Aerial Robotics Kharagpur Task-1 Documentation

Rishabh Agarwal

Abstract— **In this task, I was supposed to implement the minimax algorithm in a given tictactoe environment. Minimax is a kind of backtracking algorithm, which uses recursion to search through the game-tree, and 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 Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.**

The main objective of the task was to get accustomed to working on pre-written code and work with the given environment. The skeleton code was given and I had to add the recursive minimax function. Since the given code was in Python, I had to learn the basics of it. the implementation procedure involved thorough understanding of the minimax algorithm and then modifying it to incorporate it in the given environment. For that I also had to go through the skeleton code to get a hang of the functions and objects of the given environment.

I. INTRODUCTION

In this task we had to write an agent to play a game of tic-tac-toe using the minimax algorithm. We first had to clone the github repository containing the gym tic-tac-toe environment. In minimax there are two agents-minimizer and maximizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible. Every board state has a value associated with it. When the game comes to an end, if the maximizer wins it returns the value +10, if the minimizer wins it returns -10, and if it is a draw 0 is returned. The agent thus chooses the move that will lead to it having the maximum score at the end of the game.

II. PROBLEM STATEMENT

In this task we had to create an agent that finds the most optimal move. For this we have used the minimax algorithm, a recursive backtracking algorithm that looks into all possible outcomes, typically used in turn-based, two player games. There are two main assumptions of the algorithm-the first that there is no element of chance in the game and the second being that human player must be playing optimally for the algorithm to run most efficiently. In the game there are two players-maximizer and 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 players score results into the decrease in another players 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. To find the most optimal move, the algorithm uses recursion to search through the game tree. The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree. The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion. In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. If it is the maximizer's turn the algorithm will the maximum of all utility values, and if it is the minimizer's turn it finds the minimum among all. This process is carried on recursively.

The cloned repository had the file 'env.py' which was the main file containing the code for the environment in python. It had all the functions described in it-to show the board, to update the board, to check the status of the game. The file 'minimax_agent.py' already had the code for the functioning of the human agent and we just had to add the recursive minimax algorithm. The game was set up with the minimax agent having the first chance and being denoted by 'O's and the human agent by 'X's. The function 'act' invokes the minimax function and returns the best possible move to the 'play' method. The move to be made is represented by an integer from 0 to 9, each representing a position on the board.

III. RELATED WORK

Apart from minimax algorithm, **Q-Value Learning** is a very efficient way of making a game where the agent improves with each iteration. **Alpha-Beta Pruning** can also be used as it increases efficiency by cancelling out the reduntant nodes.

IV. INITIAL ATTEMPTS

Since I was new to programming in python, it took me some time to figure out what exactly the code meant by looking up the different keywords and functions. Also the given environment had lots of extra elements associated with Q-Value learning, which was of no use to me. This was a cause of confusion and once I was able to figure out the part of code I needed to stick to, the process became a bit more clear.

V. FINAL APPROACH

The first issue I faced was **setting up the environment** as I was using python for the first time. I downloaded python and pip and cloned the repository form github. After compiling

the code I was able to run the 'human_agent.py' given in the examples. Going through the code in 'env.py' and the other agents given in the examples gave me a fair bit of idea as to how the different methods were functioning.

The next task was **implementing the minimax algorithm.** I added the function **minimax1** which was called by the **act** function defined under the MinimaxAgent class. The function takes two arguments-the **state** (tuple containing board and marker) and **ava_actions** (array containing possible actions). It then calculates the utility value associated with each move in ava_actions by calling the minimax1 method and sending as parameters the state of the board after that action takes place, the available actions after removing that particular action and the initial depth of 0. The **after_action_state()** method is used to find the state after a particular action takes place. The removed action is again added to the list of available actions for the next iteration, at its corresponding position. The act method returns the best possible move by checking which move has the highest utility value returned by the minimax1 function.

The minimax method takes as arguments- the state, ava_actions, and depth. Depth is basically used to differentiate between two moves that have the same utility value. If such a case arises, the move having lower depth is preffered as it causes the agent to get the win in a shorter time. At first we check what the status of the game is, by calling the **check_game_status()** method. The 0^{th} index of the state tuple is passed, which contains the board. The values returned by this function are 0, 1 or 2 for tie, 'O' winning or 'X' winning respectively. If it is a tie, 0 is returned. If 'O' wins, **10-depth** is returned. If 'X' wins, **-10+depth** is returned. Now we check if it is the maximizer's (minimax agent's) turn by checking the value stored in the 1^{st} index, which is the value of the marker. If true then we call the minmax1 function recursively and return the maximum value obtained. Otherwise, if it is the minimizer's turn, we call the minmax1 function recursively and return the minimum value obtained.

