CS747-Assignment 3

Vansh Kapoor
200100164

High Level Idea

Firstly, using geometry, I determine the most appropriate angle at which the cue ball should be struck for a given state in order to maximize the probability of potting a specific ball. To achieve this, I identify the most suitable pocket for the target ball and calculate the angle at which the target ball should move to maximize its probability of being potted. Additionally, I identify the top three balls with the highest likelihood of being potted.

Subsequently, in the given state, I employ decision-time planning through tree search to identify the most suitable ball to pot and determine the force that should be applied to the ball. When there is more than one target ball remaining, I limit my tree search to a depth of 1. If only one target ball remains, then I extend the tree search to a depth of two, which increases the probability of potting a ball in at most two attempts.

To determine the estimated force to be applied, I calculate the distance between the cue ball and the target ball, taking into account the provided friction coefficient. I then employ a tree search to incrementally explore values above this calculated distance in small steps.

I followed a unique approach that utilizes **overlap between angle ranges** to identify the most suitable pocket for the target ball which gives excellent results!

Optimal Angle For Cue Ball

```
def tangent(striker,ball,radius):
    d=numpy.sqrt((striker[0]-ball[0])**2 + (striker[1]-ball[1])**2 )
    alpha=numpy.arcsin(2*radius/d)
    #remove

    theta=numpy.arctan((striker[0]-ball[0]+1e-12)/(striker[1]-ball[1]+1e-12))
    if(striker[1]<ball[1]):
        theta=theta-math.pi
    angles=[theta+alpha-(math.pi/2),theta-alpha+(math.pi/2)] #Will have to deal
    ↪  with [-1.3,-0.7] later

    return(angles)

def angle(ball,holes,hole_radius):
    t=[]
    for i in holes:
```

```
        d=numpy.sqrt((ball[0]-i[0])**2 + (ball[1]-i[1])**2 )
        alpha=numpy.arcsin(hole_radius/d)
        theta=numpy.arctan( (ball[0]-i[0]+1e-12)/(ball[1]-i[1] +1e-12) )
        if(ball[1]<i[1]):
            theta=theta-math.pi
        t.append([theta-alpha,theta+alpha])
    return t
```

The function `tangent(striker, ball, radius)` provides us with the range of angles representing the potential motion of the target ball after a collision with the cue ball.I considered a radius of 2 times the target ball's radius. By utilizing the tangents to this circle from the cue ball, I estimated the range of deflection angles for the target ball from its current position. The output of `tangent(striker, ball, radius)` provides us with the min and max values of these angles.

Meanwhile, `angle` gives us with the range of angles for the velocity required for the target ball to be potted in each hole from a list of holes with a radius of `hole_radius`. The `hole_radius` is implemented in a similar manner to `tangent(striker, ball, radius)`, where the radius here is equivalent to the hole radius. In this case, the output is a list whose length corresponds to the number of holes. Each element of the list provides the min and max angles at which the target ball could be potted in a specific hole.

I now find the top three balls with the maximum amount of overlap between these two ranges of angles, i.e., the range of angles for the velocity required for the target ball to be potted and the range of angles representing the potential motion of the target ball.

My input cue ball angle is the **mean angle of the overlap angle**. This provides **robustness to the Gaussian Noise** added. By providing this robustness, the algorithm performs well even in cases of decently high noise variance.

```
def control(theta,str,ball,radius):
    new=[]
    new.append(ball[0]+2*radius*math.sin(theta))
    new.append(ball[1]+2*radius*math.cos(theta))
    ang=numpy.arctan( (str[0]-new[0]+1e-12)/(str[1]-new[1]+1e-12 ) )
    if(str[1]<new[1]):
        ang=ang-math.pi
    return ang
```

The above function `control` gives the angle at which the cue ball has to be struck for a given desired angle `theta` for the target ball.

```
def overlap(ang1,ang2): #send striker angles in ang1
    section1=[]
    section2=[]
    a1=ang1
    len1=0
    len2=0
    for i in range(2):
```

```
    ang1[i]=ang1[i]%(2*math.pi)
    ang2[i]=ang2[i]%(2*math.pi)

if ang1[0]<ang1[1]:
    s11=ang1
    s12=[]
else:
    s11=[ang1[0],2*math.pi]
    s12=[0,ang1[1]]
if ang2[0]<ang2[1]:
    s21=ang2
    s22=[]
else:
    s21=[ang2[0],2*math.pi]
    s22=[0,ang2[1]]

if(len(s12)==0 and len(s22)==0):
    if (s11[0]< s21[1] and s21[0]<s11[1]):
        section1= [max(s11[0],s21[0]),min(s11[1],s21[1])]

elif(len(s12)==0 and len(s22)==2):
    if (s11[0]< s21[1] and s21[0]<s11[1]):
        section1= [max(s11[0],s21[0]),min(s11[1],s21[1])]
    elif (s11[0]< s22[1] and s22[0]<s11[1]):
        section1= [max(s11[0],s22[0]),min(s11[1],s22[1])]

elif(len(s12)==2 and len(s22)==0):

    if (s11[0]< s21[1] and s21[0]<s11[1]):
        section1= [max(s11[0],s21[0]),min(s11[1],s21[1])]
    elif (s12[0]< s21[1] and s21[0]<s12[1]):
        section1= [max(s12[0],s21[0]),min(s12[1],s21[1])]

else:

    section1= [max(s11[0],s21[0]),2*math.pi]
    section2= [0,min(s22[1],s12[1])]

if len(section1)!=0 :
    len1=section1[1]-section1[0]
if len(section2)!=0:
    len2=section2[1]-section2[0]

ovrlp=len1+len2

if ovrlp==0:
```

```
        return 0,(a1[0]+a1[1])/2

    elif len2==0:
        return ovrlp,(section1[1]+section1[0])/2
    else:
        return ovrlp,(section1[0]+section2[1]+2*math.pi)/2
```

