One Player 3D Tic-Tac-Toe with 64 slots

By: Gavin Sidhu, Zahra Saghaie Dehkordi, Nidhi Gowri Srinath, Naimisha Churi

Project Description:

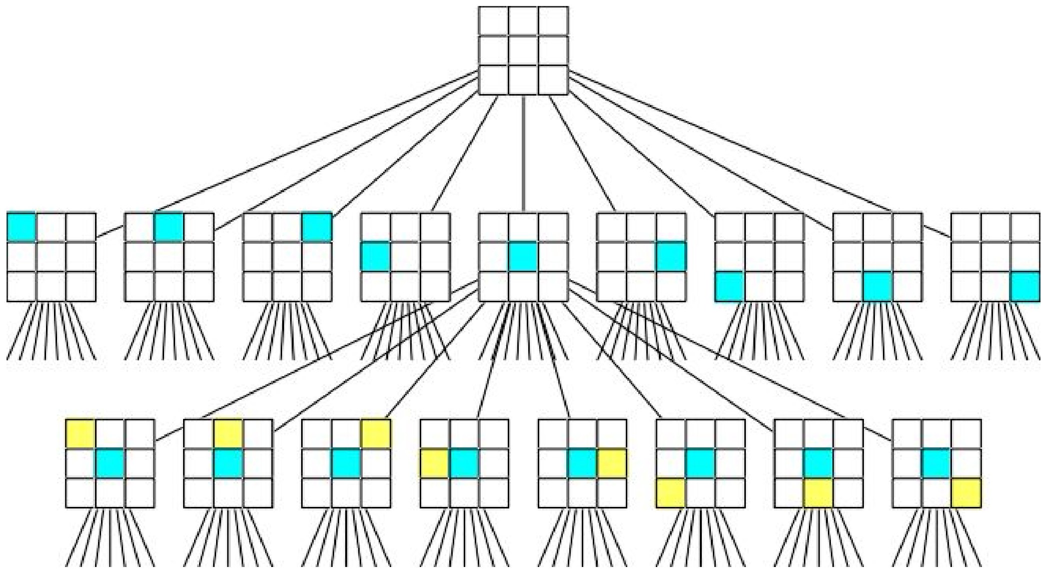
The aim of this project is to implement a one-player 3D Tic-Tac-Toe containing 64 slots with three difficulty levels: easy, medium, and hard. One of the most common applications of AI is to improve the way games are designed and played. The alpha-beta pruning technique is used in our project to extend the game's difficulty levels. The benefit of developing a system that uses the alpha-beta pruning technique to solve a tic-tac-toe game is that it considers all possibilities before taking the most optimal step. The Minimax algorithm is a backtracking algorithm that employs a searching technique to determine the best move. It is frequently used to optimize interactive turn-based games like chess, backgammon, tic-tac-toe, and so on. Every state in the game will be assigned a value. The optimization is performed alternately by the minimizer and the maximizer based on the values, i.e., the maximizer for positive values and the minimizer for negative values.

Alpha-beta pruning is an optimization technique used for the minimax algorithm. The popularity of alpha-beta pruning has to do with its computational capabilities. The algorithm helps explore deeper levels of the tree while also reducing the search time significantly. Apart from the values used for the minimax algorithm, alpha-beta pruning uses an alpha and a beta value, as suggested by the name. Alpha denotes the best value that can be obtained by the maximizer, and beta represents the best value that can be obtained by the minimizer. Fundamentally, the alpha-beta pruning technique reduces the number of evaluated nodes.

In the case of our project, we have used the alpha-beta pruning technique to create a 3D tic-tac-toe with 64 slots. Conventionally, a 3D tic-tac-toe board contains 27 slots, which provided us with a challenge to extend this functionality and user interface to reflect a 4x4x4 board. This variant of the tic-tac-toe game contains 76 winning lines. On each of the 4x4x4 boards, a player can win using one of the four rows, columns, and two diagonals. This adds up to a total of 40 lines. The remaining 36 lines can be made using the board's depths. The basic rules for the project were to include multiple levels of alpha-beta pruning to increase the difficulty of the game. The easy level required the alpha-beta to be 2 levels deep, medium required a depth of 4 levels, and hard required a depth of 6 levels. The algorithm works under the assumption that the player will play the best possible move.

