3D Tic-Tac-Toe Game

Justin Smith, Sampada Sakpal, Zhi Wang, & Bianca Jhonson

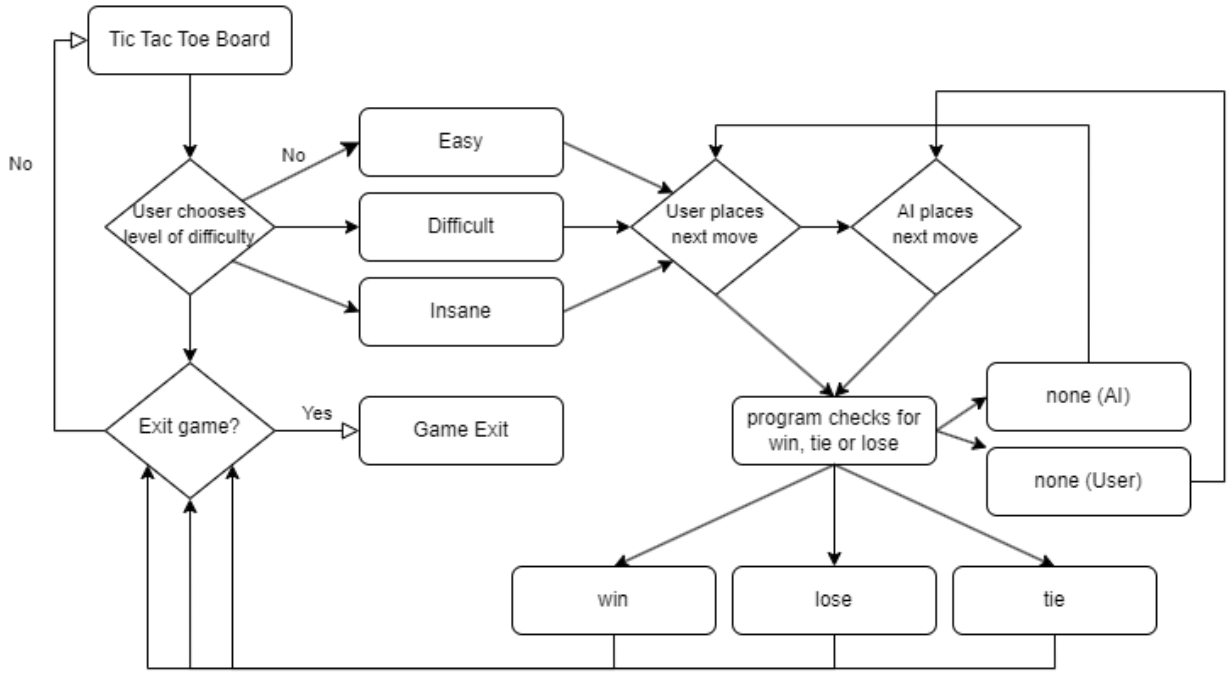
INSTRUCTOR: Dr. Zbigniew W. Ras

Tic Tac Toe Implementation

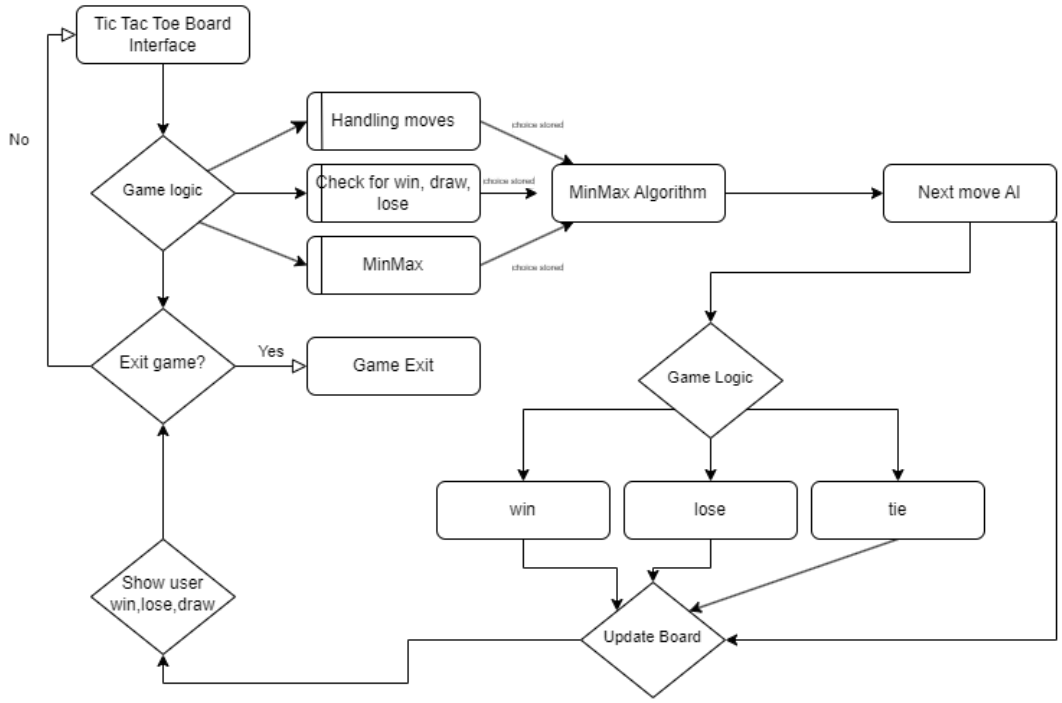
This project entails the development of a 3D Tic Tac Toe game, serving as the final project for the ITCS-8150 course. This interactive application is designed to facilitate gameplay interaction between a user and an artificial intelligence (AI). The AI component, a pivotal feature of this project, leverages advanced algorithmic techniques, specifically employing the alpha-beta pruning optimization layered upon the fundamental minimax algorithm, enhancing the computational efficiency and strategic decision-making capabilities of the AI opponent in the context of the game.

