Aerial Robotics Kharagpur Task Round Documentation Task 5 :

Nikhil Giri

***Abstract*—This Documentation gives a summary of how I approached task 5 given to me during ARK Software Selection Round**

From the time when I was first given these tasks to the time i am writing this documentation I have learned and explored a lot of things. I learnt a lots of algorithms and also tried implementing them. These algorithms are essential for drones to judge and correct their path on basis of algorithms they encounter.

In this task we create a AI for Tic-Tac-Toe game suing the Minimax Algorithm. For General 2D Tic-Tac-Toe a simple implementation does the job. In 3D Tic-Tac-Toe the exhaustive search won't work so we optimize and implement it.

I. INTRODUCTION

Computers are better at decision making because they have inhuman capacity to crunch numbers.Best ways to test this is to make computer play games. More complicated the game, more numbers and calculations will be needed to find the best move. In this task we have to create an AI for both 2D Tic-Tac-Toe and 3D Tic-Tac-Toe.

II. PROBLEM STATEMENT

The task is divided into two parts.

Part 1: In this part we have to use openAI gym environment for 3D Tic-Tac-Toe and write an 'AI' agent inside it for computer to play. For this, use Minimax Algorithm. Minimax algorithm basically is just an exhaustive search of your game space. A skeleton for player-vs-player agent is provided at to us.

Part 2: Once we are done with 2D Tic-Tac-Toe, Now we have to create a agent for 3D Tic-Tac-Toe. The search space increases vastly in case of 3D Tic-Tac-Toe and thus basic Minimax application is not enough we have to use various and other techniques to optimize it and get quicker results.

III. RELATED WORK

For this task, I learnt Minimax algoritm and Alpha-Beta pruning. I implemented both of them.I also learnt Q-Table and implemented it but the result was sub-par on comparison with the Minimax agent. Further, I spent days learning about DQN but could not complte learning it. So it was not implemented.

IV. INITIAL ATTEMPTS

First off, I downloaded the repo with 3D Tic-Tac-Toe gym environment. That day and most of the next day went in carefull study of the gym environment and its different functions. Once I was sure that I got the inner workings of the environment file correctly, I went on further to create an Minimax agent. For Part 1 of the task i decided to change Human Agent in such a manner that the human can't fill the other 2 planes in 3D TTT, rather only one plane was playable which means the game was now a 2D TTT. Now lets see the Final approach.

V. FINAL APPROACH

I do not want to go into detail of how i changed the example file to make the 2d TTT Minimax agent. All i did was to restrict human to play into other two planes. And for inner calculations Created a matrix called world which stored the current board state.

Now lets discuss the Minimax agent for 2D:

```


class AI(object):
    global world
    def __init__(self,mark):
        self.mark = mark

    def act(self, ava_actions,world):
        if is_empty(world):
            row  = randrange(3)
            col  = randrange(3)
            action = '0' + str(col) + str (row)
            world[int(row)][int(col)] = int(self.
    mark)
            return self.mark + action
        else:
            a  = findBestMove(int(self.mark))
            print("A is  : ", a)
            world[int(ALoc[0])][int(ALoc[1])] = int
    (self.mark)
            action = '0' + str(ALoc[1]) + str (ALoc
    [0])
        return self.mark + action


count = []
def minimax(level, player_mark : int, is_max : bool
    ):
    global world
    global count
    opponent_mark = 2/player_mark
    #check if the previous move ended the game
    if is_max : score = game_over(world,player_mark
    )
    else : score = game_over(world,opponent_mark)

    if score ==10 :
        return 100 -(level*10)
    if score ==-10 :
        return -100 + (level*10)

    if is_full(world) and score ==0 :
        return 0   #board full , there is a tie
    if is_max:
        value = -1000
        for i in range(3):
            for j in range(3):
                if world [i,j] ==0:
                    world[i,j] = player_mark
                    value = max(value,minimax(level
    +1,opponent_mark,False))
                    world[i,j] = 0

    else:
        value = 1000
        for i in range(3):
            for j in range(3):
                if world [i,j] ==0:
                    world[i,j] = player_mark
                    value = min(value,minimax(level
    +1,opponent_mark,True))
                    world[i,j] = 0
    return value


def findBestMove(player_mark) :
    global world
    best_move = -10000
    global ALoc

    for i in range(3) :
        for j in range(3) :
            if (world[i][j] == 0) :
                world[i][j] = player_mark
                value = minimax(0, (2/player_mark),
     False)

                world[i][j] = 0

                if (value > best_move) :
                    ALoc = [i, j]
                    best_move = value
    return best_move
```

