Prateek Garg
20D070060
prateekg@iitb.ac.in

Programming Assignment 1

CS747: Foundations of Intelligent and Learning Agents, Autumn 2023

2025-02-21

Task 1

The implementations of UCB, KL-UCB, and Thompson Sampling Algorithms.

1.a UCB Algorithm

State Variables: $counts[i]$, $values[i]$ $\forall i$, $total\_counts$

- $counts[i] := c_i$ denotes the number of times, arm i has been pulled.
- $values[i] := v_i$ denotes the empirical average of reward observed from a particular arm.
- $total\_counts := t$ denotes the total number of pulls from the algorithm.

Pull Step: returns an arm to be played

1. Increment t by 1.
2. for i in $[1, .., .n]$, return i if $c_i == 0$
    - Important for well defined ucb as well as c_i is 0, so it means it should be explored first
3. Calculate ucb for each arm i using $ucb_i = v_i + \sqrt{\frac{2\log(t)}{c_i}}$
4. return $i = \arg\max_i ucb_i$

Reward Step: takes $reward := r$, arm index i

1. Increment c_i by 1
2. update v_i using (new) $v_i = \frac{c_i - 1}{c_i} v_i + r/c_i$
    - We could have just save the cumulative rewards, but it might lead to overflow

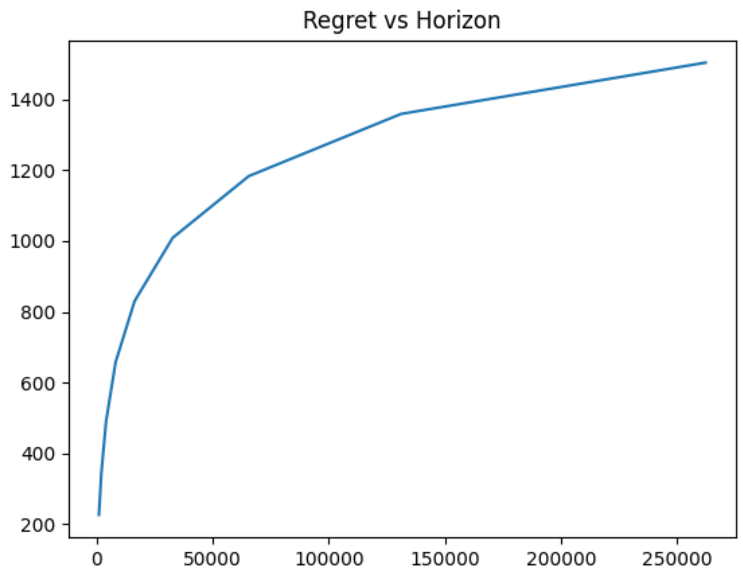


Figure 1: Regret accumulated by UCB Algorithms

1.b KL-UCB Algorithm

Implementation of KL-UCB is same as UCB algorithm, except value of $ucb[i]$ is different and calculated numerically. Here we illustrate the subroutine to calculate $ucb[i]$ for KL-UCB algorithm.

```python
def kl_bern(p,q):
    eps = 0.00000001 # For numerical stability
    p[p==0] += eps
    p[p==1] -= eps
    q[q==0] += eps
    q[q==1] -= eps
    return p*np.log(p/q) + (1-p)*np.log((1-p)/(1-q))

def rhs(count,t, c=0):
    te = math.log(t)
    if te == 0: te = 0.00000001 # For numerical stability
    return (te + c*math.log(te))/count

def get_ucb_kl(counts,values,total_counts,prec=0.00000001,c=0, max_iter=30):
    num_arms = len(counts)
    rhss = rhs(counts,total_counts,c)   # calculating rhs for each arm
    q = np.zeros(num_arms)              # placeholder for solutions
    l = np.zeros(num_arms)
    l+= values                          # Lower estimate of the solutions
    u = np.ones(num_arms)               # Upper estimate of the solutions

    for _ in range(max_iter):
        q = (u+l)/2               # Candidate Solutions
        kls = kl_bern(values,q)   # KL-div for each arm
        NEG_MASK = kls < rhss     # indices where lower estimate needs to be updated
        l[NEG_MASK] = q[NEG_MASK]                                # updating lower estimate
        u[np.logical_not(NEG_MASK)] = q[np.logical_not(NEG_MASK)] # updating upper estimate
        NOT_UPDATE_MASK = abs(u-l) < prec    # checking if every value is within required precision
        if all(NOT_UPDATE_MASK) : break      # Terminate the loop
    return q
```

- kl_bern function returns kl-divergence between two Bernoulli distributions parameterised by p and q, if p and q are vectors, it does so parallely.
- rhs function returns rhs value for each arm i
- get_ucb_kl returns the ucb value calculated using Newton Raphson method.
  - Has two parameter: prec(precision) and max_iter for Newton Raphson

