should write a program to simulate the robot in the two worlds described above. The robot will be using delayed-reinforcement learning to learn policies of how to act in these worlds. You will find Section 10.4 of the textbook particularly useful when implementing your simulation. You are welcome to use any programming environment that you like, as long as the simulation is written using the C, C++, or Java language.

Your program should use the value iteration method described in class and in Section 10.4 of the textbook to learn action policies in World 1 and World 2 shown above. Different policies should be learned for each world.

In order to implement your simulation, you should use the following parameters for learning:

- Each of your simulation runs should last for 100,000 actions that the robot takes.
- Whenever the robot manages to reach the goal cell (9, 9), it should be automatically transported back to its initial cell (0, 0) to continue its learning process. Keep track of the total number of times that the robot reaches the goal node during the simulation.
- For your simulation, use Gamma ($\gamma$) = 0.9 and Beta ($\beta$) = 0.3
- The initial values associated with the nodes in your value iteration procedure should be random numbers between 0.0 and 1.0. See the appendix at the end of this handout for more information on how to generate random numbers in C, or use the java.util.Random package if you are programming in Java.
Problem 1. (50 points) Initial Simulation

Write a program that uses delayed-reinforcement learning as described above to find a policy that will move the robot from its initial states to the goal node. You should show the values associated with each node in the grid for both Worlds 1 and 2 after the simulation has completed. An example of such values for the nodes in the grid is shown below.

Value grid:
13.13 17.29 24.33 29.75 34.89 51.90 59.01 67.24 82.46 0.61
11.71 17.23 18.99 28.71 35.60 43.90 49.85 63.13 78.23 97.44
13.62 16.51 19.76 24.25 28.74 45.74 54.67 58.29 79.24
11.57 14.33 15.29 20.73 24.34 24.51 30.85 48.07 55.34 49.32
8.25 10.49 12.06 17.07 19.51 21.97 28.74 40.98 40.92 38.73
6.60 9.44 11.73 12.60 15.12 20.01 23.24 29.13 31.81 32.07
6.30 7.09 9.25 12.00 15.41 18.13 20.25 23.04 23.98 22.38
5.72 7.13 8.63 9.39 14.01 14.60 15.80 17.55 13.59
5.10 5.87 6.79 8.29 10.64 10.62 11.27 12.34 6.41 0.99
4.67 5.03 5.67 6.69 7.73 7.95 8.50 6.10 0.29 0.28

Note that the goal cell itself has low value since the reward is received when *moving into the cell*.

Also, for each cell in the grid (for each world), show the preferred action of the robot (i.e., the action that maximizes the expected reward from that cell). A simple way of representing such action policies is simply to use a character for each cell, where:

‘^’ denotes Up
‘v’ denotes Down
‘>’ denotes Right
’<’ denotes Left

An example of such an action policy is given below.

Policy grid:
> > > > > > > v
> > > > > ^ > > ^
> ^ > ^ > > ^ > ^
> ^ > > > > > > ^
^ > ^ > ^ > ^ ^ ^
> > > > > > > > ^
> > > > ^ > > ^ ^ ^
> > > > > > > ^ ^ ^
> ^ > > ^ ^ ^ ^ ^
^ ^ ^ ^ ^ ^ ^ ^ ^

The policy given here is, in fact, optimal for any initial position of the robot. This is the case, since from any position in the grid, the robot will move directly toward the goal cell (albeit not always in a uniform sequence of Up and Right moves).

Note that for the simulations you run in this problem, your robot should not perform any “exploration” (i.e., taking random actions). The robot should always take the action with
the maximal expected value. In Problem 2 below, you’ll be addressing the exploration issue.

**Answer the following questions regarding your simulation runs.**

After 100,000 steps, how many times did the robot reach the goal in World 1?

After 100,000 steps, how many times did the robot reach the goal in World 2?

Is the policy you found in World 1 optimal for all initial states of the robot? Why or why not?

Is the policy you found in World 2 optimal for all initial states of the robot? Why or why not?

**Problem 2. (20 points) Exploration vs. Exploitation**

In this problem, you’ll examine the exploration vs. exploitation trade-off in your simulations. In order to do this, you will add an “exploration factor” to your robot simulation program. This means that for any given move, the robot should take a random valid action (an action that does not move the robot into a wall or off the grid), as opposed to taking the action that maximizes expected reward, with probability $P$. Add this enhancement to your program.

