EE566 Report

Tic-Tac-Toe

Manas Wadhwa 11940670

Ayushi Thakur 11940220

Porash Chauhan 11940840

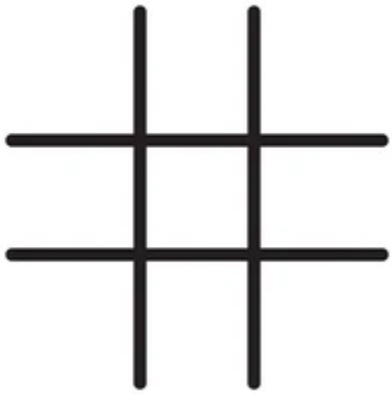
Ritik Dhanore 11940370

Akshar Sharma 11940090

Description of the game

Tic Tac Toe is a classic two-player game played on a 3x3 grid. The objective of the game is to line up three of your symbols in a row before your opponent, either horizontally, vertically, or diagonally.

When the game starts the board is empty and looks like the image below:-



The symbol "X" is given to one player, while the symbol "O" is given to the other player. By selecting an open area on the grid, each player places their symbol once. The game is won by the first person to line up three of their symbols. The game is a tie if every square is filled but no one has won.

So the 3 outcomes of the game are:-

- Win
- Lose
- Tie

Modeling MDP

MDP consists of a State, Action, Reward, Transition probabilities, and Discount factor.

States: Every cell in the board of Tic-Tac-Toe will either be 0,1 or 2. 0 here means empty (not played), 1 means player 1, 2 means player 2. Thus a state is simply a string of 9 numbers where each number represents the entry in that cell.

Action: Given a state, action would mean the set of playable cells i.e cells with entry 0. For eg:

Rewards: The following is a list of the Tic Tac Toe rewards:

The prize is +1 if player 1 triumphs.
The prize is -1 if player 2 triumphs.
The prize is zero if the game is a tie.

Transition probabilities: For the game of tic tac toe,

If Player 1 makes a move, the resulting state is determined by placing a 1 in the chosen cell following which player 2 will play a move, this is the next playable state of player 1. *More details later...*

Discount Factor: set to 0.9

So we can write MDP as a tuple: (S, A, P, R, gamma) where:

- S is the set of possible states
- A is the set of possible actions
- P is the set of transition probabilities

- R is the reward function
- Gamma is the discount factor

Hence in Tic-Tac-Toe, we have:

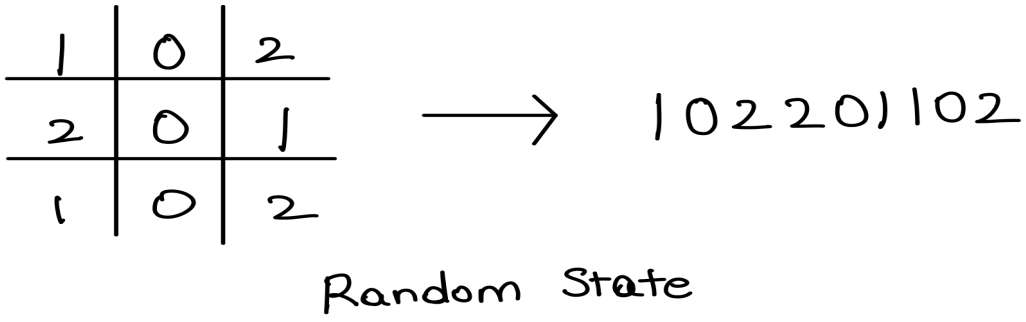
- S={all possible 3*3 matrices with “1”, “2”, or “0“}
- A={all possible unoccupied cells is the current state}
- P(s, a, s’)= p
- R(s)=+1 if player 1 wins, -1 if player 2 wins, and 0 otherwise.
- gamma= 0.9

State

We are planning to map the board configuration as one state. So different board configurations will mean different states. Also, we add three more terminal states to this set of board configurations namely win, lose, and tie.

As described above two players take turns to play the game. Let’s assume without loss of generality that we are learning to find the optimal response policy of Player 1 against a fixed policy of Player 2.

State Representation



Here, Player 1 fills the emtpy cells with 1 while player 2 fills them with 2. Empty cells are by default filled with 0's. Hence the starting state of the game will be represented as 9 0's i.e 000000000

If we think naively, we will find that even in a board that is 3 x 3 there are 3^9 possible states according to this way of defining the states. However, actually, only some of these states are achievable in an actual game. We can simulate the game between the two players to find all possible achievable states by player 1 and player 2. Here is the code that we wrote to find these states.

```
def play(current, p_id):

    for i,val in enumerate(current):

        if val == 0:
            temp = copy.deepcopy(current)
            temp[i] = p_id
            #check if this is an end state or not
            if isend(temp,1) or isend(temp,2):
                return
            else:
                if p_id ==2:
                    p1.add("".join(map(str, temp)))
                    play(temp,1)
                else:
                    p2.add("".join(map(str, temp)))
                    play(temp,2)
```

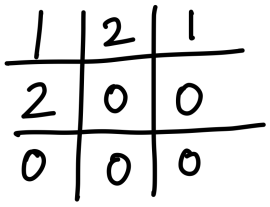
It's a DFS recursive tree traversal of the game to find the possible states of player 1 and player 2 and store them in an array that can be retrieved later.