Listing 1. 2D Minimax Implementation

Now lets go Line by Line:

Initially when the Ai agent is called to act, it checks the world. If the world is empty, then in that case is randomly selects any position and plays that move as it can search till the all the possible end states , this doesn't hurt our chances to win or draw at all.

Now if the world is not empty, Then it calls the function Find Best move. Find Best move essentially calculates the reward values for all the possible empty positions and maximum value of them ans store then in global List Aloc and returns it back to Ai agent which then gives the GYM environment the returned action.

Inner Workings of the Find best move :

When function Find best move is called it checks for empty position on the world and then assumes that the player plays there and then calculate the chances of winning in this position. After that it revert backs the move and find the next empty space and then repeats the same process. After all empty spaces has been filled once it retuns the move which fetched the maximum result.

Now how does the the function calculate the chances of winning?...the answer is that it calls the Minimax Function.

MINIMAX Function :

When Minimax fucntion is called for a certain game state (i.e. world) it first checks if the player which called the find best move wins or lose in this game state. If win then it return a positive value, lose then negative. If neither happens and the world is full i.e. no empty space then it returns 0 which mean there is a tie (Draw).

Now if the game is not finished then it will find all empty spaces in the world and fill then with the player mark one by one and analyze all the game state formed by it by perspective of the opponent and then assume opponent fill space left empty spaces one by one. Then it again analyze all the game state formed by it by perspective of the player, until the game ends and return some value. Now it keeps on doing a series of alternate minimum and maximum on it till it reaches the level which was called by the Find best move and then return the values to the find best move for that position.

Part 2: 3D implementation of the Minimax Algorithm

If the same minimax was implemented on 3D TTT, the number of game states and depths the code has to go will take the prgram ages to complete. And we simply can't wait so much for a single move. So we modified the algorithm and even the Gym environment

Update in gym environment : The Gym env uses a code to check if any player won for this purpose the code used was very inefficient and did a lots and lots of useless calculations. Since these code was going to be run for every game state, this cost us very much so i have changed and modified the environment.

Level limit and evaluation : Then, next it is not possible to see up to depth 27 or 26. Even depth 10 will take lots of minutes, So it is clear we have to limit our depth level search to a small number like 4 or 5. But most game states wont complete till 4 levels. then how will we evaluate the game state. TO solve this problem we create another function which takes our world as input and gives a value depending on how good our position is.

It checks how many lines have one or two of our values and give reward. It gives negative reward for opponent . So the states where our win chances looks high will be given higher rating and will be selected.

Forced Moves : By observation and tests it is clear that the player occupying the body center has higher chances of winning. So we force a move by hard coding it. If the Ai got first move then it will definitely play (1,1,1) move. If the Ai doesnt get first move but whenever the world is input such that the center position is empty, it will play in that move only. This saves a lots of time and also plays the best possible strategy making the AI really really tough to compete against.

Also if it is the second move and the 1st agent played (1,1,1) move then it chooses one of the eight corner randomly.

ALPHA-BETA Pruning: Though i cant explain the how of the concept here it essentially saves redundant tree transversals via checking the previous obtained values. When the search space is as big as ours it is very very useful.