Our solution uses the Python programming language for the code and Tkinter for the graphical user interface. The whole project contains close to 450 lines of code and a pseudo-code for our logic, and a representation of a 3x3 tic-tac-toe game tree has been included below. The pruning performance on a 4x4x4 board is quite like the one shown with the 3x3. This game tree shows us how the optimization works for the game while pruning nodes that are not useful in getting the desired output. If nodes are pruned in the levels just below the root node, the game is completed with the required outcome early. Pruning is much more impactful when used with larger trees. Because tic-tac-toe is a more logically simple game, the essence of pruning and its benefits are not fully realized. The game takes longer to implement optimized moves as we specify more levels of the tree. The algorithm visits multiple nodes before deciding on a move that is nearly impossible to defeat. If the number of levels in the alpha-beta pruning tree is increased, the program may become computationally exhausted. Regardless of these implications, the game ends with the AI winning the majority of the games and drawing the rest.

```
alphaBeta(board, player, alpha, beta):
    if board is a leaf node:
        return value of the board
    if player = 1:
        best = -∞
        for each child node:
            value = alphaBeta(child, -1, alpha, beta)
            best = max(best, value)
            if alpha >= beta:
                break
        return best
    else:
        best = ∞
        for each child node:
            value = alphabeta(child, -1)
            best = minimax(best, value)
            if alpha >= beta:
                break
        return best
```



Tic-Tac-Toe Game Tree

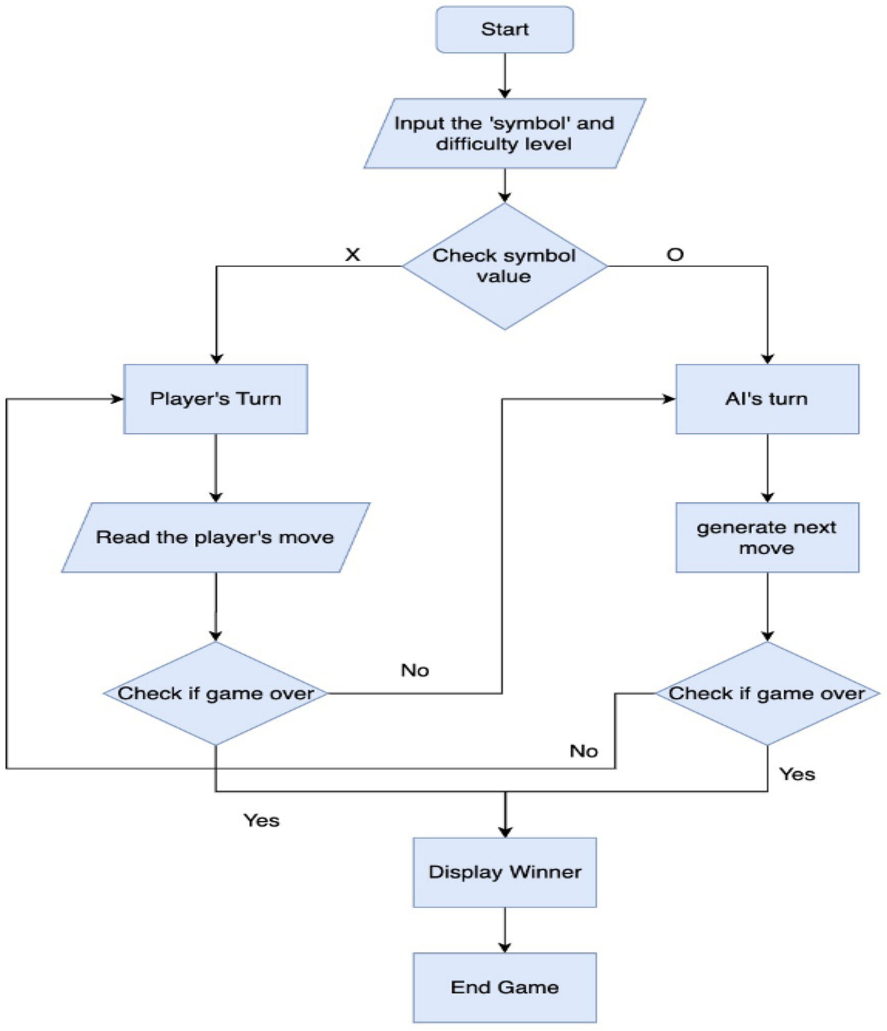
Image Source: Google Images

The traversal in the preceding pseudo-code occurs in post order. Essentially, if the player is 1, and otherwise, the code has similar functionality, except that when the player is 1, we assign the best value to be -∞ and use a maximize function, and otherwise, we assign the best value to be ∞ and use a minimize function. The game tree shows the myriad of possibilities in choosing the appropriate move for a 3x3 game. Each move yields another set of possibilities. At each level, the optimal node is chosen.