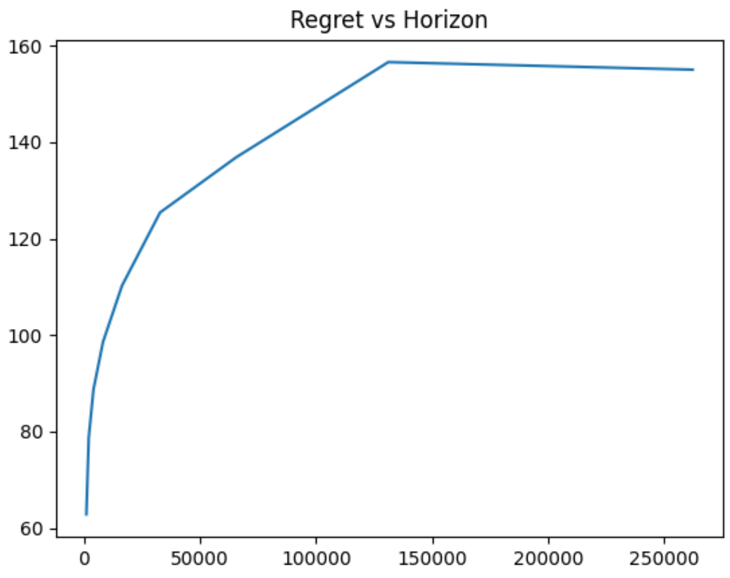


Figure 2: Regret accumulated by KL-UCB Algorithm

1.c Thompson Sampling Algorithm

State Variables: $success[i], failures[i]$

- $success[i] := s_i$ denotes the number of pulls on i^{th} arm gave reward 1

- $failures[i] := f_i$ denotes the number of pulls on i^{th} arm gave reward 0

Pull Step: returns the arm to be played

1. $\mu_i \sim \beta(s_i + 1, f_i + 1) \forall i$
   (a) done parallely using numpy library function
2. return $\arg\max_i \mu_i$

Reward Step: takes $reward := r$, arm index i

1. if $r == 0 : f_i = f_i + 1$
2. if $r == 1 : s_i = s_i + 1$

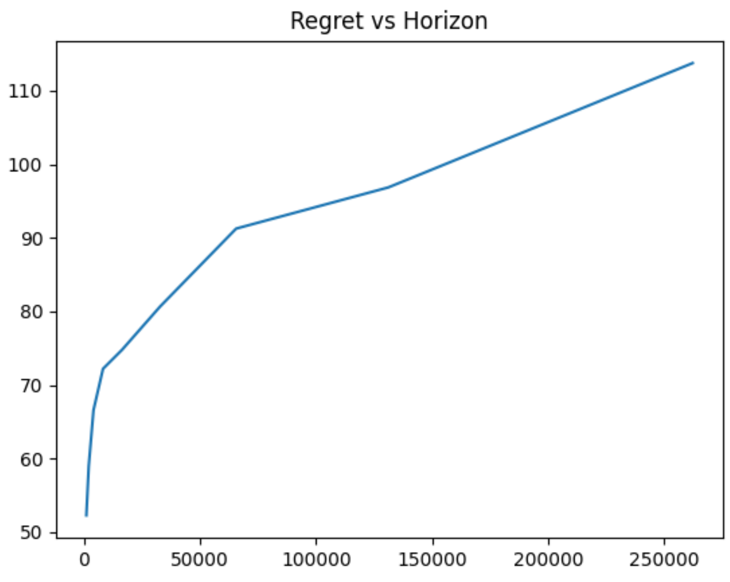


Figure 3: Regret accumulated by Thompson Sampling Algorithm

Task 2

2.a Differences between means of arms and UCB Algorithm

Regret of UCB algorithm is,

$$R_T = O\left(\sum_{a:p_a \neq p^*} \frac{1}{p^* - p_a} \log(T)\right)$$

for instance $[p1, p2], p_1 = p^*$, it reduces to,

$$R_T = O\left(\frac{1}{p_1 - p_2} \log(T)\right)$$

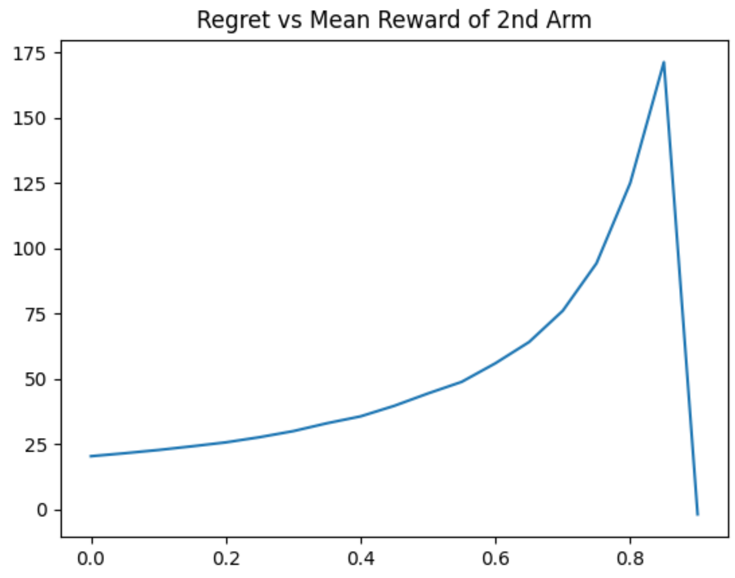
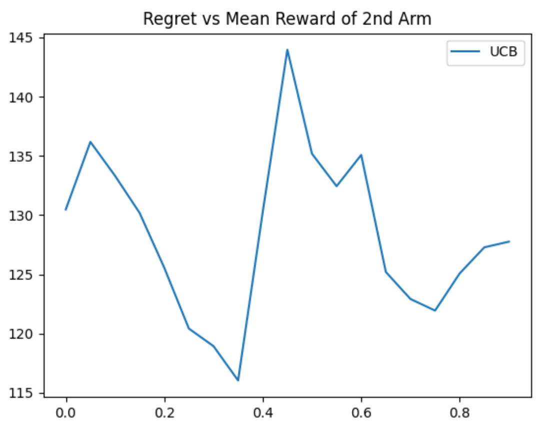


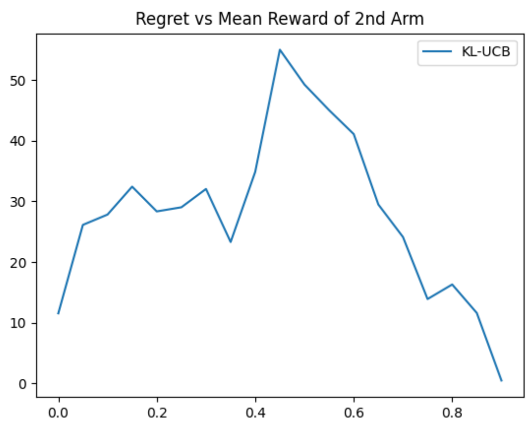
Figure 4: Regret accumulated by UCB Algorithm on instance $[p_1, p_2]$, where $p_1 = 0.9$,p_2 (on x-axis varies 0 to 0.9 (both inclusive) in steps of 0.05 over the horizon of 300000)

Fixing $T = 30000$, we plot the regret of UCB algorithm and indeed we see that the regret increases as p_2 increases but plummets to 0 when $p_1 = p_2 = 0.9$, since now both arms are the optimal arms.