The above function gives us the range value of overlap between two intervals of angles. This function implementation is not so trivial due since we are considering there is a wrap around at $2*\pi$. The output also provides us the mean angle of the overlap, which gives us the angel of deflection of the target ball after collison with the cue ball.

Decision Time Planning Under Uncertainty

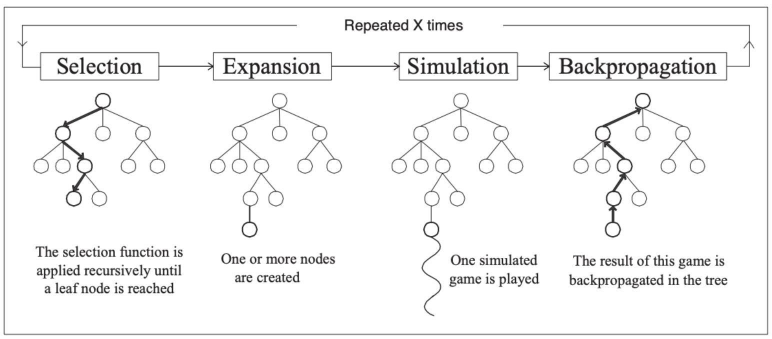

Figure 1: Outline of a Monte-Carlo Tree Search.

We use decision time planning using **tree search** to find the optimal angle due to noisy angle input to the cue ball. This method is very effective in-case of releasing information regarding the optimal action in case of uncertainty.

I limit my tree search to a depth of one initially, exploring a range of forces for a given input angle to the cue ball, calculated as described in the previous section. By iterating over **2-3 episodes**, I greedily choose my force.

However, when only one target ball remains, I extend the tree search to a depth of two. This extension enhances the likelihood of successfully potting the ball within just two attempts. This proved to be quite useful especially in edge cases where none of the actions could result in the ball being potted. I **discounted** the reward obtained in the future time-step by a factor of **0.25**. Thus, by utilizing and discounting the **future time-step reward**, I established an effective policy for the cue ball.

```
for jj in range(len(array_1)):
                                r=numpy.random.randint(1,100)
                                next_state2=self.ns.get_next_state(next_state,
                                ↪  (best_angle_1, array_1[jj]), r)
```

```
                        sum+=(len(next_state)-len(next_state2))*0.25
                        if((len(next_state)-len(next_state2))==1):
                            break
```

Parameter Tuning For Finding Force

```
if maxim==3:
            c1=0.2
            c2=8
            c3=0.05
            k=3

for i in range(maxim):

            bst_ball=ball_pos[new[i,2]] #hole[bst,2] is a key always
            bst_hole=self.holes[int(new[i,1])]
            possible_angles=tangent(striker,bst_ball,ball_radius) #actual

            d=(dist(bst_ball,striker)/115.34)*0.1+c1
```

I spent a considerable amount of time fine-tuning the parameters for the ranges of forces applied to the tree search algorithm. Initially, I utilized the friction coefficient value provided in `config.py`. I calculated the distance achievable by applying a force of **0.1** to the cue ball, which resulted in a distance of approximately **116** units for the given friction coefficient. I determined the minimum force required for the cue ball to cover the distance between itself and the target ball. I iterated over a finite number of values with small incremental steps in force.

My **intermediate experiments** that made be realize the importance of tree search, specifically the advantage of tree search for a **depth of two** when only a single target ball is remaining. Using a depth of two proved to be quite advantageous since the cue ball took an action that had very high probability of potting the single target ball in at max two tries.

The geometry approach followed by me gave be a considerable advantage in selecting the best balls as well the appropriate angle of collisions for the cue ball. I used decision time planning, which involved parameter tuning only to find the appropriate force input for the cue ball. I considered steps of size 0.05 for force with the starting value as
(min_force to reach the target ball) + (margin=0.2) +(step_size)*iter