CS747 - Foundations of Intelligent and Learning Agents

CS747 Autumn 2023

Chinmay Makarand Pimpalkhare
200100115

Assignment 3 | Monte Carlo Tree Search-based Efficient Billiards Agent

1 Geometrical Computations

1.1 What location should we aim at?

Implementation:

```python
def hit_ball_to_hole(ball_pos, hole_list, ball, hole, mu_value=2):
    hole_location = numpy.array(hole_list[hole])
    mu = mu_value
    #print("Hole location :", hole_location)
    ball_location = numpy.array((ball_pos[ball][0], ball_pos[ball][1]))
    #print("Ball location: ", ball_location)
    direction_from_ball_to_hole = get_unit_vector(ball_location, hole_location)
    #print("Unit vector from ball to hole: ", direction_from_ball_to_hole)
    ball_radius = config.ball_radius
    ideal_center_x = ball_location[0] - mu*direction_from_ball_to_hole[0]*ball_radius
    ideal_center_y = ball_location[1] - mu*direction_from_ball_to_hole[1]*ball_radius
    ideal_center  = numpy.array((ideal_center_x, ideal_center_y))
    #print("Ideally we want white to hit at: ", ideal_center)
    white_location = numpy.array((ball_pos["white"][0], ball_pos["white"][1]))
    #print("Currently white is at: ", white_location)
    distance_ball_and_white = get_distance(ball_location, white_location)
    #print("Distance between centers of ball and white:" , distance_ball_and_white)
    distance_ideal_and_white = get_distance(ideal_center, white_location)
    #print("Distance between the ideal center and white:", distance_ideal_and_white)
    threshold_distance = numpy.sqrt(numpy.maximum(distance_ball_and_white**2 - \
                                              mu*mu*(ball_radius**2), 0))*1.05
    #print("The threshold distance is:", threshold_distance)
    if distance_ideal_and_white <= threshold_distance:
        #print("Point is reachable")
        vector_white_to_ideal = get_unit_vector(white_location, ideal_center)
        #print("Unit vector in the direction is:", vector_white_to_ideal)
        angle = compute_angle_on_screen(vector_white_to_ideal)
        #print("Angle to be moved in:", angle)
        return angle
    #else:
        #print("Cannot reach given point, returning None!")
```

This is the function which I am using in order to obtain the position at which at the aim needs to be taken in order for the chance of the ball going into the hole to be maximum.

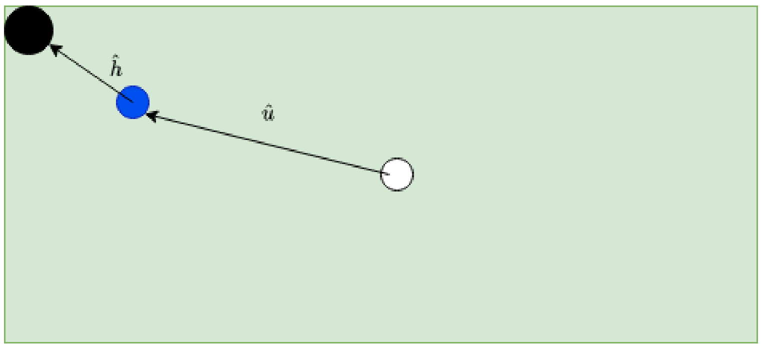

Figure 1: The geometry of the table

Let us try to understand this from the geometry of the table. We define unit vectors $\hat{h}$ and $\hat{u}$ as shown in the figure. The former joins the colored ball to the hole while the latter is from the centre of the white ball towards the center of the colored ball. Ideally, for there do be a good chance of a potting, the two unit vectors should overlap as much as possible.

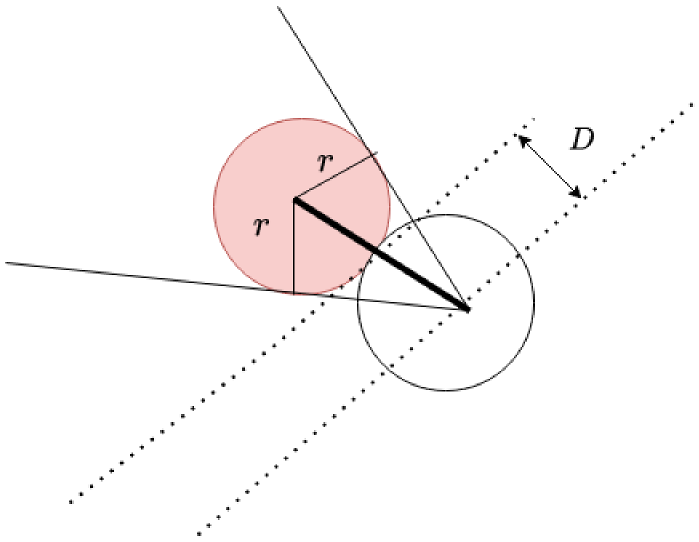

Figure 2: When the two balls are in contact

When the two balls are in contact, the distance between the two balls will be r. Now, we want to compute the exact location where the white ball should the colored ball, in case of no noise condition, at such a collision, we can be sure that the ball will be potted. Suppose, the unit vector from colored ball to hole is $\hat{h}$, when the white ball strikes the final direction of the colored ball should be along $\hat{h}$. We can use the properties of elastic collisions to determine the coordinates of the ideal point $\boldsymbol{p}$ by

$$\boldsymbol{p}_x = \boldsymbol{c}_x - \mu r \hat{i}^T \hat{h}$$
$$\boldsymbol{p}_y = \boldsymbol{c}_y - \mu r \hat{j}^T \hat{h}$$

