Reinforcement Learning Report

Vasu Soni

November 5, 2023

Contents

1 Pool Code Explanation

1.1 Python Code with Line Wrapping

```
import os
import sys
import random
import json
import math
import utils
import time
import config
import numpy
random.seed(73)

def best_hole(ball_pos, holes, modified_holes):
    ans = {}
    besthole = []
    for ball, pos in ball_pos.items():
        if ball != 'white' and ball != 0:
            nearest_hole = find_nearest_hole(ball_pos[ball], holes)
            if distance_between_points(pos, nearest_hole) < 50:
                besthole.append((pos[0]-10, pos[1]-10))
                ans[ball] = besthole
                continue
            for hole in modified_holes:
                if deviation(ball_pos[0], pos, hole) < 82:
                    besthole.append(hole)
            ans[ball] = besthole
    return ans

def deviation(white, ballpos, hole):
    return abs(find_angle(white, ballpos) - find_angle(ballpos, hole)
        ) * 180

def find_angle(C1, C2):
    deltax = C2[0] - C1[0]
    deltay = C2[1] - C1[1]
    angle = math.atan2(deltax, deltay)
    if angle >= 0 and angle < math.pi / 2:
        angle = -(math.pi / 2 + angle) / math.pi
    elif angle >= math.pi / 2:
        angle = -(angle - 3 * math.pi / 2) / math.pi
    elif angle < 0 and angle > -math.pi / 2:
        angle = -(math.pi / 2 - abs(angle)) / math.pi
    elif angle <= -math.pi / 2:
        angle = (abs(angle) - math.pi / 2) / math.pi
    return angle

def move_along_line(x, angle, distance):
    angle = math.pi * angle
    new_x = x[0] + distance * math.sin(angle)
    new_y = x[1] + distance * math.cos(angle)
    return new_x, new_y

def distance_between_points(point1, point2):
    x1, y1 = point1
    x2, y2 = point2
```

```
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

def find_nearest_hole(pos, hole_positions):
    nearest_hole = None
    nearest_distance = float('inf')

    for hole in hole_positions:
        if distance_between_points(hole, pos) < nearest_distance:
            nearest_distance = distance_between_points(hole, pos)
            nearest_hole = hole
    return nearest_hole

class Agent:
    def __init__(self, table_config) -> None:
        self.table_config = table_config
        self.prev_action = None
        self.curr_iter = 0
        self.state_dict = {}
        self.holes = []
        self.ns = utils.NextState()
        self.modified_holes = [(64, 64), (64, 436), (500, 50), (500,
            450), (936, 64), (936, 436)]

    def set_holes(self, holes_x, holes_y, radius):
        for x in holes_x:
            for y in holes_y:
                self.holes.append((x[0], y[0]))
        self.ball_radius = radius

    def find_force(self, white_to_ball, ball_to_hole, deviation):
        v1 = 0.0133 * white_to_ball + (0.0133 * ball_to_hole + 0.06 *
            0.98) * (0.98 / math.cos(math.radians(deviation)))
        F = (v1 + 3 / 14) / 18.428
        return F

    def value(self, nextstate, state, deviation_angle, force):
        n = 0
        n += 4 * (len(state) - len(nextstate))
        besthole = best_hole(nextstate, self.holes, self.
            modified_holes)
        for ball, pos in nextstate.items():
            if ball != 'white' and ball != 0 and ball in besthole:
                if distance_between_points(pos, find_nearest_hole(pos
                    , self.holes)) < 50:
                    n += 1
                    for hole in besthole[ball]:
                        if hole in [(500, 40), (500, 460)] and
                            deviation(nextstate[0], pos, hole) < 10:
                            n += 2
                        if hole not in [(500, 40), (500, 460)] and
                            deviation(nextstate[0], pos, hole) < 10:
                            n += 1
        if deviation_angle < 25:
            n += 2
        if deviation_angle > 85:
            n = n - 5
        if force > 0.4:
            n = n - 1
```

```
        if force < 0.06:
            n = n - 1000
        return n

    def action(self, ball_pos=None):
        bestforce = 0.4
        goodstates = []
        besthole = best_hole(ball_pos, self.holes, self.
            modified_holes)
        bestangle = 0
        for ball, pos in ball_pos.items():
            if ball != 'white' and ball != 0 and ball in besthole:
                for hole in besthole[ball]:
                    angle = find_angle(pos, hole)
                    new_pos = move_along_line(pos, angle, self.
                        ball_radius * 2)
                    angle = find_angle(ball_pos[0], new_pos)
                    force = 0.1 + self.find_force(
                        distance_between_points(ball_pos[0], pos),
                                                distance_between_points
                                                    (pos,
                                                    find_nearest_hole(
                                                    hole, self.holes))
                                                    ,
                                                deviation(ball_pos
                                                    [0], pos, hole))
                    nextstate = self.ns.get_next_state(ball_pos, (
                        angle, force), 73)
                    goodstates.append((self.value(nextstate, ball_pos
                        , deviation(ball_pos[0], pos, hole), force),
                                       (angle, force)))
                    bestangle, bestforce = max(goodstates, key=lambda
                         item: item[0])[1]
        return bestangle, bestforce
```

article

Pool Agent Strategy Explanation Your Name November 5, 2023

2 Introduction

Explain the strategy and approach used in my pool agent.

