Task 5: 3D Tic-Tac-Toe Agent

Ashwani Kumar Kamal

***Abstract*— The purpose of this document is to show how a simple AI can be made unbeatable using simple sample serching techniques. Main focus would be on implementation of the minimax algorithm on games playable by 2 players (TicTacToe in our case). Game environment used was the GymTicTacToe Environment which can be found here. The AI lists all the possible states of game at any current point asd takes the move which favours its win probability.**

I. INTRODUCTION

II. PROBLEM STATEMENT

Part 1: You have to write an agent to play the 2-D Tic-Tac-Toe game using Minimax algorithm. Minimax algorithm basically is just an exhaustive search of your game space. You'll be using the gym-tictactoe environment. A skeleton for player-vs-player agent is provided at here.
Part 2: You will now have write an agent to play 3-D Tic-Tac-Toe. It will be harder to use the Minimax algorithm for this part as the search space is much larger. You will have to look into ways of optimising Minimax algorithm. If you want, you may implement more advanced methods like Q-Value Learning and Genetic Algorithms. Feel free to contact us if you have any trouble in getting the environment setup

III. RELATED WORK

Studied about the Minimax algorithm, alpha beta pruning

IV. INITIAL ATTEMPTS

Downloaded the repository given and ran the example skeleton. Went through the environment (all the classes, constants, methods etc). Restricted the space to 3 x 3 for solving part 1. Got an idea that I'll have to change the environment a bit for returning correct scores (end states). Went through the 3 part Minimax algorithm article given on Geeks for Geeks.

V. FINAL APPROACH

A. Changing the sample and end state conditions

Made necessary changes so that playable area is strictly restricted to 3 x 3 cells only, accordingly changed the end-state condition checking in $check()$ method. Also made changes in $available\_actions()$ method (the most vital function to our cause in this task). Also had to change the return value of $action()$ method to something better. Earlier the reward element of return tuple contained the value of winning party ('1' or '2'), I changed it to return -10 if '1' (HumanAgent played by X) wins and +10 is '2' (AIAgent played by O) wins and 0 if there is a draw (check if available_actions is empty or not). $\_result$ parameter has the value 10 for O wins, -10 for X win, 'None' for game in progress, 0 for draw. Also $depth$ variable has been employed to track the level or complexity of current move. The end-state condition returned is either -10 + $depth$ or 10 - $depth$, reason being that for the AI to force a draw, we need it to make that move which has higher depth (running the game for longer time). More detailes in reference.

B. Implementing Minimax

Important point to note is that $action()$ method needs the information of current round being played, current player mark, current board (which is global throughout the program). Hence we have two options: either make a copy of board each time we invoke Minimax (to find the chain of moves and select the best one) or do everything in the same board and undo our moves after concluding best move. As the first one is space expensive, I went with the latter. Everything is controlled using the $world$ parameter of main class, hence all we need to do is change the $world$ array to it's original form after recursion. Also the current round needs one undo.

```
# Undo action
state[int(action[0])][int(action[1])] = 0
board._round -= 1 #Undo round
```

According to the built of the environment, game ends when one wins or a move occurs which has already been made (happens after $available\_actions()$ is empty- a draw happens), hence an extra move has been played after $turns$ variable (contains how many turns have been played till now) becomes 10.
Rest is the actual implementation of Minimax which was fairly easy to do.
The implementation happens in two steps: $bestMove()$ function first receives the arguments corresponding to AI's turn which then calls the $minimax()$ function with appropriate parameters. Since I decided to go for AI's turn being second (O) so each time the functions are invoked the player's mark have been well taken care of. Undoing round is essential to our cause as it raises an Error if player's mark doesn't match with the current round being played.

C. Work on 3D Tic-Tac-Toe

Started with 3D Tic Tac Toe with 2 planes ($3*3*2$ sample space) made necessary changes in environment. Running Minimax algorithm directly will leave you with nothing but vain as the sample size is too much large and without any optimisation techniques, the algorithm will either run out of memory or it will take a considerable chunk of time

(probably of the order of hours) to come up with the first move. I didn't had enough time to go through DQN or Q value learning algorithm. But I think I do know what the basic outline is- we make a self learning model which tracks its mistake and adds in its database, much like deep learning.

VI. Results and Observation

The AI is working fine and as expected for $3 * 3$ Tic-Tac-Toe board but work is still needed for the $3 * 3 * 3$ one.

VII. Future Work

Following methods should supposedly make the AI a lot more faster than it is right now.

- Alpha Beta pruning
- Memoising end-states- next step would be memoising all states
- Invoking the $\_check()$ method only when number of X and O are multiples of 3

CONCLUSION

A simple algorithm like Minimax with the right implementation can make an unbeatable AI for games played by 2 people. However as the complexity of the game increases, we have to disfavour the use of Minimax. It is an exhaustive search of the whole sample space and will not perform well for large sample. In such situations we should opt to making somehthing which learns using neural networks and deep learning.

References

[1] Sebastian Lague "Algorithms Explained – minimax and alpha-beta pruning", Youtube, 2018

[2] Akshay L. Aradhya "Minimax Algorithm in Game Theory", GeeksforGeeks, 2021

[3] Jason Fox, "Tic Tac Toe: Understanding the Minimax Algorithm", Never Stop Building LLC, 2013