2.b UCB and KL-UCB Algorithms vs difference of means of arms



(a) UCB



(b) KL-UCB

Figure 5: Regret accumulated by two algorithms on instance $[p_1, p_2]$, where $p_1 - p_2 = 0.1$ and p_2(on x-axis varies 0 to 0.9 (both inclusive) in steps of 0.05 over the horizon of 300000)

We see that the regret of UCB algorithm is *almost* constant$(130 \pm 11\%)$, which is in agreement with the regret bound seen in 2.a as $p_1 - p_2 = 0.1$ is held constant.
Regret of KL-UCB algorithm is,

$$R_T = O\left(\sum_{a:p_a \neq p^*} \frac{p^* - p_a}{KL(p_a, p^*)} \log(T)\right)$$

for instance $[p1, p2], p_1 = p^*$, it reduces to,

$$R_T = O\left(\frac{p_1 - p_2}{KL(p_2, p_1)} \log(T)\right)$$

We see that the regret of KL-UCB algorithm varies a *lot*$(25 \pm 120\%)$, since we have $KL(p_1, p_2)$ term in denominator which changes with every value of p_2 as shown in 6

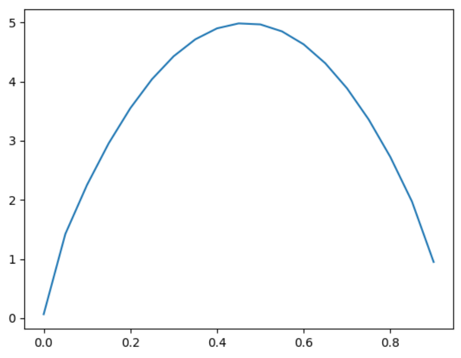


Figure 6: $\frac{p_1 - p_2}{KL(p_2, p_1)}$ as function of p_2, $p_1 - p_2 = 0.1$

This also explains why we observe a peak around $p_2 = 0.45$

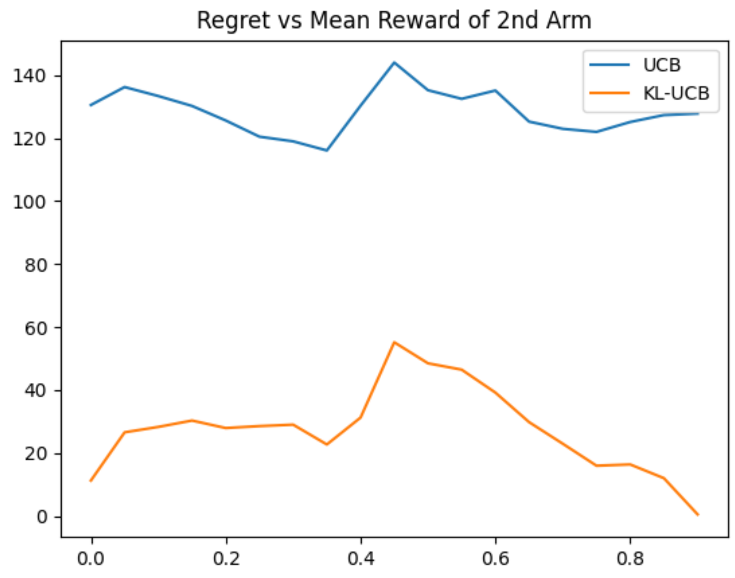


Figure 7: Comparison of regret accumulated by two algorithms on instance $[p_1, p_2]$, where $p_1 - p_2 = 0.1$ and p_2(on x-axis varies 0 to 0.9 (both inclusive) in steps of 0.05 over the horizon of 300000)

It is also observed that regret accumulated by KL-UCB is far lesser than the UCB, which is also expected because KL-UCB has tighter bound on regret.

Task 3

After experimenting with almost every algorithm, we found that Thompson Sampling is easiest to adapt and gives lower regrets than any other algorithm. So the goal is to design a Thompson style algorithm to Faulty Bandit case.
Let E be the event that fault occurs, and we $f = P(E)$, $P(r|E) = 0.5$

3.a Modifying the Posterior Update

Posterior in the reward step of Thompson Sampling is,

$$P(x_i|r) = \frac{P(r|x_i) \cdot P(x_i)}{\mathbb{E}_{x_i \sim P(x_i)}\left[P(r|x_i)\right]}$$

$x_i \sim \beta(s_i+1, f_i+1)$, $r|x_i \sim Bern(x_i) \implies P(r=1|x_i) = x_i$ which results in

$$x_i|r \sim \beta(s_i+1+r, f_i+1+1-r)$$

In case of the faulty arm, $P(r|x_i) = 0.5\cdot f + x_i \cdot (1-f)$, we get

$$\begin{aligned} P(x_i|r) &= \frac{P(r|x_i)}{\mathbb{E}_{x_i\sim P(x_i)}\left[P(r|x_i)\right]}\cdot P(x_i) \\ &= \frac{0.5\cdot f + x_i\cdot(1-f)}{0.5\cdot f + \mathbb{E}_{x_i\sim P(x_i)}\left[P(r|x_i,\neg E)\right]\cdot(1-f)}\cdot P(x_i) \end{aligned}$$