CODE IMPLEMENTATION :

```


class AI(object):
```

```python
    global world
    def __init__(self,mark):
        self.mark = mark

    def act(self, ava_actions,world):
        if len(ava_actions) >25:
            #if it is first or second turn
            #applying the forced move concept
            if(world[1,1,1] ==0):
                #check if the centre block is empty, if so then take that move
                world[1,1,1] =self.mark
                return self.mark + "111"
            else:
                #if centre is filled i.e the opponents first move is centre
                #then move at corner of block 0 and 1 is the optimal move
                block = randrange(2)*2
                row   = randrange(2)*2
                col   = randrange(2)*2
                action = str(block) + str(col) + str (row)
                world[int(block)][int(row)][int(col)] = int(self.mark)
                return self.mark + action
        else:
            a   = findBestMove(int(self.mark))
            world[int(ALoc[0])][int(ALoc[1])][int(ALoc[2])] = int(self.mark)
            action = str(ALoc[0]) + str(ALoc[2]) + str (ALoc[1])
        return self.mark + action


def minimax(level, player_mark : int, is_max : bool, alpha, beta):
    global world
    #print(level)
    opponent_mark = 2/player_mark
    #check if the previous move ended the game
    state = check_game_status(world)
    if state != -1:
        if is_max : ai_mark = player_mark
        else : ai_mark = opponent_mark

        if state == ai_mark :
            #print("Well this is awkward")
            return 1000 -(level*10)
        if state == 2/ai_mark :
            return -1000 + (level*10)

    if is_full(world) and state == -1 :
        return 0   #board full , there is a tie
    if level > 2 :
        if is_max : return estimate(world, player_mark)
        else : return estimate(world,opponent_mark)

    acto = TicTacToeEnv.available_actions(env, world)

    if is_max:
        value = -100000
        for _action in acto :
            _loc = (list((_action)))
            i =(int)(_loc[0])
            j= (int)(_loc[1])
            k = (int)(_loc[2])

            world[i,j,k] = player_mark
            value = max(value,minimax(level+1, opponent_mark,False,alpha,beta))
            alpha = max(alpha,value)
            world[i,j,k] = 0

            if beta < alpha:
                break

    else:
        value = 100000
        for _action in acto :
            _loc = (list((_action)))
            i =(int)(_loc[0])
            j= (int)(_loc[1])
            k = (int)(_loc[2])

            world[i,j,k] = player_mark
            value = min(value,minimax(level+1, opponent_mark,True,alpha,beta))
            beta = min(beta,value)
            world[i,j,k] = 0
            if beta < alpha:
                break
    return value


def findBestMove(player_mark) :
    global ava_actions
    global world
    best_move = -10000
    global ALoc

    for _action in ava_actions :
        _loc = (list((_action)))
        i =(int)(_loc[0])
        j= (int)(_loc[1])
        k = (int)(_loc[2])
        world[i,j,k] = player_mark
        value = minimax(0, (2/player_mark), False, -1000000,1000000)

        world[i,j,k] = 0

        if (value > best_move) :
            ALoc = [i,j,k]
            best_move = value
    return best_move
```

Listing 2. 3D Minimax Implementation

The code is pretty much the implementation of the the above discussed topics

VI. Results and Observation

The result is a AI which cannot lose in case of 2D and plays perfectly in 3D. The 3D AI will always win if it goes first.

One Might notice that i evaluate the state if level ¿ 2 i.e level = 3. This means that the net possible states if the AI goes second is are : 26 in find best move

25 in level 0

24 in level 1

23 in level 2

22 in level 3

That means 26*25*24*23*22 = 7.9 Million game states have to evaluated (Some of them are unreachable though)

In this case for a move the AI on an average takes 35 seconds. This level can be increased but the performance more or less remains the same but each move takes more time,

VII. Future Work

I have also made Q-table 2D implementation as well as a random agent to train Q-table Agent but even after 10 hrs of training and millions of episode the Q-table was only 45 percent draw (Win not possible against Minimax 2D)

CONCLUSION

It was a lot of fun. lot of time went in learning Q-table and DQN but i could not implement it. But It was very interesting to make a human defeating AI. Minimax Algorithm is really very powerful tool,

References

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

[2] Akshay L. Aradhya ”Minimax Algorithm in Game Theory”, Geeks for Geeks, 2021

[3] Carl Felstiner, ”Alpha-Beta Pruning”, 2019