Diagrams:

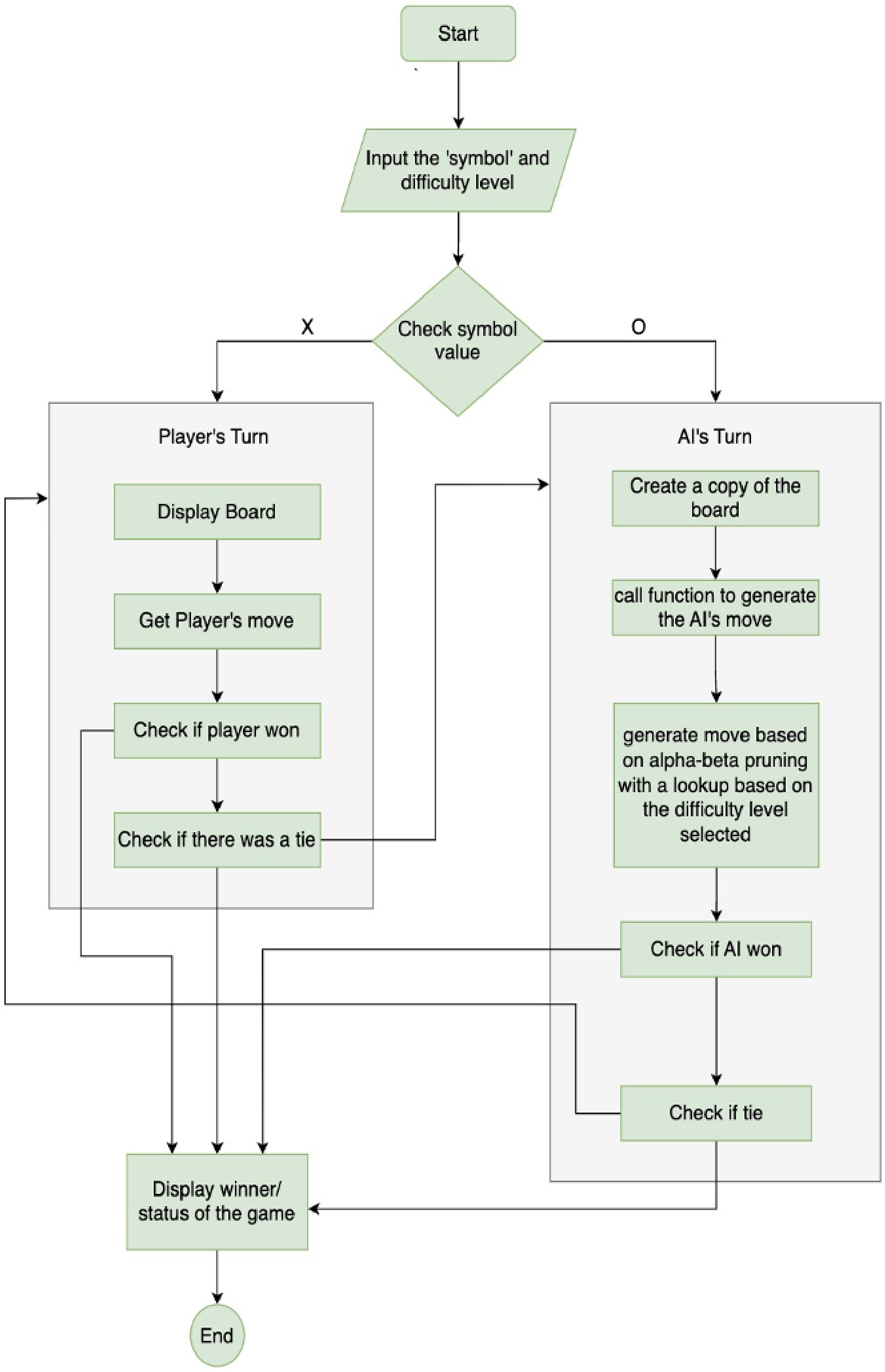
System flow diagrams show a very high level of operations of the systems and how each decision and action taken affects the system flow. A data flow diagram shows the flow of data between the different entities and data stores in a system whereas a flow chart shows the steps involved to accomplish a task. In a sense, the data flow diagram provides a very high-level view of the system. We have tried to capture the flow of the algorithm and what flow the game follows in the diagrams to give a clear idea of it. The system flow diagram shows the flow of the game on a very high level while the data flow diagram shows the components of the game and how each component is interacting with the other