assuming $x_i \sim \beta(\alpha_i,\beta_i) \implies \mathbb{E}_{x_i\sim P(x_i)}\left[P(r=1|x_i,\neg E)\right] = \frac{\alpha}{\alpha+\beta}$,

$$\begin{aligned} P(x_i|r=1) &= \frac{0.5\cdot f + x_i\cdot(1-f)}{0.5\cdot f + \frac{\alpha_i}{\alpha_i+\beta_i}\cdot(1-f)}\cdot P(x_i) \\ &= \frac{0.5\cdot f}{0.5\cdot f + \frac{\alpha_i}{\alpha_i+\beta_i}\cdot(1-f)}\cdot P(x_i) \\ &+ \frac{\frac{\alpha_i}{\alpha_i+\beta_i}\cdot(1-f)}{0.5\cdot f + \frac{\alpha_i}{\alpha_i+\beta_i}\cdot(1-f)}\cdot\left(\frac{\alpha_i+\beta_i}{\alpha_i}\cdot x_i\cdot P(x_i)\right) \end{aligned}$$

Looking closely we see that this is a mixture of Beta Distributions,

$$P(x_i|r=1) = \frac{0.5\cdot f}{0.5\cdot f + \frac{\alpha_i}{\alpha_i+\beta_i}\cdot(1-f)}\cdot f_{\alpha_i,\beta_i}(x_i) + \frac{\frac{\alpha_i}{\alpha_i+\beta_i}\cdot(1-f)}{0.5\cdot f + \frac{\alpha_i}{\alpha_i+\beta_i}\cdot(1-f)}\cdot(f_{\alpha_i+1,\beta_i}(x_i))$$

where $f_{\alpha,\beta}(x_i) = P(x_i)$, if $x_i \sim \beta(\alpha_i,\beta_i)$ similarly, can be shown that,

$$P(x_i|r=0) = \frac{0.5\cdot f}{0.5\cdot f + \frac{\beta_i}{\alpha_i+\beta_i}\cdot(1-f)}\cdot f_{\alpha_i,\beta_i}(x_i) + \frac{\frac{\beta_i}{\alpha_i+\beta_i}\cdot(1-f)}{0.5\cdot f + \frac{\beta_i}{\alpha_i+\beta_i}\cdot(1-f)}\cdot(f_{\alpha_i,\beta_i+1}(x_i))$$

Combining the two we get,

$$P(x_i|r) = \frac{0.5\cdot f}{0.5\cdot f + \frac{\beta_i*(1-r)+\alpha_i*r}{\alpha_i+\beta_i}\cdot(1-f)}\cdot f_{\alpha_i,\beta_i}(x_i) + \frac{\frac{\beta_i*(1-r)+\alpha_i*r}{\alpha_i+\beta_i}\cdot(1-f)}{0.5\cdot f + \frac{\beta_i*(1-r)+\alpha_i*r}{\alpha_i+\beta_i}\cdot(1-f)}\cdot(f_{\alpha_i+r,\beta_i+1-r}(x_i))$$

Looking even more closely, we get that,

$$P(x_i|r) = P(E|r) * f_{\alpha_i,\beta_i}(x_i) + P(\neg E|r) * f_{\alpha_i+r,\beta_i+1-r}(x_i)$$

In general it will be intractable to use this update rule, since the next update will result in 3 components and so on. So on t^{th} time we will have 2^t components, thus memory requirement will be $O(2^T)$ where T is horizon, whereas normal Thompson Sampling has $O(1)$ memory requirement.
To tackle this, we observe that this posterior can be interpreted as "Do not update the belief is event E occurs, but update otherwise" and indeed if we were to use the exact posterior for the sampling step, we would be sampling over these t events$(E_1, E_2, ..., E_t)$ first, choose the corresponding component and then sample from it. So, in our approach, we sample one mixture component and discard the other,

$$e \sim Bern\left(\frac{0.5\cdot f}{0.5\cdot f + \frac{\beta_i*(1-r)+\alpha_i*r}{\alpha_i+\beta_i}\cdot(1-f)}\right)$$

$$P(x_i|r) = \begin{cases} f_{\alpha_i,\beta_i}(x_i), & \text{if } e=1 \\ f_{\alpha_i+r,\beta_i+1-r}(x_i), & \text{otherwise} \end{cases}$$

which would be equivalent to re-using the sample of $E_1, E_2, ..., E_t$ from previous timesteps.
It is worth noting that we take a different prior of $x_i = 0.5\cdot f + (1-f)\cdot y_i, y_i \sim \beta(\alpha_i,\beta_i)$ a similar expression is obtained for the posterior update rule. Here x_i should be thought of as the actually *observed* mean of i^{th} arm and y_i is corresponding *true* mean. This reduces the support of x from $[0,1]$ to $[f*(0.5), 1-0.5*f]$
Also, for this very reason, if we use Thompson Sampling without using this prior information at all, we still achieve very competitive results, since it recovers the observed mean asymptotically.