Here, $\mu = 2$. ($\mu = 0$ will aim at the center and not the optimal point) Also, there will be a case when the incoming ball will just graze the colored ball. We should try to minimise these cases since in such a case, almost 0 velocity is imparted to the colored ball. The next figure explains all this well

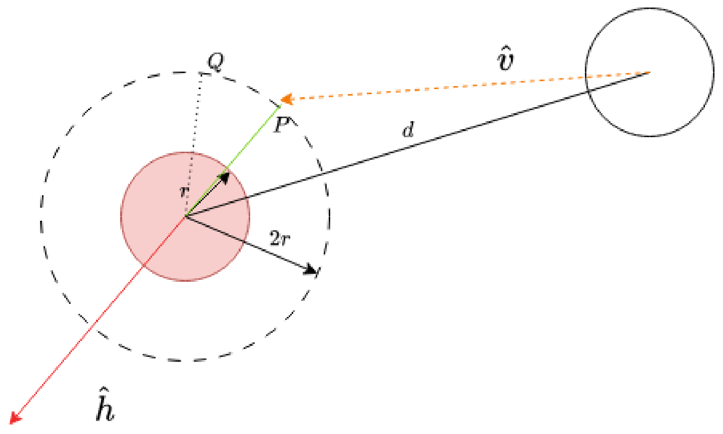

Figure 3: A summary of how to select the optimal point of collision

In the figure, P is the optimal point, while Q is the grazing point. Now, any point beyond Q is not going to be reached and hence we must make sure that we do not even try to go for such a point. We note that the distance between P and the center of the white ball is an increasing function as we start moving from P to Q. Hence we compute this distance and set a threshold on it to ensure that we are not aiming for bad points. Thus, looking at the above diagram, we want the white ball to move along $\hat{v}$. Sometimes, such a movement might not be possible and we return 'None' for such cases. Also, we have passed the parameter 'mu' which allows us to reuse this function if we want to directly aim at the centre of the colored ball (explained later on). This can be achieved by passing mu=0. Also note that in the diagram the point Q is the point where the tangent from the centre of the white ball to the circle of radius $2r$ with the colored ball at the centre.

2 Decision Time Planning

We are implementing decision time planning to decide which hole we should be aiming at and also which ball should we be trying to hit. First, lets see some functions that we are going to be using on the way in order to achieve this.

```
    def check_which_ball_is_closest_to_white(ball_pos):
    distances = {}
    white_location = numpy.array(ball_pos["white"])
    for i in range(1, 10):
        if ball_pos.get(i) != None:
            color_location = numpy.array(ball_pos[i])
            distance = get_distance(white_location, color_location)
            distances[i] = distance
    closest_ball = min(zip(distances.values(), distances.keys()))[1]
    closest_distance = distances[closest_ball]
    return distances, closest_distance, closest_ball
```

We use this function to check which ball is closest to the white ball. We are getting a dictionary which provides the distance of the white ball from each of the colored balls. This will help us to decide which ball we must aim at

```

def dict_of_angles(ball_pos, ball, hole_list):
    white_location = numpy.array(ball_pos["white"])
    color_location = numpy.array(ball_pos[ball])
    unit_vector_from_white_to_color = get_unit_vector(white_location, color_location)
    angdict = {}
    for i in range(0,6):
        hole_array = numpy.array(hole_list[i])
        unit_vector_color_to_hole = get_unit_vector(color_location, hole_array)
        dot_product_which_tells_goodness = numpy.dot(unit_vector_color_to_hole, \
                                                     unit_vector_from_white_to_color)
        angdict[i] = dot_product_which_tells_goodness
    return angdict
```