Errors encountered and their solution: In the very beginning I was not using the after_action_state method as a result of which I was changing the values of the board and marker by myself. But this was avoided by using the given method. In the act method while modifying the ava_actions array the action is first removed from the array and then added back to it. Initially I was just appending it to the end of the array, but that lead to wrong output. So I started printing my ava_actions array after every iteration and realized that some of the actions were being left out. Hence I decided to sort the ava_actions array every time I added an element to it. But it would have been computationally more expensive, so ultimately I added a variable to store the position from which the action is removed and added the action back at that position. Another error I faced was the incorrect calling of the method without the use of self.

Miscellaneous problems faced: The most common error I faced was related to the calling of functions in python. This was my first time working on a python platform so I had to read up on how to access elements of tuples and lists, classes, and how to define and call methods. I understood that the init function was equivalent to constructors and 'self' keyword to 'this' keyword. Also the slight variation to the for loop and improper indentation throwed up syntax errors. However as I got the hang of it, implementing the algorithm became easier.

VI. RESULTS AND OBSERVATIONS

The other algorithms that could have been used are- **Alpha-Beta Pruning** or **Q-Value Learning**. The former is not a new algorithm but just an optimization to the minimax algorithm which significantly reduces the computation time by cutting off branches in the game tree which need not be searched because there already exists a better move available. This allows us to search much faster and even go into deeper levels in the game tree. The complexity of the standard minmax algorith is $\mathbf{O}(b^d)$ whereas the alpha-beta pruning has a best case complexity of $\mathbf{O}(b^{d/2})$ and a complexity of $\mathbf{O}(b^{3d/4})$ if the tree is randomized (where b is the branching factor and d is the depth). The latter achieves state-of-art skill by simply trial and learn. At each step, the computer 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 for choosing the simpler minimax algorith instead of the faster methods mentioned above is that the game has a very small sample space. There is a time lag only during the first move that the minimax agent makes, thereafter the number of choices keeps decreasing so the depth of the game tree also decreases. So these optimizations are not needed in a 3x3 Tic-Tac-Toe game. But in games like chess, or even 5x5 Tic-tac-toe the sample space is huge so these optimizations become a neccesity.

VII. FUTURE WORK

The minimax algorithm can be implemented to any 2 player board game with some minor changes to the board structure and how we iterate through the moves. But sometimes it is impossible for minimax to compute every possible game state for complex games like Chess. However, minimax is still useful when employed with **heuristics** which approximate possible outcomes from a certain point. For a game like chess, for instance, pieces are often assigned values to approximate their relative strengths and usefulness for winning the game. We can also use the **Alpha-Beta Pruning** optimization or **Q-Value Learning**.

Games with random elements (such as backgammon) also make life difficult for minimax algorithms. In these cases, an **Expectimax** approach is taken. Using the probabilities of certain moves being available to each player (Eg. the odds of dice combinations) in addition to their conflicting goals, a computer can compute an ”expected value” for each node.

CONCLUSION

Minimax is only useful when the computer is playing an opponent as knowledgeable as itself. However, this will not be the case, and a computer running the minimax algorithm may sacrifice major winnings because it assumes its opponent will ”see” a move or series of moves which could defeat it, even if those moves are actually quite counterintuitive.

In reality an exhaustive use of the minimax algorithm tends to be impractical. A computer can compute all possible outcomes for a relatively simple game like tic-tac-toe but it won’t help. In tic-tac-toe there is no opening move which guarantees victory; so, a computer running a minimax algorithm without any sort of enhancements will discover that, if both it and its opponent play optimally, the game will end in a draw no matter where it starts, and thus have no clue as to which opening play is the ”best.” Even in more complicated games like chess, even if a computer could play out every possible game situation (an impossible task), this information alone would still lead it to the conclusion that the best it can ever do is draw (which would in fact be true, if both players had absolutely perfect knowledge of all possible results of each move). It is only for games where an intelligent player is guaranteed victory by going first or second that minimax will prove sufficient.

REFERENCES

[1] https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/minimax.html

[2] https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/

[3] https://math.stackexchange.com/questions/1471238/complexity-analysis-of-alpha-beta-pruning-of-a-full-tree

[4] https://medium.com/@carsten.friedrich/part-3-tabular-q-learning-a-tic-tac-toe-player-that-gets-better-and-better-fa4da4b0892a