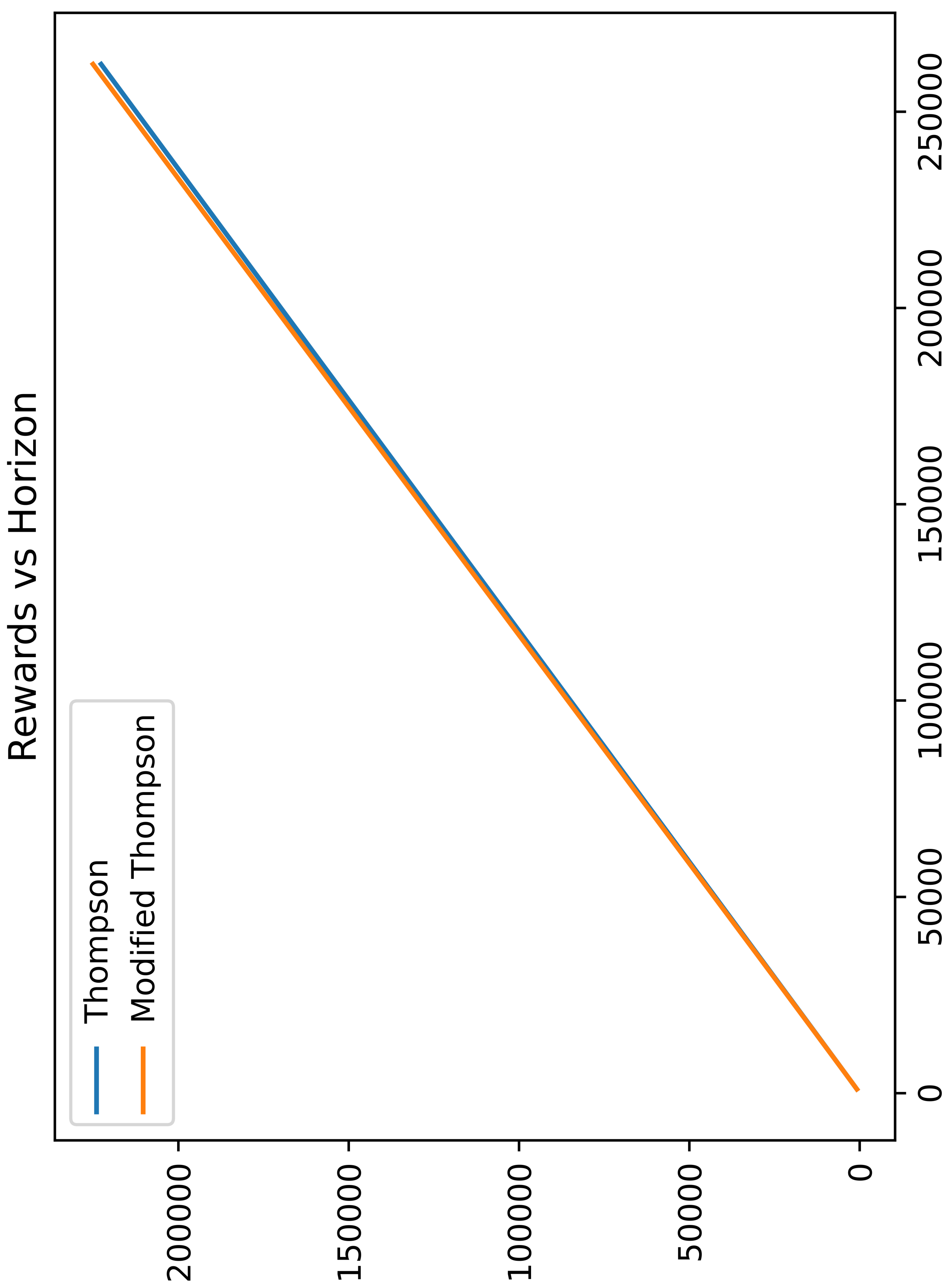


Figure 8: Comparison of rewards accumulated by Thompson and Modified Thompson Sampling, Modified Thomspon marginally performs better.

Task 4

Unlike the case in task 3 where we don't observe if the fault occured or not, here we observe the set which was used, so we can update the *beliefs* accordingly. Let $x_{i,a}$ denote the mean of arm i from set a. Then the expected reward when arm i is pulled is $= \mathbb{E}_a\left[x_{i,a}\right]$ which in this case reduces to $\frac{x_{i,0}+x_{i,1}}{2}$.
State Variables: $success[i,a], failures[i,a]$

- $success[i,a] := s_{i,a}$ denotes the number of pulls on i^{th} arm of set a gave reward 1
- $failures[i,a] := f_{i,a}$ denotes the number of pulls on i^{th} arm of set a gave reward 0

Pull Step: returns the arm to be played

1. $\mu_{i,a} \sim \beta(s_{i,a}+1, f_{i,a}+1) \forall i, a$
   (a) done parallely using numpy library function
2. return $\arg\max_i \frac{\mu_{i,0}+\mu_{i,1}}{2}$