System Flow Diagram:



System Flow Diagram

Data Flow Diagram:


Start
Input the 'symbol' and difficulty level
Check symbol value
X
O
Player's Turn
Display Board
Get Player's move
Check if player won
Check if there was a tie
AI's Turn
Create a copy of the board
call function to generate the AI's move
generate move based on alpha-beta pruning with a lookup based on the difficulty level selected
Check if AI won
Check if tie
Display winner/ status of the game
End

-

Data Flow Diagram

Problem Steps:

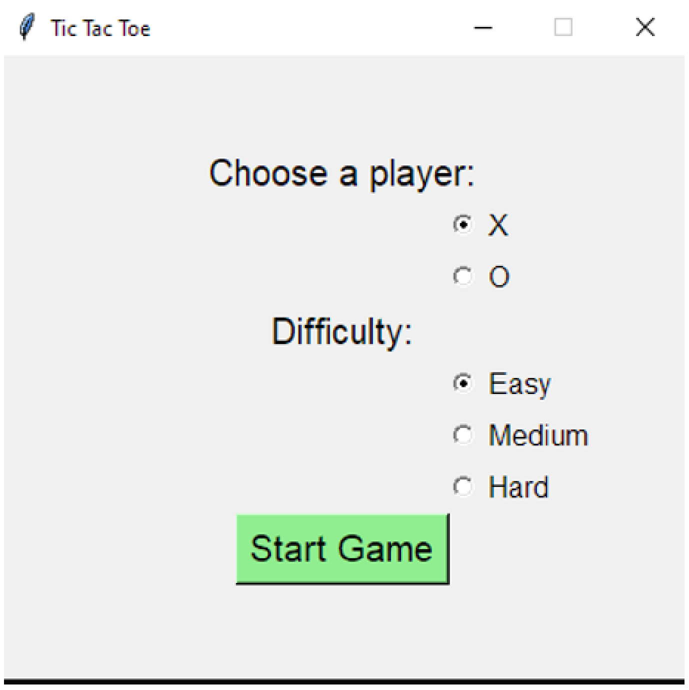
The code can be split into two parts: the logic and the user interface. The logic has been coded in the Python programming language, and the interface uses Tkinter, which is a GUI toolkit.
The UI includes a simple page that allows the player to choose from a few options. The player gets to choose either the X or the O symbol. The player can also select one of three difficulty levels: easy, medium, or hard. The levels of the alpha-beta pruning tree are determined by the player's choice. Following these selections, the UI displays the 4x4x4 grid. If the player chooses to play as X, they will be the first to begin the game. If the player selects O, the AI begins playing the game. The UI displays the winner after the winning moves are completed.

The logic is composed of a variety of functions that all work together to implement the tic-tac-toe game. The following are descriptions of each of the functions.

- def choose_player(): This function determines who starts the game. The symbol is passed as a parameter to the function. The symbol X represents the first move. It informs the program whether the game will be started by the player or by the AI.
- def results(): This function is called to create a copy of the board and use the player's move to pass to the alpha-beta pruning to determine the AI's move.
- def actions(): This function oversees allowing the player or the AI to place their symbol in one of the board's 64 positions.
- def AI_move(): This is the function that is used by the AI to find the optimal location to make their next move. It takes into consideration the depth of the tree and the minimax algorithm itself.
- def alphaBeta(): This function implements the alpha-beta pruning as mentioned in the pseudo-code above. The function minimizes and maximizes based on the player's moves and score. It also implements pruning based on the value of the depth passed to the function as a parameter. The value of the depth also depends on the difficulty level chosen by the player.
- def player_move(): This function determines which player is currently expected to make a move. It records the moves made by the player to determine the next best move for the AI.
- def change_difficulty(): This function determines the depth of the tree to be used for its respective difficulty levels.
- def start_Game(): This function is used to start the game and initialize the boards.
- def utility(): This function is used to check the board to confirm if the current moves could be a part of the defined winning combinations. The AI refers to these combinations to make its optimal move.
- def check_gameover(): This function is used to check if the game is over.
- def updateStatus(): This function is used to update the status of the game. It tells the player if they win or lose the game.
- def value(): This function checks the positions of the moves that have already been made. It then couples up with the remaining functions to proceed to find the best moves. This function is used to find the position where the AI must make its next move.