The above function returns the angles between the vectors $\hat{h}$ and $\hat{u}$ for every hole for a given ball. The best case is when the dot product is almost equal to 1 (possibility of straight shot) and hence dot product should be higher. We use it to decide which holes should be checked using the next state. We make use of both of these aspects while running the final loop, which we shall now show

```
        def action(self, ball_pos=None):
        ## Code you agent here ##
        ## You can access data from config.py for geometry of the table, configuration of
    the levels, etc.
        ## You are NOT allowed to change the variables of config.py (we will fetch variables
     from a different file during evaluation)
        ## Do not use any library other than those that are already imported.
        ## Try out different ideas and have fun!
          print("Number of balls remaining", len(ball_pos) - 2)
          closeness, _, _ = check_which_ball_is_closest_to_white(ball_pos)
          sorted_closeness = dict(sorted(closeness.items(), key = lambda x:x[1]))
          #print("Closeness:", closeness)
          #print("Sorted closeness:", sorted_closeness)
          count = 0
          for key in sorted_closeness:
              #print("Key:", key)
              angdict = dict_of_angles(ball_pos, key, self.holes)
              #print("Dictionary of dot products: ", angdict)
              sorted_angdict = dict(sorted(angdict.items(), key = lambda x:x[1], reverse=
    True)[0:3])
              #print("Sorted angle dictionary:", sorted_angdict)
              for yek in sorted_angdict:
                  if hit_ball_to_hole(ball_pos, self.holes, key, yek) != None:
                    angle = hit_ball_to_hole(ball_pos, self.holes, key, yek)
                    miter = 12
                    for k in range(0,miter):
                        force = 0.3 + (0.6 - 0.3)/miter*k
                        #print("Count is:", count)
                        count += 1
                        if len(self.ns.get_next_state(ball_pos, (angle, force), numpy.random
    .randint(0,1000))) < \
                            len(ball_pos):
                                print("Ball:", key, "Hole:", yek, "Angle:", angle, "Force:",
     force)
                                return (angle, force)
          if len(ball_pos) > 3:
            print("Random shot")
            random_key = random.choice(list(sorted_closeness))
            angle = hit_ball_to_hole(ball_pos, self.holes, random_key, numpy.random.randint
    (0,5), 0)
            return (angle , 0.5)
          else:
              print("Controlled random shot")
              random_key = random.choice(list(sorted_closeness))
              angle = hit_ball_to_hole(ball_pos, self.holes, random_key, numpy.random.
    randint(0,5), 0)
              return (angle , 0.5)
```

This is the code and we shall try to explain it line by line.

1. First, we order the colored balls in the order in which they are distant from the white ball, in ascending order. Thus, the closest ball has the starting position

2. Then, we iterate through all the colored balls on the table in the order we specified earlier.

3. For each ball, we consider the three best holes on the basis of dot product. We could have considered all the holes, but that will just lengthen the running.
4. We determine the optimal angle at which the white ball needs to be hit at so that the chance of success is highest. This is as per the geometrical calculations we showed in the first part. Now, we check if such an optimal angle is feasible. If yes, then this is the base angle which we try. If not, then we try some other hole.
5. If the angle is feasible, then we run a Monte Carlo Tree Search using the get next state function privided in utils.py. There are hyperparamaters inside this which we will explain later.
6. If the next state returned through simulation has lesser number of balls on the table as compared to the current state, we execute that action. Otherwise, we increase the force a little and try another simualtion.
7. It is possible that none of the simulations yield any potting of the balls. In such a case, we try to focus hitting the ball with some velocity and hence directly aim at the center of the coloured ball. The hope is that this will give rise to a new configuration wherein it might be easier to pot a ball on the next try

2.1 Hyperparameters

A massive amount of hyperparameter tuning had to be used inside this algorithm. We explain the important parameters and how we achieved them

miter = 12 Number of iterations for a given hole and ball. We have used **12**, which means 12 forces within the max and min range are tried out

min force = 0.3 Minimum force with which we are hitting the white ball to check for success

max force = 0.6 Minimum force with which we are hitting the white ball to check for success

Force for random shot = 0.5 Random shot means aiming at the centre of a colored ball in case we want to try for a better configuration. We apply a constant force in this case (obtained through trial and error). There is also a "controlled random shot" version, but we obtained the same optimum value of 0.5 for that also from trial and error

Threshold distance This distance multiplied by the thresholding weight is used to check the grazing condition. This distance d^* can be obtained by

$$d^* = \sqrt{d^2 - 4 \times r^2}$$

where d is the distance between the centres of the white and colored ball and r is the radius of the balls. This is a parameter for every state and is not constant or set by me throughout the program.

Thresholding weight = 1.05 For checking grazing conditions during the collision. Ideal value is 1, but we allow for some flexibility by which some tough shots get achieved. Set through trial and error.

2.2 Intermediate (failed but helpful) experiments

```
    def set_mu_and_force(goodness, distance=1000):
    goodness_threshold = 0.1 #Goodness_Threshold
    distance_threshold = 80#Distance_Threshold
    maximum_force = 0.75
    min_force = 0.62
    force_if_too_close = 0.6
    force_of_direct_hit = 0.8
    if goodness > goodness_threshold:
        mu = 2
        if distance <= distance_threshold:
            force = force_if_too_close
        else:
            force = numpy.minimum(maximum_force - (maximum_force - min_force)*goodness**1,
    1)
    else:
        print("Entering else loop")
        mu = 0
        force = force_of_direct_hit
    return mu, force
```

Had implemented a function which manually decided the force on the basis of dot product, however, it does not work out well in some specific situations. Hence, did not use it later on.