Reward Step: takes $reward := r$, arm index i, set pulled a

1. if $r == 0 : f_{i,a} = f_{i,a} + 1$
2. if $r == 1 : s_{i,a} = s_{i,a} + 1$

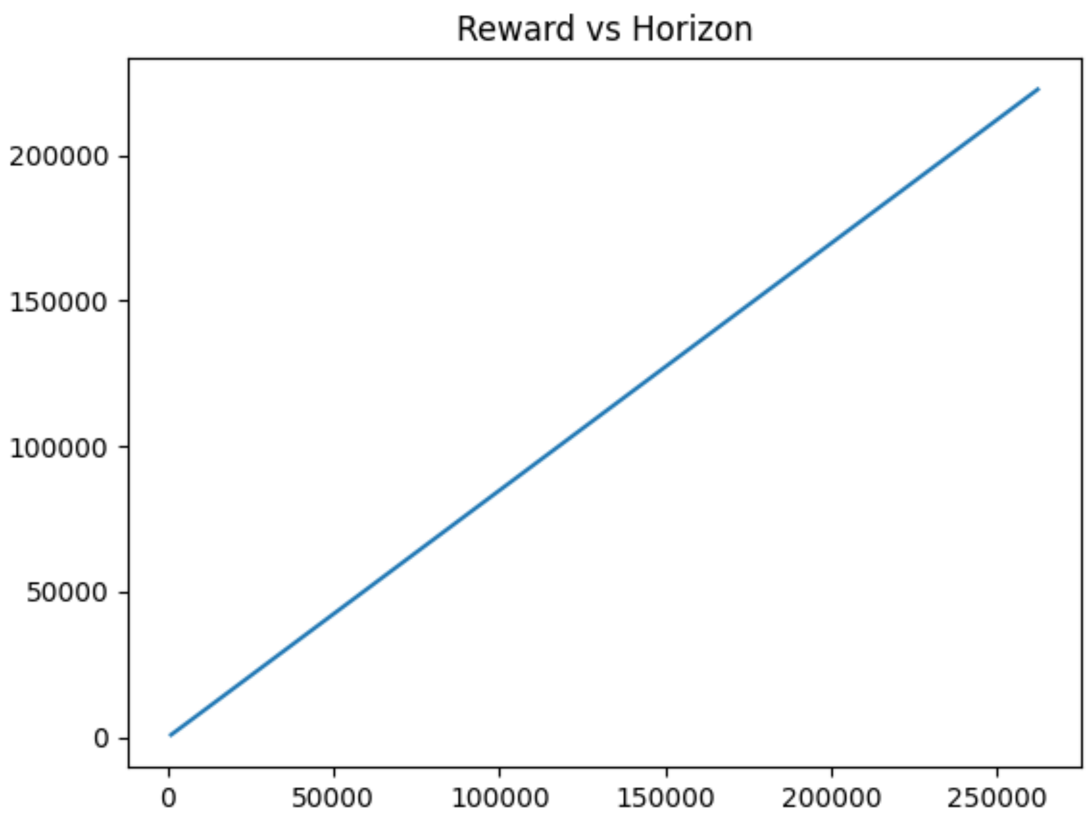


Figure 9: Reward accumulated by our Algorithm

Prateek Garg
20D070060
prateekg@iitb.ac.in

Task 1

The implementation of Value Iteration, Howard's Policy Iteration and Value function using linear programming algorithms.

1.a Value Iteration

We initialise the value vector(V^0) with all zeros, and apply Bellman Optimality Operator(B^*) – defined by Transition function T, Reward function R – repeatedly until convergence.
For convergence we consider $||V^{t+1} - V^t|| < \texttt{tol}$, $\texttt{tol}$ = 1e-7 and we also consider that $||V^* - V^t|| < \texttt{tol}$, to enforce this, we have

$$\gamma^N \cdot ||V_0 - V^*|| < \texttt{tol}$$

since $||V^* - V^N|| < \gamma^N \cdot ||V_0 - V^*||$

$$N \log \gamma + \log ||V_0 - V^*|| < \log(\texttt{tol})$$

$\because V^0 = 0$

$$\frac{\log(||V^*||/\texttt{tol})}{\log(1/\gamma)} < N$$

for numerical stability, $\epsilon = 1e-7$

$$\frac{\log(||V^*||/\texttt{tol})}{\epsilon + \log(1/\gamma)} < N$$

Since, from our definition V^t has converged to a value($||V^{t+1} - V^t|| < \texttt{tol}$) we check, if that value is indeed optimal within tolerance.

$$\frac{\log(||V^t||/\texttt{tol})}{\epsilon + \log(1/\gamma)} < N$$

We also define `MAX_ITER=1E6` for maximum number of iterations to run before terminating, to ensure algorithm terminate always.
Observation: Value iteration converge very slow if $\gamma = 1$ as expected.

1.b Howard's Policy Iteration

Starting from a random policy π^0, we repeatedly apply policy improvement step and get better policy. Since we need to do policy evaluation in each step, There are two methods – Value Iteration with Bellman operator(B^π), inverting the matrix $I - \gamma \cdot T^\pi$. Matrix Inversion is faster for small number of states but Value Iteration is faster for large number of states. For our implementation, we use matrix inversion. Matrix multiplication becomes non-invertible for $\gamma = 1$, so we set the values of terminal states to 0 and for the rest we calculate using matrix inversion of truncated matrices which doesn't include terminal state transitions.
Observation: Howard's PI is the fastest algorithm out of all 3, and thus our default algorithm as well.

1.c Linear Programming

We use `PuLP` solver to encode this as linear program. The objective is to maximise $-\sum_s V(s)$ subject to some inequality constraints and some equality constraints described below.
Inequality Constraints:

$$V(s) \geq \sum_{s' \in S} T(s, a, s')\{R(s, a, s') + \gamma V(s')\}, \forall s \in S, a \in A$$

Equality Constraints:

$$V(s) = 0, \forall s \in S_{terminal}$$

Equality constraints are required in case $\gamma = 1$

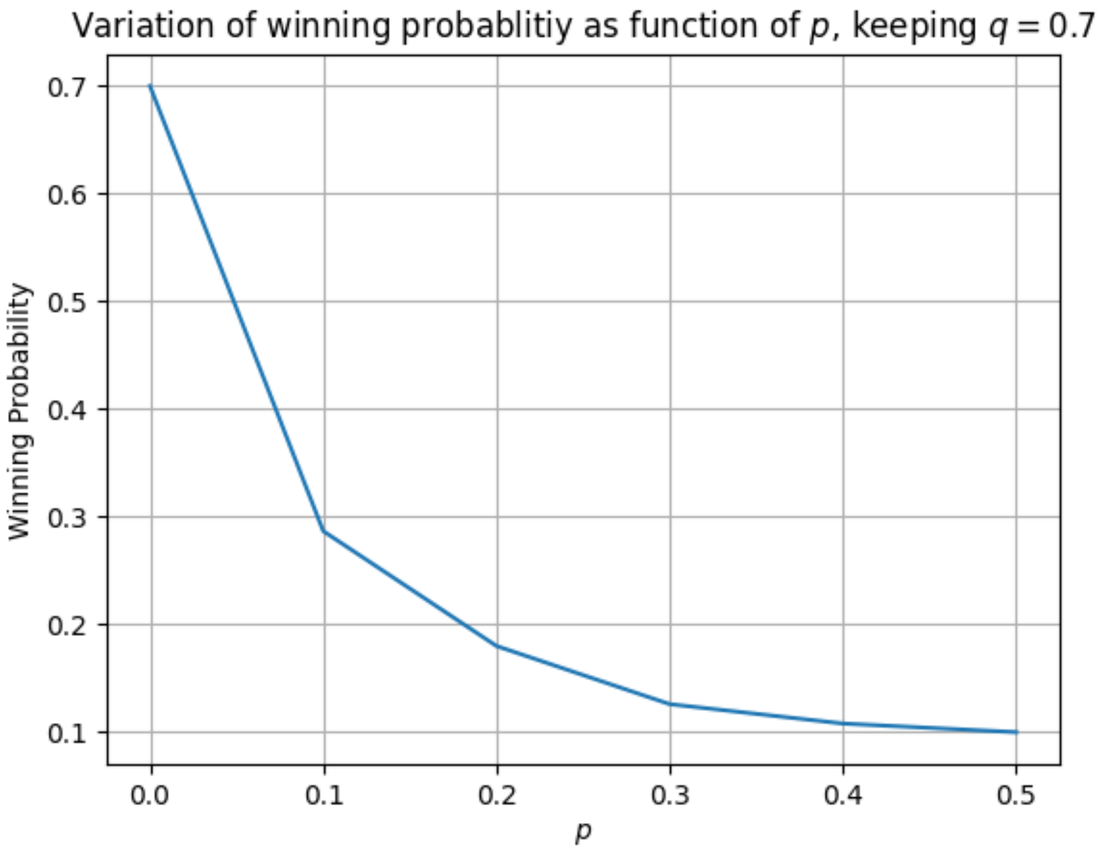
Task 2

Designing MDP for the football game.

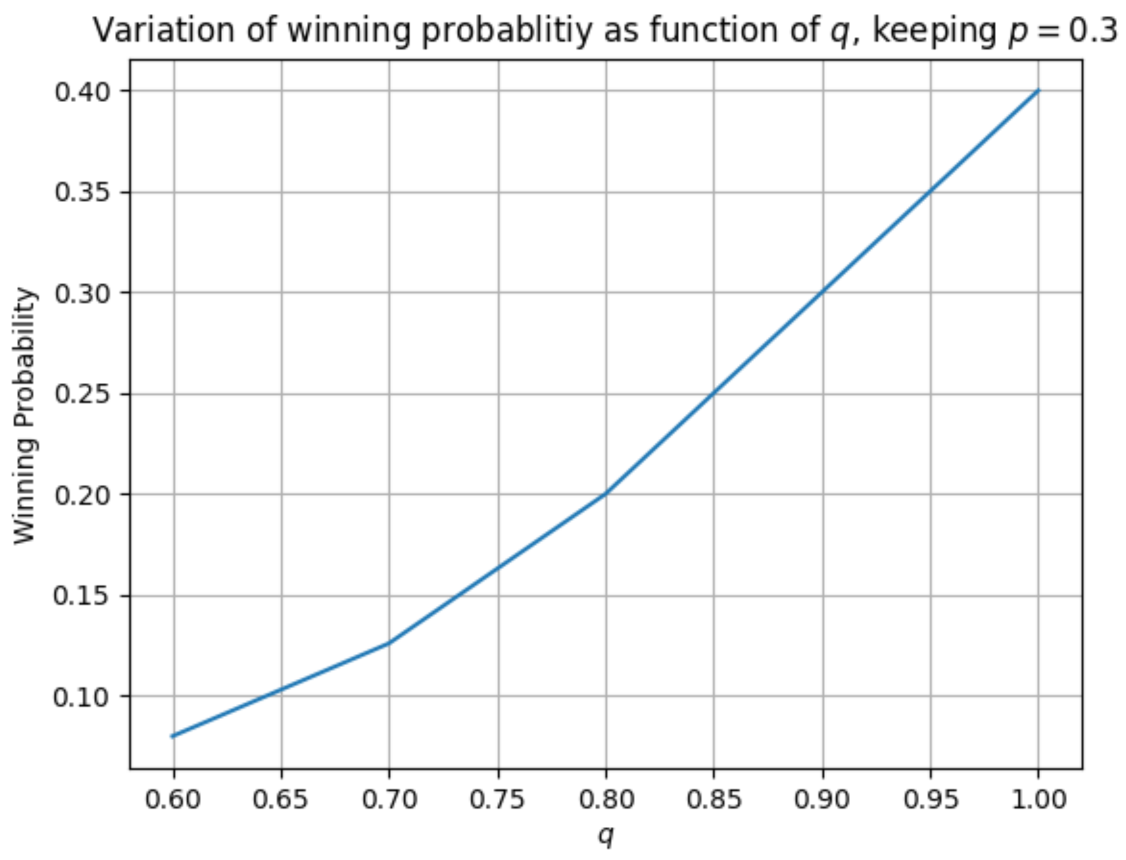
2.a Formulation

We map each state $[B1, B2, R, P]$ to a number s which varies from $[1, 8192]$, we have 2 extra states $\{0, 8193\}$. 0 is the terminal state, which the last state before game ends without a goal. If game ends with a goal, the terminal state is 8193. The transition function is defined by the rules of the game and the corresponding probabilities, once the players decides an action $\{0...9\}$. The reward function is zeros for almost all transitions except when transitioning to $s = 8193$, when the reward is 1. The expected reward of each state gives us the value function which is also the probability of winning(since $\mathbb{E}[r] = p\dot{1} + (1-p) \cdot 0 = p$) given the starting state is that particular state. This is all done in `encoder.py`

2.b Comparison & Inferences



Since as p increases the probability of failure of an attempted movement increases($\{2p, 0.5+p, p\}$ are probabilities that the game ends depending upon the case), thus the probability of winning decreases.