Input Design:

The input for the app is quite simple. It requires the user to select the symbol and the difficulty level they want to choose. The images below represent the input UI design that is used for these applications.


Tic Tac Toe

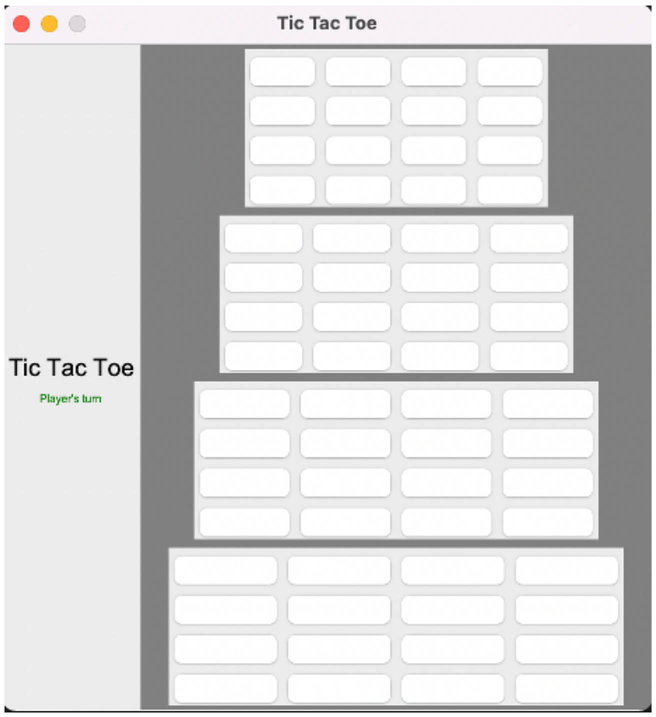
Choose a player:

[✓ X] O

Difficulty:

[✓ Easy] Medium Hard

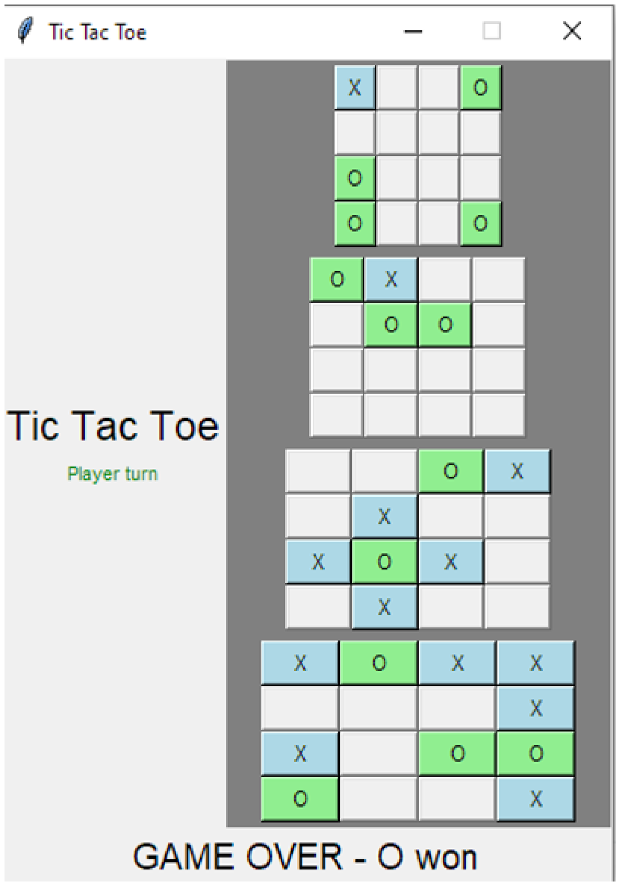
The above screenshot represents the first input that is required by the player. The player and difficulty levels are chosen using radio buttons. The player can select their preferences and then click on the "start game" button. Clicking on the button opens the grid for the game, where the player's or the AI's first move is played.



