CS181 Assignment 5—Markov Decision Processes and Reinforcement Learning

Lucas Freitas and Angela Li

April 27, 2013

1. (a) We have a probability distribution $P(\text{points} \mid \text{target})$ and a utility function $U(\text{points, score})$. The expected utility of aiming for a target t is:

$$\sum_{k \in K} P(\text{k}|\text{t})\; U(\text{k, S})$$

Where K is the set of possible points scored after one throw, and S is the current score. To determine the optimal action, we use the maximum expected utility principle:

$$\underset{t \in T}{\text{argmax}} \sum_{k \in K} P(\text{k}|\text{t})\; U(\text{k, S})$$

Where T is the set of possible targets we can aim for.

(b) This utility function works fairly well for paring down the score in a relatively small number of dart throws*, but only until it reaches the point where it is possible to win in one throw. At that point, it makes decisions poorly in that it values a non-winning throw as nearly as good as a winning throw.

For example, if the current score was 20, a throw resulting in a 20-point gain would end the game (and thus should be valued extremely highly). However, a throw resulting in a 19-point gain (which requires, at the very least, one more throw to win the game) would be valued at only 5% less utility than a winning throw. So the proposed utility function is not conducive to good decision-making in states where it is possible to win in one move—in those cases, the winning move should be valued significantly more highly.

*Even a quick paring-down of the score, however, is not always desirable. Consider that there are probably significantly many more ways to score 10 points in a throw than there are ways to score 1 point in a throw. However, if the current score was 20, the utility function would reward a score of 19 higher than a score of 10, even though both point values require at the very least one more dart throw to win; a throw of 10 shoul, however, be rewarded more because it creates more opportunity for a winning throw (of 10 points) than a throw of 19 (which requires a winning throw of 1 point).

2. (a) In our MDP model, the states are the possible scores in the game (so every integer in the range [0, `START_SCORE`] is a state), and the actions are the possible areas (ring and wedge) that the dart player can aim for in any given turn.

Therefore `get_states()` should return `range(throw.START_SCORE + 1)`.

(b) The reward function should not depend on the action a - it should only take into account if the user has won ($s == 0$) or not yet. Thus, we can write the function as `return 1 if s == 0 else 0`. The problem about this reward function is that it doesn't show our preference of winning earlier than late.

That is why the discount factor is important - we should use it to show that bias for earlier wins. If the discount factor was 1, then we would still not be able to add that preference (we would still not care if we won early or late), and if the discount factor was 0, that would mean that we don't care about winning later at all, which is not good, since we will definitely need future actions in order to win eventually. Thus, we should have a discount factor between 0 and 1, closer to 0 in order to show the preference for winning early.

(c) Implementation included in appendix.

(d) There is no guarantee that the darts game can be won in a certain number of steps, so it doesn't make sense to impose an arbitrary finite horizon on the game—after all, the optimal policy should place primary importance on being able to get to a score of 0, and secondary importance on accomplishing that score in as few steps as possible.

(e) Running experiments for the small game, it seems that the optimal policy resulting from value iteration focuses on trying to hit an area that is going to make the user win with that throw. For instance, the program always starts aiming at edge 3, ring 3 at first, which would give a score of 9 and an instantaneous win. The same happens for scores in the range $[1, 6]$ - for a given score k in that range, the program aims at an area that gives a score of k. For 7 and 8, on the other hand, the program aims at 2 and 3 respectively, which makes sense, since hitting an area and getting exactly 7 or 8 is not very likely (actually not even possible for 7). Thus, the program first reduces the score and then tries to win in the next hit.

(f) The table below shows the program policy for values of γ from 0 to 1. Each row k for a given γ has the value of the area that is aimed at when the score in the game is k and the discount factor is γ.

γ/k	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	6	1	1	1	1	1	1	1	1	1	1
2	6	2	2	2	2	2	2	2	2	2	2
3	6	3	3	3	3	3	3	3	3	3	3
4	6	4	4	4	4	4	4	4	4	4	4
5	6	5	5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6	6	6	6
7	6	2	2	2	2	2	2	2	2	1	1
8	6	3	3	3	3	3	3	3	3	1	1
9	6	9	9	9	9	9	9	9	9	9	9

From the table, we can firstly see that for $\gamma = 0$ the are that is aimed at is always the same (area that gives score of 6). That is not surprising, since a discount factor of zero would mean that we are not looking into the implications of our actions at all, so it doesn't matter what score we currently have, our strategy will always be there same.

Secondly, we can see that the strategy for $k = 1$ to 6 or $k = 9$ and $\gamma > 0$ is to always aim at an area that gives the same score as k. The actual wedge and ring targeted actually changes for some γ (for 2, for instance, it changes from wedge 2, ring 4 to wedge 1, ring 5 for $\gamma \geq 0.9$).

Finally, we see that for $\gamma = 0.9$ or $\gamma = 1.0$, we do have a change in strategy, aiming to 1 instead of 2 or 3. That indicates that since γ is getting higher, we care less and less about how long the game lasts, and the program is just aiming on finishing the game instead of finishing it fast.

3. (a)

(b)

(c)

4. (a)

   (b)

   (c)

Appendix

Code for 2.c

```
def T(a, s, s_prime):
  # takes an action a, current state s, and next state s_prime
  # returns the probability of transitioning to s_prime when taking action a in state s

  probability = 0.0

  # -2 -1 0 1 2
  for w in range(-2, 3):
    # hit the wedge (0)
    if abs(w) == 0:
      p_wedge = 0.4
    # hit region outside the wedge (-1 or 1)
    elif abs(w) == 1:
      p_wedge = 0.2
    # hit region outside of that (-2 or 2)
    else:
      p_wedge = 0.1

    # get the wedge and do % to loop around in case of going around circle
    wedge = (a.wedge + w) % throw.NUM_WEDGES

    # same thing, but now for the ring
    for r in range(-2, 3):
      # hit the ring
      if abs(r) == 0:
        p_ring = 0.4
      # hit region outside the ring
      elif abs(r) == 1:
        p_ring = 0.2
      # hit region outside of that
      else:
        p_ring = 0.1

      # get the ring and do % to loop around in case of going around circle
      ring = abs(a.ring + r)

      score = throw.location_to_score(throw.location(ring, wedge))
      if score == s - s_prime:
        probability += p_wedge * p_ring

  return probability
```