CS181 — Assignment 5: Markov Decision Processes and Reinforcement Learning

Spring 2010
Professor David Parkes
Out Friday April 8th
Due Tuesday, April 19th at 11:59 pm

General Instructions: This assignment consists of both theory and experimental components. You must submit your writeup as a PDF.

Darts

In the game of darts, players throw darts at a dartboard and receive points depending on where the dart hits. The standard dartboard is divided into 20 numbered wedges, as shown in the picture below.

There are six regions on the dartboard, determining the number of points scored:

Region number	Region	Points
0	Center	50
1	Inner ring	25
2	First patch	wedge number
3	Middle ring	3 * wedge number
4	Second patch	wedge number
5	Outer ring	2 * wedge number
6	Miss	0

Thus, the greatest number of points from a single throw is 60 points, which is achieved by hitting the middle ring of the 20 wedge. The world beyond the outer ring, region 6, is a miss worth 0 points.

In the version of darts called 301, a player starts with 301 points. In each round, a player throws one dart at the dartboard. The number of points for the throw is subtracted from the player's current score, unless the player's score would become negative. In the latter case, the player's score is unchanged. **The goal is to reach 0 points with as few darts as possible.** For example, suppose a player throws hits the middle ring of the 20 wedge for six straight rounds. Then, the player's score is reduced by 60 in each of the first five rounds and reaches 1. In the sixth round the score will remain at 1. To finish the game, the player will have to throw exactly 1.

(The full version of the game is slightly more complex. There are two players, and each player takes a turn throwing three darts. The goal is to reach 0 before the other player. For the purposes of this assignment, we will assume that darts is a one-player game and that the goal is to minimize the number of darts thrown.)

This assignment focuses on designing a strategy for playing darts. You can download support code in Python at:
`http://www.seas.harvard.edu/courses/cs181/docs/Asst5.zip`

As you will see, the code has been written so that the size of the dartboard and the initial player score can be varied. As the number of wedges varies, the amount of points from hitting the center of the board is $2.5 * \#ofwedges$, while the amount of points from hitting the inner ring is $1.25 * \#ofwedges$. We will only consider games with a number of wedges equal to a multiple of 4, as we wish to consider games in which all scores are integers. Constants defining the standard darts game, a medium-size game, and a very small game have been provided. For this assignment, you may wish to use the small game to develop and debug your algorithms. You should use either the medium-size or very small game for experimentation, as specified.

The files with support code are `darts.py`, `throw.py`, `mdp.py`, `modelbased.py`, and `modelfree.py`. You will not need to change `throw.py`, except to change the size of the darts game. Look at the `main()` function in `darts.py` to see how to shift from running code from the mdp, model-based, and model-free modules. Again, functions that you will need to implement are identified in comments with the `<CODE HERE>` symbol, and a more detailed description of programming tasks can be found in questions 2 and 4.

1. **[10 Points]**

   You are part of a team designing a darts-playing robot. The team is divided into two groups. The other group is responsible for designing the dart thrower, which attempts to hit a particular location (i.e. wedge number and region) of the dartboard given a target. Your group is responsible for deciding where that target should be.

   As a first pass, your group decides to use a decision theoretic strategy to decide where to throw. To implement it, you will need:

   - A probability distribution over the possible results of a a dart throw, given a target.
   - A utility function, that specifies the value to you of scoring a certain number of points, given your current score.

   You will then keep selecting the throw with best utility.

   This question focuses on a possible design of this utility function. Assume, for now, that the probability distribution will be obtained from the other group.

   (a) **[4 Points]** Given a probability distribution and utility function, write equations showing how to determine the optimal action, given the current score.

   (b) **[6 Points]** One of your colleagues proposes the following utility function:

   $$U(points, score) = \begin{cases} points & : points \leq score \\ 0 & : points > score \end{cases}$$

   In other words, the utility is equivalent to your change in score. What do you think of this proposal? If a dart player is designed using this proposal, what decisions would you expect it to make well, and what decisions would you expect it to make poorly?

2. **[28 Points]**

   You observe that if only you had a probability distribution over the result of a dart throw, given a target, you could cast the problem as an MDP. Someone on the other team says

   > We haven't really worked it out for sure, but we think the distribution is this. If you aim for a wedge, then there's a 0.4 probability of hitting that wedge, a 0.2 probability of hitting each of the wedges next to it, and a 0.1 probability of hitting each of the wedges that are two away. Similarly, if you aim for a region, there's a 0.4 of hitting that region, a 0.2 probability of hitting region outside it, and a 0.1 probability for the region outside that. There's also an 0.2 probability of hitting the region inside the region of aim, a 0.1 for the region inside that,

except that if you go inside the center, you begin to move outward again, staying on the same half of the dartboard. So, the region "inside" the center is the inner ring. For example, if you aim at the center, you will have 0.4 probability of hitting the center, 0.4 probability of hitting the inner ring, and 0.2 probability of hitting the first patch. And if you aim at the inner ring, you will hit the center with probability 0.2, the inner ring with probability 0.5, the first patch with probability 0.2 and the middle ring with probability 0.1. The wedge and region that you hit are completely independent of each other.