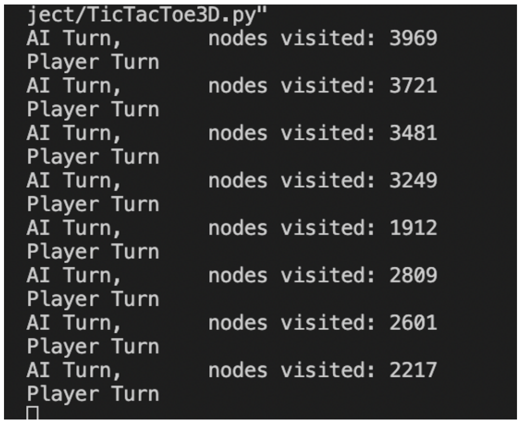
Once the board is opened, the player can begin to make moves and select the grid location of their choice.

<u>Output Design:</u>

The output essentially contains the results of the game. Once the player wins or loses, the results are displayed, and the program shows the winning combinations. Another output that is returned in the terminal is the number of nodes visited by the algorithm for each of the moves that the AI makes. We can see a strong contrast in the values for easy, medium, and hard difficulty levels. The number of nodes visited for the medium and hard levels is extremely high. This results in a delay for each move played by the AI.

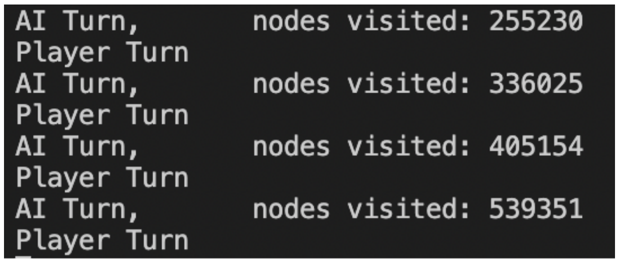


The interface shows the winning moves and the grids at the end of the game. It also updates the winner of the game. The terminals showing the nodes visited are as follows:



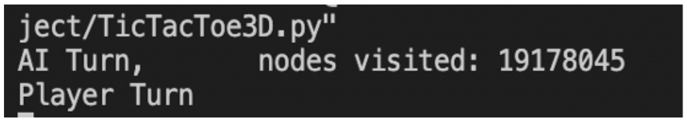
```
ject/TicTacToe3D.py"
AI Turn,        nodes visited: 3969
Player Turn
AI Turn,        nodes visited: 3721
Player Turn
AI Turn,        nodes visited: 3481
Player Turn
AI Turn,        nodes visited: 3249
Player Turn
AI Turn,        nodes visited: 1912
Player Turn
AI Turn,        nodes visited: 2809
Player Turn
AI Turn,        nodes visited: 2601
Player Turn
AI Turn,        nodes visited: 2217
Player Turn
```

The above image shows the nodes that are visited by the AI before it makes its move in easy mode. It is evident that the number of nodes is in the thousands, which means that the response is almost instantly visible. The computation of alpha-beta is happening at 2 levels deep, which means that it is traversing faster.



```
AI Turn,      nodes visited: 255230
Player Turn
AI Turn,      nodes visited: 336025
Player Turn
AI Turn,      nodes visited: 405154
Player Turn
AI Turn,      nodes visited: 539351
Player Turn
```

The above image shows the nodes that are visited by the AI before it makes its move to medium mode. The number of nodes visited is in the hundreds of thousands. This is an indication of how the nodes are visited extremely slowly. The computation of alpha-beta is happening at 4 levels deep, which means that the traversal is slow. Each move takes up to 3–4 minutes to execute.



```
ject/TicTacToe3D.py"
AI Turn,      nodes visited: 19178045
Player Turn
```

The above image shows the nodes that are visited by the AI before it makes its move in hard mode. The number of nodes visited before making a single move is above 19 million. This poses a challenging computation for the problem, requiring more than 10 minutes for each move.

Conclusion:

This project is intended as a representation of the alpha-beta pruning technique of the Minimax algorithm applied to solving the 3D 64-slot tic-tac-toe game. During the course of this project, the team has been able to apply and understand how pruning can work to decide the optimal moves for an interactive turn-based game like tic-tac-toe. Further, with the knowledge of applying the algorithm to this use case, we can try and apply it to other turn-based games like chess, backgammon, and so on. An improvement for the current project would be to optimize how alpha-beta pruning performs in hard mode at six levels of depth. Mathematically, we would be considering a board of 16! possibilities, which is equivalent to about 2.092288 e13, which is a pretty large number.