Aerial Robotics Kharagpur Documentation Template

Neha Dalmia

***Abstract*—**
In this project, I have implemented the minimax algorithm in a game of tic-tac-toe. Minimax is a backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Backgammon, Mancala, Chess, etc. Such games are called games of perfect information because it is possible to see all the possible moves of a particular game. We were provided with a skeleton code and had to alter it in order to implement the minimax algorithm. The objective of this task was also to familiarise oneself with working on pre-written code. The pre-written code had heavy components of machine learning which had to be navigated through and the process required a thorough understanding of what the various methods given in the code do and what they return. The process also involved understanding various methods of implementing a given algorithm depending on the source code a person is working on. Several additions had to be made to the basic algorithm in order to make it usable with the given code provided to us. This involved using methods and variables provided in the skeleton code. My approach was centered around first thoroughly understanding the Minimax algorithm and implementing it on my own. This was followed by thorough reading and understanding of the skeleton code provided and looking up unfamiliar terms I encountered in the process. Then I implemented the algorithm in the code provided.

I. INTRODUCTION

In the first task we had to write an agent to play a game of Tic-tac-Toe using minimax algorithm. We had to use the gym-tictactoe environment available on github. I implemented the minimax algorithm using recursion in which the game board was updated in each recursive call. If the game board was such that the game had come to an end, a score was returned, 10 if the agent wins -10 if the agent loses and 0 in case of a draw. The agent then chooses the move which will lead to it maximising its score and the human agent having the minimum score. I used after_action_state methos to return the board and a particular move and ava_actions list to check which index numbers were available to play. This list had to be returned to its original state after the function call to prepare it for the next call.

II. PROBLEM STATEMENT

In the first task task we had to write an agent to play tic tac toe using gym tic tac toe environment. We could have used adavanced techniques like Q value learning as well.Minimax is a decision-making algorithm, typically used in a turn-based, two player games. The goal of the algorithm is to find the **optimal next move**. In the algorithm, one player is called the maximizer, and the other player is a minimizer. If we assign an evaluation score to the game board, one player tries to choose a game state with the maximum score, while the other chooses a state with the minimum score.In other words, the maximizer works to get the highest score, while the minimizer tries get the lowest score by trying to counter moves.It is based on the zero-sum game concept. In a zero-sum game, the total utility score is divided among the players. An increase in one player's score results into the decrease in another player's score. So, the total score is always zero. For one player to win, the other one has to lose. Examples of such games are chess, poker, checkers, tic-tac-toe.Our goal is to find the best move for the player. To do so, we can just choose the node with best evaluation score. To make the process smarter, we can also look ahead and evaluate potential opponent's moves.For each move, we can look ahead as many moves as our computing power allows. The algorithm assumes that the opponent is playing optimally.Technically, we start with the root node and choose the best possible node. We evaluate nodes based on their evaluation scores. In our case, evaluation function can assign scores to only result nodes (leaves). Therefore, we recursively reach leaves with scores and back propagate the scores. We were supposed to use the **gym tic-tac-toe environment**. Int this environment env.py contained the various methods which were needed to check the moves available after a turn was played by human or minimax agent, to check the status of the game and return a value based on who won or if it was a draw. It also included methods to find out the status of the board after each turn was played. The environment was set up to start with the minimax agent. It denoted 'O's to the minimax agent and 'X's to the human agent. We had to return the move of the minimax agent. The move of the human agent was input by the user by entering a value between 1 to 9. Invalid moves were not allowed. Use of these functions minimised my interaction with state in the end which simplified the entire process and made it more concise.

III. RELATED WORK

Apart from minimax algorithm, **Reference Learning** is a highly efficient approach to making a tic tac toe game in which a tic tac toe agent improves each time. For games which are more complex than a simple 3X3 tic tac toe, **Alpha-Beta Pruning** can be used to cancel out reduntant nodes and increase the speed of the algorithm.

IV. INITIAL ATTEMPTS

I used the minimax algorithm to solve the problem by utilising the predefined methods in the skeleton code. Initially I tried to operate without interacting with the state at all but as I was unable to do that, I had to use the state while calling my check_game_status method.

V. Final Approach

In this segment I am going to present my step by step approach of tackling the given problem statement. First issue I had to face was **setting up the environment** as I had not done programming in python before. I downloaded visual studio code in python and downloaded the repo. After downloading all the library packages, I finally managed to compile the code and get it running. The next task was **implementing the minimax algorithm**. Here I used a recursive tree aproach to deal with it. Every recursive call of the method **minimax in the class MinimaxAgent** started with checking whether the game had ended using the **check_game_status** method in which the board was passed. This was the base case. Then we returned 10 if the minimax agent won, -10 if the human agent won in the present board status and 0 if it was a draw. If the board status was such that the game was still continuing, we would then check if it was the human agent's turn or the minimax agents turn. Then we removed the first available action in **ava_actions** and then called the method again, this time changing the value of **turn** such that the next time it will take the turn of the other agent. If the recursive call started from the minimax agent, the moce which returns the highest value ie 10 will be chosen and in case of human agent the moce which returned the lowest value aka -10 will be chosen. After every recursive call, the move whoch was removed from aca_actions was added again at the same position in order to prevent resual of the value which would have given unexpected results in the program. This method would eventually return the value which is the result of maximising the score of the minimax agent and minimising the score of the human player. This value was returned to the method **act**.This is the method which actually returns the move that the minimax agent eventually chooses. It takes the game state and the list of available actions are arguemnts and returns the optimal move for the minimax agent based on the current status of the game and the moves that are available to be played. It runs a loop which accesses the elements of ava_actions and calls the recursive method minimax for each action. The call is made with the current action removed the with the turn of the human player. If the value returned by the minimax method is greater than the maximum value till then, the most optimal move becomes that particular move in ava_actions. After the recursive call the move is added to ava_actions in the same position and the comparison is made to decide the move is better than any previous move. if it is, then the index number of the grid of the board is stored in **pos**. The method act eventually returns the value of pos as the move to be played by the minimax agent.

