Project: Connect 4

My project consists of a 'Connect Four' game which can be played either against another player or against an AI. The game is played in the terminal on a 6x7 board made of '-' and '|' characters, along with one row along the top made of ' ' characters and one player piece. Player 'x' always moves first. The player can choose the column they want to play in by pressing 'a' to indicate a move to one column to the left, or 'd' to indicate a move to one column to the right. Once the player's piece is above the desired column, they can press 's' to drop the piece into the column, and the other player's turn starts. 'q' can be pressed at any time to end the game, resulting in a draw. Otherwise, the game is played until a player wins by connecting 4 or more pieces in a row or by filling the entire board with no possible moves remaining.

Implementation

Below is a list of functions used as abstractions to smaller problems within the larger problem of playing the game of 'Connect Four'.

n = one board dimension for time complexity analysis

main() function:

The **main()** function is called at the beginning of the game and continues until the program is terminated. All other functions may be called from the main() function so it is responsible for collecting all user input and organizing the effective outputs for them such that the 'Connect Four' game runs correctly. This function does not return a value. This function has a time complexity of $O(n^2)$ disregarding other function calls, otherwise it's $O(depth^n)$.

Before the game begins:

The **init_game()** function gives the player the option to play versus a friend or an AI. The player is then given the option to review the rules and controls, after which the game starts. This function has a time complexity of O(1).

The **init_firstplayer(playAI)** function gives the player the option to play either first or second against the AI. In this function, *playAI* is passed in as True if the player has chosen to play against the AI, and False otherwise. The function returns 'o' to move first if the player has chosen to play against AI and has chosen to play second, and 'x' otherwise. This function has a time complexity of O(1).

The **init_topRow(topRow, xPos, player)** creates a top row: [' ', ' ', ' ', *player*, ' ', ' ', ' '], and returns it to the main function. In this function, *topRow* is passed in as [], *xPos* is passed in as 3 and, *player* is passed in as the first player's piece. This function has a time complexity of O(n).

The **init_board(board)** creates an empty 6x7 board from a 2D array where every index is '-' and returns it to the main function. In this function, *board* is passed in as [].This function has a time complexity of $O(n^2)$.

Player versus player functions:

The **updateTopRow(topRow, xPos, newxPos, player)** function is called when a move left, right, or down is made. It replaces the player piece in the top row from where it was to where it needs to be in the case of a left or right movement and replaces the player's piece with the opponent's piece in the case of a move down. It then returns the updated top row. In this function, a *topRow* is passed in as the previous stated of the top row, *xPos* is passed in as the previous position of the piece in the top row, *newxPos* is passed in as the new position of the piece in the top row, and *player* is passed in as the current player's piece. This function has a time complexity of O(1).

The **updateBoard(board, xPos, yPos, player)** function is called when a piece is dropped onto the board. It places the played piece onto the board where it belongs. It takes the current board (for the AI a simulated future board) the x and y coordinates of the piece to be played on the board, and the player whose piece is being placed on the board. It returns the updated board. This function has a time complexity of O(1).

The **win(board, xPos, yPos, player)** function is called when a piece is dropped to see if that move was a winning move. It checks in all directions(except up) to see if there are three more of the player's dropped piece in a row beside it. It takes a board, the x, and y coordinates of the most recently played piece and the player piece of the player who played it. It returns true if the game was won on that move and false otherwise. This function has a time complexity of O(1).

The **getvalidmoves(board)** function is called when a player tries to drop a piece. It checks to see which of the top spaces in each column on the board are empty ('-'). It takes the current board (for the AI a simulated future board) to check the top row of. It returns a list of all rows which may be played in (ex: [1, 4, 5]). This function has a time complexity of O(n).

The **move(char, xPos)** function is called when a player moves left or right. It takes the player input as char and the previous x coordinate of the piece in the top row. It adjusts xPos accordingly and returns it. This function has a time complexity of O(1).

The **drop(board, xPos, player)** function is called when a player drops a piece onto the board. It takes the current board (for the AI a simulated future board), the x coordinate of the piece to be dropped, and the player whose piece is dropped. It uses a binary search to find the lowest y coordinate of the dropped piece possible without covering any existing pieces. It returns the corresponding y coordinate. This function has a time complexity of O(logn).

The **boardisfull(board)** function is called after a piece is dropped onto the board. It takes the current board (for the AI a simulated future board) and checks if any of the columns in the top row of the board are empty ('-'). It returns true if no moves are remaining, and false otherwise. This function has a time complexity of O(n).

The **output(board, topRow)** function is called after any type of move and, all the calculations for that move, have been updated and completed. It simply prints the top row, then the board so the player can easily see them. It prints ' ' between each value in topRow and '|' between each value in *board* so it is easier for the user to visualise a real connect four board. This function does not return a value. This function has a time complexity of $O(n^2)$.

The **endgame(board, topRow, gameWon, player)** function is called after a game is finished. It prints the board one last time then announces the winner of the game. It takes *board* and *topRow* to be printed, and it takes *gameWon* to decide whether or not the game was won or finished in a draw. I then uses this information to print a message like "Player : x wins!!" or "The game resulted in a draw." This function does not return a value and the program terminates immediately after this function is called. This function has a time complexity of O(1) disregarding function calls, otherwise it's $O(n^2)$.