Run your simulation on Worlds 1 and 2, with the following values of $P$ (the probability of taking a random action):

- $P = 0.1$
- $P = 0.3$
- $P = 0.5$
- $P = 0.8$

As in Problem 1, show the values associated with each node in the grid for both Worlds 1 and 2, for each run (8 total grids). Also, for each cell in the grid for each world, show the preferred action of the robot (i.e., the policy learned by the robot).

Note that the use of exploration in this problem will generally cause you to get qualitatively different policies than you found without using exploration in Problem #1. If you don’t find this to be the case, you should carefully check your code for bugs.

**Answer the following questions regarding your simulation runs.**

In which of the 8 runs did you find an optimal policy to the goal for any initial state of the robot in the grid?

After 100,000 steps, how many times did the robot reach the goal in each of the 8 runs? How does the value of $P$ seem to effect the number of times that the robot reached the goal node in each run?

Did you find an optimal policy in every run? Explain why or why not.
Problem 3. (30 points) Island-Driven Policies

In this problem, you’ll use some of the flavor of island-driven search in order to find a policy for the robot to reach the goal. In worlds that contain many states (generally far more states than the example worlds presented here), it is sometime desirable to have intermediate nodes that provide rewards, so as to help a robot find the path to the eventual goal. In some cases, however, providing such intermediate rewards may be problematic.

For this problem, you should only use World 1. You do not need to do anything with World 2 in this problem.

We would like to add some additional intermediate rewards to the simulation in World 1, to reward the robot for getting “half-way” to the goal. In your simulation, add the following rewards (in addition to the previous rewards for moving into cell (9, 9)):

The robot should get a reward $X$ (the variable $X$ will be set to various values described below) when performing action Up in cell (5, 4), or action Right in cell (4, 5). That is, the robot gets an intermediary reward when it takes an action that moves it into cell (5, 5) from the left or bottom side. The reason for having this intermediary reward is to help the robot get "half-way" to the goal.

While still using the exploration mechanism you implemented in Problem 2 (using the value $P = 0.5$), try running your simulation (for World 1 only), with the following values for the intermediate reward $X$:

- $X = 0.1$
- $X = 1$
- $X = 10$
- $X = 50$.

As in Problem 1, show the values associated with each node in the grid for World 1 after the simulation has completed, for each value of $X$ (4 total grids). Also, for each cell in the grid, show the preferred action of the robot (i.e., the policy learned by the robot) for each of the 4 runs.

Note that you may encounter some interesting policies in this problem. I can’t give you too much detail about when I mean by “interesting” without giving away too much of the problem. Rest assured that such interesting policies are normal for this problem, and seeing them is, in fact, the whole point of the problem.

Answer the following question regarding your simulation runs.

For which values of $X$ did you get an optimal policy to reach the goal from any initial cell? Explain for each value of $X$ why an optimal policy was or was not found.

Deliverables

Turn in the printouts of the results of your simulation runs described above. Also, turn in the answers to each of the questions given above. Finally, turn in a printout of the source code to your simulation.
Appendix: Random Number Generation in C
You may find the following code useful for generating random real and integer values. You are not required to use this code. It is merely provided for convenience if you happen to be programming in C/C++. You can also find this code on-line (to save you typing time) by following the link "Code For Random Numbers" on the CS121 class web page.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* These functions come from Eric Roberts' "The Art and Science of C". */

/* Function: InitRandom */
* --------------------
* This function seeds the random number generator with the current time.
* It should be called once (and only once) prior to calling any other
* random number function.
*/
void InitRandom(void)
{
    srand((int) time(NULL));
}

/* Constant: RAND_MAX */
* ------------------
* Unfortunately, several libraries that supposedly conform to
* the ANSI standard do not define RAND_MAX in <stdlib.h>. To
* reduce portability problems, this interface defines RAND_MAX
* to be the largest positive integer if it is undefined.
*/
#ifndef RAND_MAX
#define RAND_MAX ((int) ((unsigned) ~0 >> 1))
#endif

/* Function: RandomReal */
* --------------------
* This function returns a random real number between low and high, inclusive.
*/
double RandomReal(double low, double high)
{
    double d;
    d = (double) rand() / ((double) RAND_MAX + 1);
    return (low + d * (high - low));
}

/* Function: RandomInteger */
* --------------------
* This function returns a random integer between low and high, inclusive.
*/
int RandomInteger(int low, int high)
{
    int k;
    double d;
    d = (double) rand() / ((double) RAND_MAX + 1);
    k = (int) (d * (high - low + 1));
    return (low + k);
}
```