You are skeptical that this is the correct probability distribution, but you decide to solve the MDP anyway.

To complete this portion of the assignment, you will need to make changes to 2 files: `darts.py` and `mdp.py`. Use `python darts.py` to build your solution after uncommenting the specified lines in the `main()` function of `darts.py`.

(a) **[2 Points]** What are the states and actions of your MDP model? Complete the `get_states()` function in `darts.py`. (you may use the `test_get_states` unit test at this point)

(b) **[4 Points]** Define the reward function: what reward or punishment do you receive for each state and action? What role does the discount factor play? Once you have specified this function, complete the reward function `R()` in `darts.py`. (see also the `test_R` unit test) [1]

(c) **[6 Points]** Construct the transition function, using the probability distribution described above. That is, complete the transition function `T()` in `mdp.py`. (validate with the `test_T` unit test)

(d) **[4 Points]** We have implemented the infinite horizon value iteration algorithm in `mdp.py`. Why might we choose to use an infinite rather than finite horizon to find an optimal policy in the dartboards scenario? You may now get your first results by running the `Warm-up` task.

(e) **[6 Points]** Describe the intuition behind an optimal policy resulting from value iteration using the small game.

(f) **[6 Points]** Still referring to the small game, how does the optimal policy vary as you change the discount factor? What is common across optimal policies?

[1] In class we defined the reward model as taking a state and an action. However in some cases you can simplify it so that it only depends on the state and is the same for all actions. Is this such a case? If so, feel free to simplify.

3. **[25 Points]**

After successfully implementing value iteration for the miniature darts game, you realize that it might be quite slow for the full version of darts. You remember that the policy iteration algorithm is generally considered to be quicker than value iteration. You explain the algorithm to your colleagues, but they are skeptical. "What if the algorithm goes in cycles and never terminates?", they ask. "Or what if it terminates at a locally optimal strategy?".

Your task is to convince your colleagues that policy iteration works.

(a) **[10 Points]** Prove that if policy iteration terminates with the policy π^*, π^* is an optimal stationary policy.

(b) **[10 Points]** Prove that if π^1 is the policy before a policy iteration phase, and π^2 is the policy after the phase, then for every state s, $V^{\pi^2}(s) \geq V^{\pi^1}(s)$. [Hint: first try to prove $V^{\pi^2}(\hat{s}) \geq V^{\pi^1}(\hat{s})$ when the action is only improved in state $\hat{s}$. Then argue this property holds for all states.]

(c) **[5 Points]** Deduce that policy iteration always terminates with an optimal stationary policy. (You can do this part even if you don't succeed in parts (a) and (b).)

4. **[30 Points]**

Your source in the dart-throwing group comes back to you and says the probability distribution he gave you was completely wrong. In fact, the distribution is more complex, and they don't know the exact probabilities. However, they can provide you with a simulator for the dart thrower.

To initialize the dart thrower in the code, call `init_thrower()`. When `init_thrower()` is called, the simulator is initialized with a random seed. This feature allows you to run multiple experiments, each with a different random seed. If you want to use the same dart thrower behavior for a series of experiments, make sure to set the random seed before calling `init_thrower()`.

Because you don't know the probability distribution, you can't solve the MDP. Instead, you propose to use reinforcement learning.

(a) **[9 Points]** We have implemented a model-based reinforcement learning algorithm, which uses an infinite horizon value iteration algorithm very similar to that of Question 2 to solve the learned MDP. You will need to implement two exploration/exploitation strategies for deciding how to act during an epoch based on the current result of value iterations. These strategies should be significantly different — not just changing the value of a parameter. Write your code for this part

of the question in the `ex_strategy_one()` and `ex_strategy_two()` functions found in `modelbased.py`. Run the next tasks for each exploration/exploitation strategy, and thus create a graph of the relationship between performance (average number of throws to complete at game) and the epoch size used to update the optimal policy. How do these graphs compare for both strategies?

(b) **[15 Points]** Implement the model-free Q learning algorithm as well as two exploration/exploitation strategies – these strategies may be the same as those in part (a). You will also have to pick a learning rate. Write your code for this part of the question in `modelfree.py`. Use the `Q-learning` task with the medium-size game to compare the performance of your algorithm (average number of throws to complete a game) for both exploration/exploitation strategies.

(c) **[6 Points]** Compare the performance of the model-based and model-free algorithms. Why do you think these algorithms performed relatively well or poorly?