Since as q increases the probability of success of an attempted pass or goal increases (for passing $\{0.5 * (q - 0.1 * \max(|x_{B1} - x_{B2}|, |y_{B1} - y_{B2}|)), q - 0.1 * \max(|x_{B1} - x_{B2}|, |y_{B1} - y_{B2}|),\}$ are probabilities that the attempted pass succeeds, $\{q - 0.2 * (3 - x_{distance}), 0.5 * (q - 0.2 * (3 - x_{distance}))\}$ depending upon the case), thus the probability of winning increases.

Prateek Garg
20D070060
prateekg@iitb.ac.in

Programming Assignment 3
CS747: Foundations of Intelligent and Learning Agents, Autumn 2023

2025-02-21

Task 1

```python
def anglexy(x,y):
    """returns angle(-1,1) of point x,y"""
    if x<=0 and y<=0: theta = np.arctan(x/y)
    elif x<=0 and y>0:theta = np.pi - np.arctan(-x/y)
    elif x>0 and y>0:theta =  - np.pi + np.arctan(x/y)
    elif x>0 and y<=0:theta = - np.arctan(-x/y)
    return theta/np.pi

def length_xy(v):
    return np.linalg.norm(v)

class Agent:
    def action(self, ball_pos=None):
        vec_w = np.array(ball_pos["white"]) # getting the coordinates of cue ball

        # removing the cue ball from position list
        del ball_pos["white"]
        del ball_pos[0]

        ball_vecs = np.array(list(ball_pos.values())) # Getting the vectors of balls
        hole_vecs = np.array(self.holes)              # Getting the vectors of holes

        hole2ball_vecs = ball_vecs[:,None,:] - hole_vecs[None,:,:] # pairwise vectors between ball
            and holes

        hit_points_vecs = ball_vecs[:,None,:]+2*self.ball_radius*(hole2ball_vecs/np.linalg.norm(
            hole2ball_vecs, axis=-1, keepdims=True))
        cue2hit_points_vecs = hit_points_vecs - vec_w[None,None,:]
        cue2ball_vecs = ball_vecs[:,None,:] - vec_w[None,None,:]
        # print(.shape)

        DOT_PRODUCT_TOL = 0
        cos_similarity = (-hole2ball_vecs * cue2ball_vecs).sum(axis=-1)/(np.linalg.norm(
            hole2ball_vecs, axis=-1) * np.linalg.norm(cue2ball_vecs, axis=-1))
        hittable_holes = cos_similarity > DOT_PRODUCT_TOL

        valid_holes_dict = {}
        best_b_so_far = 0
        best_h_so_far = 0
        best_dist_so_far = 100000
        LAMBDA = 0
        for b in range(ball_vecs.shape[0]):
            ball_h = []
            ball_h_dist = []
            for h in range(hole_vecs.shape[0]):
                if hittable_holes[b,h]:
                    ball_h.append(h)
                    ball_h_dist.append(np.linalg.norm(hole2ball_vecs[b,h])+LAMBDA*(1 -
                        cos_similarity[b,h]))
            if len(ball_h_dist) > 0:
                bmindist = min(ball_h_dist)
                ball_h_i = ball_h_dist.index(bmindist)
                if bmindist < best_dist_so_far:
                    best_dist_so_far = bmindist
                    best_b_so_far = b
                    best_h_so_far = ball_h_i
                valid_holes_dict[b] = ball_h_i
            else:
                valid_holes_dict[b] = None
        EPS = 0.95
        coin_flip = np.random.binomial(1,EPS)
        if coin_flip == 1:
            b = best_b_so_far
            h = best_h_so_far
        else:
            b = np.random.randint(ball_vecs.shape[0])
            h = valid_holes_dict[b]
            if h == None:
                h = np.random.randint(hole_vecs.shape[0])

        b_vec = ball_vecs[b]
        h_vec = hole_vecs[h]

        hit_vec = b_vec+2*self.ball_radius*(b_vec - h_vec)/np.linalg.norm((b_vec - h_vec))
        target_x,target_y =hit_vec - vec_w
        angle = anglexy(target_x,target_y)

        FORCE_FACTOR = 1.1
```

```
75          force = FORCE_FACTOR*(length_xy(hit_vec - vec_w) + (length_xy(h_vec - b_vec)/config.
                ball_coeff_of_restitution))/(960*1.414)
76          return (angle, force)
```

1.a Basic Idea

The basic idea is to hit a selected ball at an angle that it gets potted in the selected hole. To achieve this we employ basic vector calculus to find the unit vector from the hole to ball and then scale it by 2*ball_radius, this will give us the vector of cue ball –with respect to the target ball– when it collides with the target ball. Adding the vector of target ball will give us the vector w.r.t origin and to find the angle to be specified we can get the vector w.r.t initial position of cue ball.

1.b Selecting Ball and Hole

Not every hole will be directly coverable by the cue ball. In particular, any hole which subtends a acute angle at the target w.r.t to line joing target ball and cue ball. So we filter out these holes(line 30-32). This also gives rise to a tunable paramter `DOT_PRODUCT_TOL` which by the argument above should be 0 but in general it is difficult to pot ball at hole at near-right angles.
For every call to `action()`, we search for best hole-ball pair(lines 39-50) using a metric `dist(h,b)+LAMBDA*(1-`$\cos\theta$`)` where θ is the angle described above. For `LAMBDA=0` this reduces to finding ball hole pair which are closest to each other. `LAMBDA>0` emphasis that the hole should nicely lined up as well and balances between the 2 notions of goodness of hole-ball pair.
Additionally, we also track best hole per ball in a dictionary. With probability `1-EPS` we randomly samplpe a ball instead of using best hole-ball pair found, and use it's best hole. This balances exploration and exploitation.
Finally, we have an additional parameter `FORCE_FACTOR` which scales the force which is directly proportional to the distance between cue ball and target plus the distance between target and hole scaled by coefficient of restitution.