Dependencies needed to run the game; make sure pip install pygame is installed in your global package management or in your virtual env. Then run the file run_game.py in the terminal; python3 run_game.py

System Flow Diagram:



Data Flow Diagram:



Phase 1:

During phase one, we discussed the project, goals, and task designation. Justin, Sampada and Bianca completed their versions of the tic tac toe game implementation and will meet to agree on merging the three versions to start the final draft of the game. Zhi started GUI so the process of finishing the final implementation would be smoother for everyone.

Topics discussed in weekly meeting:

- Programming language to code game
- Game stored in 3D Array
- Turns are Boolean and single-threaded
- Numbers in the array to store where the Player and AI tokens are.
- GUI reads the data as we store it.
- Max depth is based on the difficulty.
- Player first, then AI.
- Program the GUI.

Topics needed to discuss after final tic tac toe game implementation is initialized:

- How do we handle user interactions?
- How often do we update the screen?
- Is the GUI asynchronous with the underlying game?
- If so, do we need to ensure that the screen updates with the user turn before the AI does theirs?
- Are we including menu flairs (hovering over items changes size/shape/color) or will it be static?
- What is the aesthetic theme? Should we worry about high contrast for colorblind users?

Phase 2:

Meeting to discuss each version of tic tac toe game made by Justin, Sampada, and Bianca. We each explained further details of our projects. After choosing the best way to include all the details of the different versions into one, we discussed how the GUI will be integrated into the backend of the game. Zhi completed the GUI and was able to add certain functionalities that we would need for the game to function correctly. The algorithm was settled and the heuristic function was agreed upon.

Topics accomplished and discussed:

- Program the AI.
- Minimax algorithm.
- Alpha-Beta algorithm.
- Difficulty levels.
- Heuristic function.

Topics still to discuss:

- Goal checking implementation?
- Penetration Testing

Phase 3:

Continuing the implementation, the algorithm used to determine wins and the heuristic evaluations is now finalized. To explain further, first, find all points that are at 0s. In the win algorithm, we care if these have actually been played, for the heuristic we just want all of these because we care about partially complete lines. Then, find all directions that you could extend a point in. Determine if for a given point, it can be extended in that direction without falling off the side of the board. For all directions that are valid, extend the point in that direction

For the win algorithm:

- If you find that the line has a 0 or is blocked, stop checking in that direction. If you find that the line is fully complete, return True and the player that won. If you find that there does not exist another 0, return True and a 0 to indicate that no one won. If you don't find any wins or ties, return False and indicate that no one won.

For the heuristic algorithm:

- If you find a 0 while extending, you're fine just don't compute anything. If you find something different than the first value you saw (two people played in this line), set score to 0 (line is blocked and provides no value). If you find something equal to the first value you saw, add to the score of the line. If you make it from one side to the other without being blocked, add this value to the score. Return the score when you're done.

Code Documentation:

GUI.py

Function draw_layers(): to draw all four layer, it Iterates over each layer to draw it. Calculates the top-left origin of each layer, shifted for 3D effect. Draws horizontal grid lines for current layer and draws vertical grid lines for the current layer

Function to draw circles and 'X's based on the board state, it Iterates over each layer, row, and column to draw marks then calculates the origin for the marks on the current layer. Next it calculates the center position for the mark. Then if the slot contains the human player's mark, draw a red circle, if the slot contains the AI's mark, draw a blue 'X'

Function to handle mouse clicks when the human player click or scroll mouse3, it will iterate over all slots to check for the clicked position, define the rectangle for the current slot, or if the click is within the slot and it's empty, mark it with a circle

Function for the AI to make a move, it creates a list of all empty slots l-layer r- row c- column. If there are empty slots, choose one at random and mark it with an 'X'