3 Some Important Functions

3.1 Find Nearest Hole

This function takes two arguments: the ball position for which we want to find the nearest hole and the hole positions (a list of all hole positions). The function iterates through the holes and returns the coordinates of the nearest hole.

3.2 Distance Between Points

This function takes the coordinates of two points between which we want to find the distance and returns the Euclidean distance between these two coordinates.

3.3 Move Along Line

This function takes three arguments:

1. Coordinates of the point from which we want to move.
2. Angle along which we want to move.
3. Distance (how much we want to move).

It returns the coordinates of the point after moving along the specified angle (forward and backward is simply dealt with by the sign of the angle). This function is crucial since it allows me to determine the coordinates where I want my white ball to be to hit the colored ball and move as desired.

3.4 Find Angle

This function takes two arguments: the coordinates of two points to find the angle of the line joining them with respect to the y-axis. Since the math.atan2(y, x) function returns the angle from the x-axis, I have adjusted the range of values by adding or subtracting pi/2 or 3pi/2 to transform the angle as required for the pool table.

3.5 Best Hole

This function takes the current state, actual hole positions, and modified hole positions as arguments. It returns a dictionary with good holes for every ball. For finding good holes, two conditions are checked:

1. The distance from the nearest hole (actual positions) is calculated and added to that if the ball key is less than 50 units away.
2. A check is made for the angle of deviation for the shot, and if it is less than 82 degrees (a value guessed to work well), the hole is added to the dictionary with the ball as the key.

4 Finding Good Holes (Step 1)

In this step, I identify the "good holes" for all the balls and store them in a dictionary called "besthole." Good holes are those with a deviation angle of 82 degrees or less or a distance of less than 50 units from the ball.

5 Determining Force (Step 2)

After obtaining the "besthole" dictionary, I calculate the force required to achieve the desired shot. The force is determined using a physics-based approach, taking into account the velocity components and friction. The formula for force is derived as follows:

$$F = \frac{0.0133 \cdot white - to - balldistance + (0.0133 \cdot ball - to - holedistance + 0.06 \cdot 0.98) \cdot \left(\frac{0.98}{\cos(deviationangle)}\right)}{18.428}$$

6 Choosing the Best Action (Step 3)

I iterate through all the good holes for each ball in the "besthole" dictionary, considering both the force and angle. After calculating the next state, I evaluate it using a value function. The action corresponding to the highest value of the next state is selected.

7 Assigning Values to Next States (Step 4)

The value function assigns integer values to the next state based on several checkpoints:

7.1 Checkpoint 1

If the number of balls in the next state is less than the current state, a reward is given equal to four times the difference in the total number of balls.

7.2 Checkpoint 2

For remaining balls, if they are near or far from the hole, rewards are assigned. Specifically, if the ball is close to a hole (within 50 units), a reward of 1 is given. For middle holes with a deviation angle of less than 10 degrees, a reward of +2 is assigned. For corner holes with a deviation angle of less than 10 degrees, a reward of +1 is given.

7.3 Checkpoint 3

The deviation angle of the action is considered. If the deviation angle is less than 25 degrees (indicating a nearly straight shot), a reward of +2 is assigned. If the deviation angle is greater than 85 degrees, a penalty of 5 points is applied.

7.4 Checkpoint 4

The force used to reach the next state is assessed. If the force is greater than 0.4, one point is deducted. If the force is less than zero, a significant penalty of 1000 points is applied (to avoid certain actions). If the force is less than 0.1, another 1000 points are deducted as it results in nearly zero movement of the cue ball.

8 Modified Hole Positions

For handling wall hits and other special cases, I have used modified hole positions. These positions are as follows:

$$ModifiedHoles = [(64, 64), (64, 436), (500, 50), (500, 450), (936, 64), (936, 436)]$$

9 Conclusion

Overall, my strategy is very similar to picking the action for which the action-value function is maximum, and the action-value is approximated using the value function. I consider various checkpoints and assign or cut points based on the state I am in and the action that took me to this state.