Player versus Computer functions:

The **addscore(searchDirection, player)** function assesses the score of one 4x1 window of pieces in a direction. It increases *score* for states such as 3 in a row for the AI which could lead to a win, and decreases score for states such as 3 in a row for the opponent which could lead to a loss. The values for the scores are arbitrary values which seemed to be working well when testing. This function takes *searchDirection* as a single 4x1 window and player as the current player whose turn it is. This function has a time complexity of O(n).

The **boardscore(board, player, xPos)** function gives a score to a board state which describes the usefulness of all moves leading up to that board state. This function cumulates all scores revieved from the addscore() function for each board state. This function takes in a simulated future board state and the piece of the player who played the most recent move. The function returns an integer score for this board state. This function has a time complexity of $O(n^2)$ disregarding function calls, otherwise it's $O(n^3)$.

The **bestmove(depth, startDepth, board, alpha, beta, xPos, yPos, player)** uses the minimax algorithm along with alpha-beta pruning to simulate *depth* moves in the future and choose the best move for the current state of the board. It takes *depth* which counts down to 0 to exit the recursive search at the base case (along with cases where the game is won or the board is full), *startDepth* to set the final return value to a column number instead of a board score on the outer layer of the recursive tree, *board* which is a simulated future board state, *alpha* and *beta* parameters for pruning moves which will likely never be played, x and y coordinates of the most recent play to update the simulated board state and check for winning moves, and the piece of the player whose turn it is. This function returns the score of the evaluated board state at *depth* != *startDepth* and the column with the best score at *depth* == *startDepth*. This function has a time complexity of $(depth^n)$.

The **AImove(board, topRow, xPos, player)** function calls the *bestmove()* function and uses its return value to provide an animation of the AI's piece moving to the proper column before it is dropped. It takes the values of the current board, the current topRow, the current x coordinate of the piece, and the player whose turn it is (always 'o' in this case). It returns the x coordinate of the move the AI plays. This function has a time complexity of $O(n^2)$ disregarding function calls, otherwise it's $O(depth^n)$.

Testing

Testing for the accuracy of the AI was mostly done using this website. One final test showed that my AI beat level 14 (hardest) AI on this website 3 out of 5 games played. The problem with this testing is that it seems like this website chooses a random set of 3 or so 'useful' opening moves which resulted in both

losing games being exactly the same and 2 of the winning games also being exactly the same. Although these results are not optimal, it would be expected that as the level of play increases, the number of final board states would decrease. Needles to say, the AI beats me 5 games to 0.

One interesting thing I noticed my AI doing was that when there were not many moves remaining and it was obvious that my AI would lose, it would 'throw' the game leading to a loss, when it may have still been possible to win given that the opponent played all of their moves incorrectly. This is definitely because the minimax algorithm assumes that the opponent would make all of the moves that the AI would make if it were in the opponent's position, leading to the other possible moves of the opponent having an embarrassing loss being pruned from the recursive search. I would like to see the AI not give up like this ,but I could not find a solution to this problem without sacrificing the time gained from pruning moves in the early game. I also noticed that the AI on the website would also 'throw' games it knew it would lose as well.

Implementation Trade-offs

While researching the game of 'Connect Four', I came across articles that said 'Connect Four' is a 'solved game' which means it has been proven if both players play perfect moves every time, the player who goes first will win 100% of the time. This information led me to find a website that tells you which moves may be played to guarantee victory. This website also provided an outline of the algorithms used in creating it. I decided instead of learning how to implement a nice GUI for the program like I mentioned in my possible additions section of my proposal, that I would try to implement a transposition table into my code. The idea behind this implementation is that identical board states may be reached from different moves, so if the scores of board states were stored in memory (a dictionary perhaps), no computation would be wasted on finding the score again, since it could be taken from the cache. This would ultimately trade memory for speed. While trying to implement this I ran into many problems such as using too much memory and not being able to remove board states from memory properly to reduce memory usage, so I was unable to find a solution to this problem. Although I was able to implement everything that I said I intended to in my proposal, the time spent on this resulted in only implementing a more efficient search order for more efficient pruning from my possible additions section.

Final notes

I did end up taking your feedback from my proposal submission and found it simple enough to implement the recursive search without using any dictionaries. This, unfortunately, led to my program not implementing any dictionaries (since I was not able to solve the transposition table problem).

If I had more time, I would still have liked to implement a more user-friendly GUI, a statistical output of how the game was played (good moves, bad moves, predicted outcomes, the likelihood of winning, etc.), and of course, anything that would make a better AI given that connect 4 is a solved game. I would also have liked to add more fun features like difficulty settings, colour options, a time limit to moves, etc.

Time complexity and other comments may be found in the python file for the code.

Sources

Minimax pseudocode: https://en.wikipedia.org/wiki/Minimax#Pseudocode

Alpha-beta pseudocode: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning#Pseudocode

Testing website: https://www.helpfulgames.com/subjects/brain-training/connect-four.html

Solved game paper: http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf

Perfect connect 4 solver: https://connect4.gamesolver.org/en/

Perfect connect 4 solver implementation: http://blog.gamesolver.org/solving-connect-four/01-introduction/