The ‘main’ game loop will handle the if statements to check if the game is running and it will continue the game. If the player clicks and a circle is placed the AI makes the move after the player, and then redraws the screen with the new marks.

game.py

The class ‘State’ stores the internal state of the game being played.
__init__(self):

- Input:
    - self (the object being instantiated)
- Output:
    - None (Side effect of instantiation)

This initializes the State object with an array of values, initially 0, 0 meaning an empty board.

h(self) accepts self parameters:

- Input:
    - self (the object)
- Output:
    - integer (heuristic value of the state)

The heuristic function. This function will determine a value that represents the "goodness" of a particular state to a particular player. Negative values mean the minimizing player is in a better position, positive values mean the maximizing player is in a better position. Then it creates extension directions. Produces all edge points. The heuristic is negative since the AI is maximizing. Then only contains valid starting positions, it extends each point in each direction. Point has a value in it. We want to encourage points even if they are not directly in a line. No case for a value is 0 is necessary.

The function play(self, x, y, z, player) has:

- Input:
    - self (the object)
    - x (the x value in the array, 0 to 3)
    - y (the y value in the array, 0 to 3)
    - z (the z value in the array, 0 to 3)
    - player (a string, 'MAX' for maximizing player turn, 'MIN' for minimizing player)
- Output:
    - Boolean (Is move legal)(side effect of changing the game state)

Does the turn of the player mentioned. Places a number to act as a token for the player in the game's representation array. The program also stores the last played move.

The isValid(self, x, y, z) function has:

- Input:
    - self (the object)
    - x (the x value of the move)
    - y (the y value of the move)
    - z (the z value of the move)
- Output:
    - Boolean (If move is valid)

Then returns if a given move would cause an error without playing the move.

The getState(self) has:

- Input:
    - self (the object)
- Output:
    - numpy ndarray, 4x4x4 (the game's internal representation)

Then returns the array that represents the current game's state.

Function, setState(self, array) has:

- Input:
    - self (the object)
    - array (numpy ndarray, 4x4x4)
- Output:
    - None (side effect of changing the game state to array)

Sets the value of the game's representation array to the array provided.

copy(self)

- Input:
    - self (the object)
- Output:
    - State (a state in the same state as self)

Provides a copy of the State object.

_potWins(self):

- Input:
    - self (the object)
- Output:
    - List (list of all points that could include wins)

Returns all points with a 0 in some direction (all winning lines have a 0 in some direction, because they cross the board.

def _potWins(self): Main idea, to get a win, you need a 0 in some dimension. Loop over pairs of dimensions, 3(4^2) < 4^3 Store all potential wins, see if they work by expanding in each direction.

Function isWin(self) take object parameters:

- Input:
    - self (the object)
- Output:
    - Boolean (if state is won)

The winner (1 if max player, -1 if min player, 0 if not won), tests if a state is won, and returns the player that won. Extend each potential win in every plausible direction. Determines plausibility of a direction. Then it checks for tie.

getLastPlayed(self)

- Input:
    - self (the object)
    - player (string describing the player whose move is wanted)
- Output:
    - Numpy array of the last played move.

Retrieves the last played move from the state.

The class 'Game' holds all the information about the current Game being played.

__init__(self, maxDepth)

- Input:
    - self (the object being instantiated)
    - maxDepth (the maximum depth that the AI is allowed to search through)
- Output:
    - None (Side effect of instantiation)

Instantiates the Game object, which includes a State and Model object.

run(self)

- Input:

    - self (the object)
- Output:
    - None (Side effect of printing out who won)

Starts the game, ending when the game is finished and printing out who won.

The next two functions actions(state) refers to the current state of the game being played:

- Input:
    - state (Current game state)
- Output:
    - List (List of actions possible in this state)

Outputs all possible playable moves in a given state.

The second function is result(state, action, player):

- Input:
    - state (the state of the game being played)
    - action (3-long Numpy array of coordinates to play at)
    - player (String of who is playing, 'max' or 'min')
- Output:
    - State

Auxillary function to make a copy of a state and play a move on it. Used to aid the AI.

model.py

__init__(self):

- Input:
    - self (the object being instantiated)
    - maxLayers (maximum depth of the tree)
- Output:
- None (Side effect of instantiation)

Initializes the Model object, with a maximum depth of alpha-beta pruning.

_dist(self, point, playerPoint, aiPoint) has:

- Input:
    - self (the object)
    - point (Numpy array of coordinates on grid)
    - playerPoint (Numpy array of coordinates of last player move)
    - aiPoint (Numpy array of coordinates of last AI move)
- Output:
    - Integer

Produces the sum of the distances between point and both previous move points.

_merge(self, left, right)

- Input:
    - self (the object)
    - left (list of 3-long numpy arrays)
    - right (list of 3-long numpy arrays)
- Output:
    - List of 3-long numpy arrays

Merges the two input lists together in heuristic order, based on _dist.

