# Programming Assignment 2

Harshil Naik (190010030)

## *Task 1:*

Transition probabilities and rewards are read into a NumPy array of size ***S*A*S***.

**Value Iteration:**

```
Algorithm:
    $V_0 = |S|\ sized\ 0\ array$
    $Tolerance = 1e-12$
    $Repeat:$
        $For\ s\ \in S:$
            $V_{t+1}(s) \leftarrow \max_{a \in A} Q(s,\ a,\ V_t)$
        $t \leftarrow t+1$
    $Until\ \left||V_{t+1} - V_t|\right|_2 > Tolerance$
where $Q(s, a, V_t)\ is\ the\ action\ value\ function\ corresponding\ to\ value\ function\ V_t$
```

**Howard's Policy Iteration:**

```
Algorithm:
    $policy = |S|\ random\ array\ with\ values\ in\ range\ [0, |A|-1]$
    $Tolerance = 1e-12$
    $Repeat:$
        $For\ all\ improvable\ states\ s\ with\ best\ improvable\ action\ a\ and\ Q(s,a)$
        $\geq V(s) + Tolerance\ switch\ to\ a$
    $Until\ no\ such\ state\ \&\ action\ exist$
```

**Linear Programming:**

```
Algorithm:
    $Objective: Minimize(\sum_{s \in S} V(s))$
    $subject\ to\ \forall s \in S, \forall\ a\ \in A:$
        $V(s) \geq \sum_{s' \in S} T(s, a, s')\{R(s, a, s') + \gamma V(s')\}$
```

**Assumptions:**
In the evaluation of the Value function (V), the matrix is invertible or there exists a unique solution to the set of Bellman Equations. Also in the case of Howard Policy Iteration, it was observed that the policy switches across two such policies due to machine precision. So I have included tolerance of 1e-12 for an action to be called an improvable action. A similar value of tolerance is used for value iteration. It was also observed that the code sometimes got stuck in a loop where the same policy was used multiple times over and over, but the value did not get updated. Hence, I added a loop break condition that brought the loop to an end if the new policy was the same as the old one.

**Observations:**
Value Iteration works the fastest for almost all the test cases, followed by Howard's Policy Iteration and then Linear Programming Solver. (Hence Value Iteration has been kept as the default solver)

## *Task 2: Anti Tic Tac Toe*

The ATTT is modelled as an MDP with several end states (those with full grid or with arrangements such that either of the players wins). Depending on the state file provided, we are assigned a player (either 1 or two), and a set of possible states for that specific player.
We assign the indices of the states as the states of the MDP, and the corresponding probability (taken from the environment, i.e fixed policy of the other player) is also stored.
We also form two functions - ***isWin*** and ***isFull*** that check whether a given state result in a win for either of the players or is a draw due to all the space being filled.
These functions help in finding the endstates for the MDP, by iterating through the start state and the possible actions for a particular state.
The newly found endstates are added to the state array, which is now an exhaustive list of all possible states of the MDP, and now can be used to form the policy file for the planner.py file.

*Run the algorithm using Value Iteration.*

***References:***

https://harshil3004.gitbook.io/reinforcement-learning/
https://towardsdatascience.com/how-to-code-the-value-iteration-algorithm-for-reinforcement-learning-8fb806e117d1