MACHINE DATA AND LEARNING **ASSIGNMENT 2 PART 3**

LINEAR PROGRAMMING TO SOLVE MDP

TEAM NUMBER 2:

KUSHAGRA AGARWAL 2018113012

SHREEYA PAHUNE 2018113011

PROCEDURE FOR MAKING A MATRIX

We created a MDP which was basically a union *Dragon's Health ([0,25,50,75,100])*, *Number of Arrows left ([0,1,2,3])*, and *Stamina([0,50,100])*. We used these **5*4*3=60 states** to signify S in our solution.

The A matrix has the dimensions=(len(S),len(X)), which we found to be 60 and 100 respectively. The way we got 100 is as follows:

For **Dragons Health=0**, there were only **12 terminal** states possible.

For **Dragons Health >0** and <=100, there were **22** (State:Actions) possible for each health.

The 22 were as follows:

From the state where arrows =0 we have the following possibilities:

Stamina=0: Recharge

Stamina =50: Dodge, Recharge

Stamina =100: Dodge

From the state where arrows =1 we have the following possibilities:

Stamina =0: Recharge

Stamina=50: Shoot, Dodge, Recharge

Stamina =100: Shoot, Dodge

From the state where arrows =2 we have the following possibilities:

Stamina =0: Recharge

Stamina=50: Shoot, Dodge, Recharge

Stamina =100: Shoot, Dodge

From the state where arrows =3 we have the following possibilities:

Stamina =0: Recharge

Stamina=50: Shoot, Dodge, Recharge

Stamina =100: Shoot, Dodge

So total = **12 + 22*4 = 100**. Hence the dimensions of the matrix A are **(60,100)**

Hence the matrix has 60 rows and 100 columns. The rows depict the *dragons health, your arrows and stamina*. For example, the first row is the one where the dragons health, stamina as well as arrows are all 0. The second row will have the dragon's health and number of arrows as 0, but the stamina would be 50 and so on.

All terminal states can only perform the **NOOP** action with Probability of 1, and hence all the start states(which are also the end states in this case) are simply given a **value 1** in the A-matrix.

If a player has arrows **a**, stamina **s** and the dragons health is **h**. Then on shooting an arrow the A matrix has the row with (h,a,s) or has index= **12*h + 3*a +s**, and a column corresponding to the **{State:Action}** index, and it will have the value **1**(as this is the start_state). If the arrow hits the dragon(which it would with a probability of 0.5) then, the New state is **h-1, a-1, s-1**. The row corresponding to these values and the same column as previously computed will have a value of **0.5*(-1)** as this is a reached state. Also if the arrow missed the dragon the state would become **h,a-1,s-1**, again with a probability of 0.5 and hence would hold the value of **-0.5**.

PROCEDURE FOR FINDING POLICY

The various matrices required were as follows:

A: Mentioned above

R: Reward Matrix. The reward for all actions apart from NOOP were set to step_cost=-5(team_number 2). The reward for NOOP actions which are valid in the terminal states is set at 0. The dimensions of the R matrix are (100,1)

X: Expected number of times each action is taken. This was set to be a cvxpy variable which will be optimised by the Linear Programming solver. Dimensions (100,1)

ALPHA: Starting state vector. This signifies the probability of the various states being the starting state and in our case this was equal to 1 for the state number 59, as we start only with this state with health=100, arrows=3 and stamina=100.

The objective function was :- **Max(R*X)**

There were 2 constraints:

1) **X >=0:** all values are expected to be positive as no negative number of actions can be taken.
2) **A*X= ALPHA:** This shows that the solution is consistent with the initial conditions.

We use cvxpy module's .Problem(objective_function, linear_constraints) and .solve() to get the optimal value of vector X

After having found X, we wish to find the optimal policy as well. We divide the X into the various states from where it starts and chose that **value of X[iter] which was the maximum** for that state. The action corresponding to that X[iter] is chosen to be the optimal policy for that state.

If a state has 3 actions: x1 for "SHOOT", x2 for "DODGE" and x3 for "RECHARGE", and it turns out that x3>x1>x2, then the action corresponding to x3, i.e, "RECHARGE" is chosen to be the optimal policy for that state.

MULTIPLE POLICIES

1) **STEP COST/ REWARD FUNCTION:** A change in the step cost from -10 to -5 changes the optimal policy and this behaviour is expected as a higher value of step cost will make the agent to finish faster whilst a smaller step cost relaxes the agent.

2) **SAME VALUE FOR X[iter] FOR MULTIPLE ACTIONS**: If 2 different actions end up having the same value for X[iter] then choosing which one will be the optimal policy is bogus, hence we end up with more than one optimal policy with the same objective function.

3) **START STATES:** What if we had started from the terminal states, then we would not have got any optimal policy whatsoever. Also starting from any intermediate state also changes the optimal policy.