_mergeSort(self, array)

- Input:
- self (the object)
- array (list of 3-long numpy arrays)
- Output:
- List of 3-long numpy arrays

Heuristically sorts the input array.

maxSearch(self, state, alpha, beta, depth, possibleActions)

- Input:
    - self (the object)
    - state (the current game state)
    - alpha (the highest value seen in this path)
    - beta (the lowest value seen in this path)
    - depth (the current depth of the recursion)possibleActions (list of all possible actions)
- Output:
    - Float (the perceived value of this state)
    - List (the list of actions to take after this state to get the best value)

Implements the search of the maximizing player in alpha-beta pruning.

minSearch(self, state, alpha, beta, depth, possibleActions)

- Input: self (the object)
    - state (the current game state)
    - alpha (the highest value seen in this path)
    - beta (the lowest value seen in this path)
    - depth (the current depth of the recursion)possibleActions (list of all possible actions)
- Output:
    - Float (the perceived value of this state)
    - List (the list of actions to take after this state to get the best value)

Implements the search of the minimizing player in alpha-beta pruning.

alphaBetaSearch(self, state)

- Input:
    - self (the object)\
    - state (the current game state)
- Output:
    - Numpy Array (Best move to make)

Calls the initial alpha-beta search and returns the optimal action. The optimal action is the one that puts the model in the best position, given its heuristic evaluation of the state.

run.py

This file is for the tests of the functions included in the game. The following functions are used in the file for testing.

stateTest() has no parameters:

- Input:
    - None
- Output: None (Exceptions on failure)

Thoroughly tests the functions of State. Heuristic test do not exist yet, so there are not tests yet. With the following functions, the game is tested:

- def debug() tests all aspects of State with try/except
- def main() starts the game with Y/N question
- def printStatus(status) will print whether there are errors or not

run_game.py

First checks current game state then initializes the game. There are multiple constants and button implementations to keep the games format. Including difficulty levels and their corresponding depths, the RGB Colors for the application, colors for buttons, font for button text. This will set up the game display, along with difficulty button functionality. Each button will change color when hovered over. The restart will continue within the game being played The function to check for button clicks to iterate over the buttons and check if the provided position is within the button

The function to draw all four layers will iterate over each layer to draw it, calculate the top-left origin of each layer, shifted for 3D effect, draw horizontal grid lines for the current layer and calculate starting point of the horizontal line based on row number. Calculate ending point of the horizontal line, to draw a green line between the start and end points, draw vertical grid lines for the current layer.

The function to draw circles and X iterates over each layer and draws the designated ‘X’ for the AI and ‘O’ for the user when placed. Iterate over all slots to check for the clicked position to place an ‘X’. This will connect to the function for the AI to make a move, and will perform the function to check if won and if tie.

The Main game loop stores buttons for difficulty selection. Included in this main, this section of game_over check is for handling the end-of-game scenario where it's necessary to determine the PLAYER interaction with the Replay and Quit buttons. This includes if the player clicks and a circle is placed, the screen will be redrawn. Then check if the player's move resulted in a win if it did, it will skip to the end of game handling or the AI makes a move after the player. Then check if the AI's move resulted in a win if it did it will skip to the end of game handling Redraw the screen based on the current state of the Game. After it will draw buttons if waiting for difficulty selection and draw the end game buttons and message. Reset the game and reset to difficulty selection. Quit the game then the gameLoop is ended

Challenges Faced During Implementation:

GUI was a bit tricky to implement at first, once the overall design of the game was created, it was a smooth implementation. Once GUI was mostly accomplished, stitching the back end to the front end was the challenging part. Once most of the back end was complete it was hard to spot any errors without a front end to test with. With the GUI the obstacles were revealed. We noticed the AI was determined the winner after the first move by the player. The bug was winTuple[1] was either 1, 0, or -1, but it was only being checked as if equals 1 or else, meaning that it was always seen as a win, and also didn't accept ties. After the player won, it would crash and it was found that some of the diagonals were set up so that it was required to move negatively in some directions, and had to reformat things in the win function. The game became slow as the difficulty increased, so it was updated so that the program calculates the list of possible actions once and updates that instead of recreating the list on each iteration of maxSearch/minSearch loop. The most recent issue is that while we're far from the theoretical maximum calls, we're also far from the theoretical minimum. Theoretical big O for alpha-beta is approximately 4096 calls according to the book for a depth of 4, but with testing it seems like the first move on ‘Difficult’ difficulty makes approximately 40000 calls to max and min search. To solve this it was decided that cutting the actions down to only the top 20 or so values ordered by heuristic value would be best It made the number of leaf nodes go from O(63^depth) to O(20^depth), a difference of 62459502209 leaf nodes.