State Transition

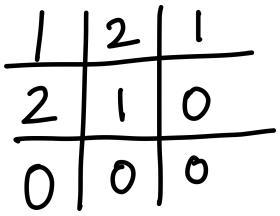
We are planning to find the optimal policy of player 1 in response to the **fixed** policy of player 2. **Transitions of player 1 from one state to another are not independent of the actions of player 2.** After player 1 plays a move, player 2 will also play its move, which will result into the new state of player 1

unless the game has not already been ended by the previous move of player 2. Here is an example to show the state transitions in detail.

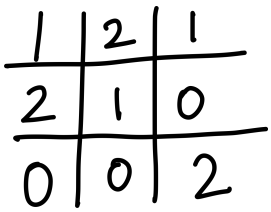
Consider the current state of Player 1 as shown below:



This can be mapped as **s1** = **121200000**. Player 1 can choose between the following actions (empty cells) {5,6,7,8,9} to play its mark. Say, it chooses to play at 5. New state will look like this **s2** = **121210000**:



Now player 2 has the options to mark 2 at positions {6,7,8,9}. Suppose Player 2 chooses to mark 9 with a probability p. New state will look like this **s1'** = **121210002**:



Now, this is when player 1 will play its move. Hence, this is the new state of player 1. Or, we can also write this transition from s1 to s1' as follows:

Transition: **s1 5 s1' 0 p**

This means that Player 1 played action 5 from state 1 and ended up at state s1' with a probability of p. This is because after player 1 played action 5, player 2 played the move of action 9 with probability p resulting is state s1'. This is how the effect of player 2 can be modelled into the state transitions of player 1. Here the 0, signifies the reward. Since, s1' is not an end/terminal state, the reward for the move is 0. Rewards are 1 for win, 0 for tie and -1 for loss.

Moving forward, we can model Player 1 as a MDP as follows:

(1) States: All possible states of player 1 as found above.
(2) Transitions with probability and reward
    (a) Following above approach, we can simulate all moves between player 1 and player 2 to find all the transition probabilities between all the states of player 1 along with their rewards.
(3) We can set the gamma (discount factor) suitably to 0.9

Having the MDP, we can proceed forward to use the value-iteration algorithm to find an optimal reponse policy to the set fixed policy of player 1. Subsequently, we can also iteratively try to set the fixed policy of player 2 to be the previous best policy of player 1 and find the optimal reponse to this new policy of player 2. We will end this process, when the best response to the policy of player 2 is itself. It would be interesting to see if this can be achieved in the game of tic-tac-toe. Next, we delve into some details of how the value iteration algorithm finds the optimal policy given a MDP.

Optimal Policy through Value Iteration

Value Iteration is a special case of policy iteration and it does a single iteration of policy evaluation at each step and then for each state, takes the maximum action value to be the estimated state value.

We need to compute the optimal state-value function V(s) for all the states s and then use the V* function to compute its optimal policy.

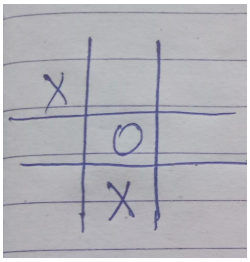
The algorithm is defined as follows:

- State-Value function V(s) is initialized to 0 for all states s.
- Repeat until convergence reaches:
    - For each state s, update the state-value function using Bellman equation:

        V(s)=max_a{sum_s'{P(s, a, s')[R(s')+gamma*V(s')]}}

        where a is an action, s' is the next state and P(s, a, s') is the probability of transitioning from s to s' through action a.
    - If the changes in V(s) across all the states is smaller than some particular small threshold, terminate the algorithm.

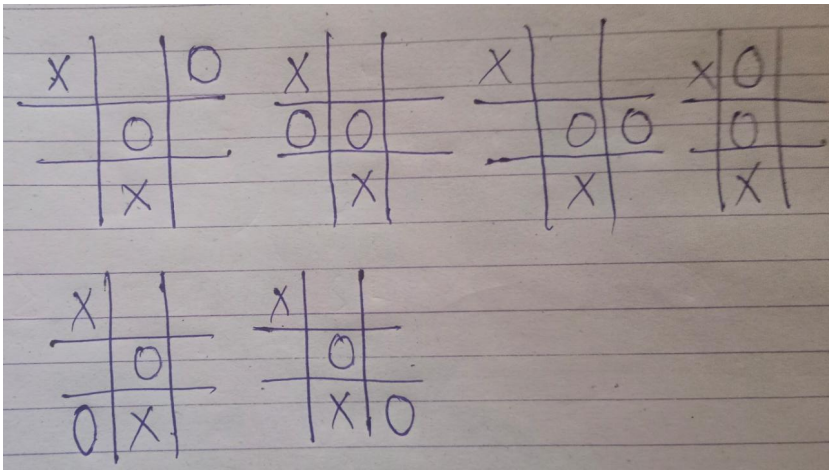
For example: Assume the current state as:



Vector representation: {1,0,0,0,2,0,0,1,0}

Now from the state space we have 6 possible actions to be taken as “O” as 6 cells are empty in the board. So after these actions, the next state set looks like this:

Vector representation:
{{1,0,2,0,2,0,0,1,0},{1,0,0,2,2,0,0,1,0}
,{1,0,0,0,2,2,0,1,0},{1,2,0,0,2,0,0,1,0}
,{1,0,0,0,2,0,2,1,0},{1,0,0,0,2,0,0,1,2}}



Now we need to find the value of each possible action by summing over the resulting states s' and their probabilities P(s, a, s') and then find the maximum of it.
V(s) for all states and all actions are calculated for this example.

After V(s) function is computed, we can compute the optimal policy for each state, which is the action that maximizes the value function as follows:

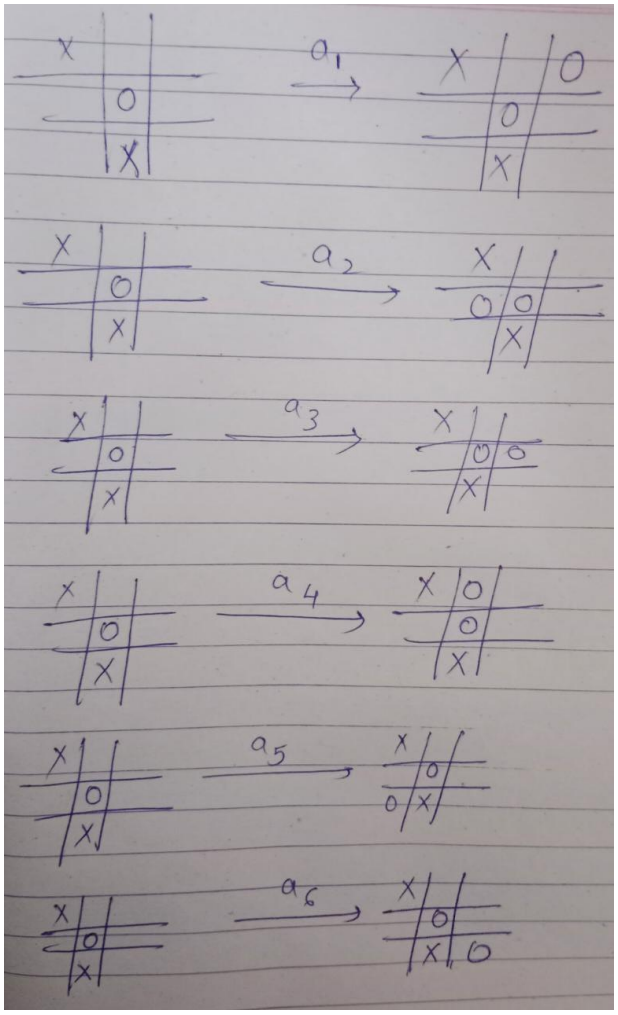
- For each state s, we need to choose an action a which maximizes the value of V*(s, a) where:

  V*(s, a) = sum_s'{P(s, a, s')[R(s')+gamma*V(s')]}

  This means that at each state, we choose an action that leads to the state with the highest value in the next time step.

So continuing the example, now our task is to calculate the V*(s,a) for each action at the given state and take the action which maximizes the value function. Hence calculating V*(s,a) for all these actions:

And going with the actions which maximize the value function.
We need to repeat this process for all states afterwards.



Optimal policy thereafter to maximize the expected cumulative reward from any given state:

- If there is a move that can win the game, take that move
- If the opponent has a move that can win the game, block that move.
- Otherwise, take the move which leads to the state with the highest value according to the V* function computed by value iteration.

In tic-tac-toe, if both players play optimally then the game will always end in a tie, so the optimal policy guarantees at least a tie.

4. Existing policies

One of the existing policies is to use the MINIMAX algorithm.
The algorithm calculates the minimax value of each state by recursively evaluating each move and the states that arise. The maximum value a player can reach in a state is known as the minimax value, and assuming their opponent plays optimally.
The following formulae can be used to determine a state's minimax value:

1. If the state is a terminal state (i.e., a player has won, lost, or the game has ended in a tie), then the minimax value is the reward associated with the state:

$$V(s) = \text{reward}(s)$$

2. If the state is not a terminal state, and it is the player's turn to move, then the minimax value is the maximum of the minimax values of the resulting states, assuming the opponent plays optimally:

$$V(s) = \max_{s' \text{ in successors}(s)} V(s')$$

3. If the state is not a terminal state, and it is the opponent's turn to move, then the minimax value is the minimum of the minimax values of the resulting states, assuming the player plays optimally:

$$V(s) = \min_{s' \text{ in successors}(s)} V(s')$$

In the above equation successors(s) means the set of states which can be reached from ‘s’ by making a legal move.

The move that results in the state with the highest minimax value is then chosen to decide the best course of action.
Let's see the below example and understand the algorithm