Errors faced: The most prominent problem I faced in this entire process was understanding and dealing with ava_actions. Here ava_actions is a list which contains the grid numbers for the boxes which are still empty and can be played in. Initially, I forgot to reappend the removed element to the list and as a result the code was not working correctly. When I did reappend it, the problem I started facing problems as initially I did not reappend it at the same position the element was removed from. Owing to this problem, the ava_actions list kept on changing in a way such that some moves which should have been considered after the recursive call were not being considered and hence the minimax agent was not playing the moves correctly and kept on losing. Hence I started printing the list elements to see why it was not considering some very viable moves and then I realised that some elements were shifting to the end of the tuple and were not getting added to their correct positions. Thus I started appending the removed element at the correct position. Other general problems that I faced were mostly centered around the general understanding of the given skeleton code and how to use its methods. I had some difficulty trying to undertsand how to pass the board status as an argument every recursive call till I understood after_action_state returned the board status after a given move and even shifted the move to the other player thereby reducing my effort in this field as initially I was updating the player whose turn it was to play myself resulting only in the human player getting every move. Understanding how to pass the board to check_game_status was another hurdle and in the end I resorted to using state[0] to pass the state of the board in my base case to check if the game had come to an end in the current board status. Another thing I had to do was read env.py to understand what is returned by check_game_status when the minimax agent wins and what is returned by check_game_status when the human agent wins and what is returned in case of a draw. Initially I assigned incorrect values to the above which consequently led to my minimax agent losing every round due to -10 being returned in case of it winning.

Miscellaneous problems faced: This was my first time working with a python-based environment hence it required a lot of understanding of classes,methods, lists and tuples in python prior to beginning execution. The init function was confusing till I understood it was the python equivalent of a constructor and how a python program can return multiple valued unlike other programming languages. The new style of indentation took a little effort to get used to as my code would throw syntax errors due to improper indentation or due to unexpected reasons like using tab instead of spacebar and the like. However, after understanding how to program in python relatively better, the code became much easier to understand and speeded up the implementation.

VI. Results and Observation

The other algorithms I could have used in place of or in addition to the current algorithm are **Alpha-Beta Pruning and Q Value Learning**. The former involves using depth which has been defined in my method definition in order to cancel out unnecessary calls to the method and in the latter the agent learns in to become a better player every turn. In Q value learning, at each step, the player (also interchangeably called agent) takes a good look at the current environment and makes an educated guess of what it thinks is the best action in the given situation. It executes the selected action and observes how that action has changed the environment. It also receives feedback on how good or bad its action was in that particular situation: If it chose a good action it will get a positive reward, if it chose a bad action it will get a negative reward. Based on that feedback, it can then adjust its educated guessing process. When it encounters the same situation again in the future, it should hopefully be more more likely to chose the same action again if the feedback was positive and more likely to chose a different action if it was negative.

The reason I chose a simple minimax algorithm in place of the much speedier methods mentioned above is because this is a game with a very small sample space and hence the time lag is only during the first move when the number of comparisons is very large and it has to go through a large number of game spaces. However, with each move the number of comparsions decreases hence these techniques are not needed in a simple 3x3 tic tac toe game. In games like chess and even 5x5 tic-tac-toe the number of comparisons would have been much larger and speeding up techniques would have been necessary. Hence I decided to stick with the simple minimax algorithm for this.

VII. Future Work

The major problem with the algorithm that has been implemented is that it will work efficiently in extremely simple games like a 3x3 tic-tac-toe game. If used in games like chess for instance the run time for the algorithm will be extremely large especially initially thereby making it highly inefficient as a game . Thus, in order to be used in these cases, additions like alpha-beta pruning can be made to the minimax algorithm. This addition has been made easy by already including the depth parameter in the method thereby avoiding redefining the function definition. Another restriction arising due to the use of minimax algorithm is that it can be used only in turn based cames which reduces it's scope of application. Also it's speed decreases if more than 2 players are involved in the game. Multiplayer games can be dealt better by using **Bound Pruning** or **Speculative pruning**.

Conclusion

The problem statement involved us working on a predefined environment and implement the minimax algorithm to return the most optimal move of the machine in a 3x3 tic-tac-toe game. Quick decision making machine algorithms are of great use as machine decision making has to be as quick as possible. Although this problem was in context of a game, understanding the sample space and making the most appropriate choice based on the information which has been provided to us is of utmost use in spheres like **drone data analytics** which involves studying landscapes and construction sites and the like.

References

[1] An optimal minimax algorithm Gilbert, E.N. Ann Oper Res(1985)