ENGG1410: Introductory Programming for Engineers

Mini-Project #1 | "Introduction to C Programming": Simulation of Tic-Tac-Toe with Enhanced AI Player

Eric Cao, Jeremiah George

November 10, 2024

Contents

Overview

Objective

Design and implement a C program that simulates a tic-tac-toe game inside the computer terminal. Over the length of the program, the number of wins and losses are tracked for each side with X's or O's.

The player (i.e. user) enters the size of grid that the tic-tac-toe game will be played on. Then, the player chooses to play with either a) themselves or b) the computer "AI" that has the capability of playing a winning move and preventing the player from winning in their next move; the AI presents some difficulty to the player.

Constraints

- The user enters an integer from 3 to 10 for the number of rows and columns the tic-tac-toe game is to be played on. If the number is out of bounds, the user is prompted to enter a valid size.
    - The user cannot change the size of the grid for the length of the program.
- The starting player is always assigned the character 'O' while the opposing player always plays 'X'.
- The user is prompted to either enter: "1" for playing with themselves (alternating between placing X's and O's), "2" for playing with the computer AI, and "3" to exit the program. The program returns the player to the choice menu at the end of every game.
- During a player's turn, the program asks for two positive integers separated by a space or a new line; the integers (from 1 to size of the board) represent the row and then the column in which the player casts their move. The program prompts the user to retry if the numbers are out of the bounds of the board.
- Every time a move is played on the board, the program outputs the entire board to the terminal.
- At the end of each match, the program outputs the number of wins by 'X' and 'O' achieved during the length of the program.

Planning

Some preprocessor macros must be used by the program:

- **Include header files** `time.h`, `stdio.h`, and `stdlib.h` to use functions in such headers in the program.
- **Define** the maximum size of the board `MAX_SIZE` to be 10.

The code for the simulation is separated into seven functions.

Table 1: list of functions for Tic-Tac-Toe simulation

Name	Description	Prototype
Main	Handles the player's first inputs to set the size of the board and the	None
Check win condition	Check whether player X or player O has won the game by filling a whole row, column, or corner-to-corner	`int checkWin(char [MAX_SIZE][MAX_SIZE], int);`
Check draw condition	Check if the entire board is filled with characters, resulting in a stalemate	`int checkDraw(char [MAX_SIZE][MAX_SIZE], int);`
Initialize playing board	Initialize the board, starting with empty characters that can be filled with X's and O's	`void initializeBoard(char [MAX_SIZE][MAX_SIZE], int);`
Score tracking	Keep track of the number of wins for X's and O's throughout the length of the program (the type of player playing notwithstanding).	`void updateScore(char);`
Display board to terminal	Output the board to the terminal in a manner that can be visually interpreted by the user.	`void printBoard(char [MAX_SIZE][MAX_SIZE], int);`
Player's move	Prompt the user for the row and column to place a character on that board.	`void playerMove(char [MAX_SIZE][MAX_SIZE], int, char);`
AI's move	Make the AI player place a character on the board depending on if a) there is a winning move, b) if the opponent has a winning move, or c) at a randomized position.	`void aiMove(char [MAX_SIZE][MAX_SIZE], int);`

Functions

Check Win Condition

```
int checkWin(char board[MAXSIZE][MAXSIZE], int size)
```

This function evaluates if either player has won on the board. The winning configurations are as follows:

- Any row is entirely filled with one character
- Any column is entirely filled with one character
- Any corner-to-corner diagonal is entirely filled with one character

The function returns:

- 1 (O wins the game)
- 2 (X wins the game)
- 0 (no winner)

When the function reaches the last cell without finding a space and is also able to match the last cell's character with first cell, then the player who placed that character is the winner of the game.

Pseudocode

```
check for (every row, for every cell in row), for (every column, for every
cell in column), for (every cell from top-left to bottom-right), for (every
cell from top-right to bottom-left):
      if (cell contains ' ') break out of loop
      else if (last cell contains 'O'
            and first cell contains 'O') return 1
      else if (last cell contains 'X' and first cell contains 'X') return 2

return 0
```

Check Draw Condition

```
int checkDraw(char board[MAXSIZE][MAXSIZE], int size)
```

This function evaluates if the entire board is full of characters, resulting in a stalemate.

The function returns:

- 0 (a draw has not occurred)
- 1 (a draw has occurred)

Pseudocode

```
check for (every row, for every cell in row): {
      if (cell contains ' ') return 0
}

return 1
```

Initialize Playing Board

```
void initializeBoard(char board[MAX_SIZE][MAX_SIZE], int size);
```

This function resets the contents of the board by filling every position with a space, representing an empty cell.

Pseudocode

```
for (every row, for every cell in row): cell = ' '
```

Updating Score Counter

```
void updateScore(char winner);
```

This function takes the character of the winning player, increments that players win counter by 1, and displays to the terminal how many times each character was the winning character.

Pseudocode

```
character parameter: winner
static integers: x = 0, o = 0

if (winner == x) x = x + 1
else o = o + 1

output -> x, o
```

Display Board to Terminal

```
void printBoard(char [MAX_SIZE][MAX_SIZE] board, int size);
```

This function displays the board to the terminal in a manner that can be visually parsed by the reader. Characters "|" and "-" are used to separate columns and rows, respectively.

Pseudocode

```
for (every row): {
      output -> (newline) (first cell)
      for (every cell in row from 2nd to last): output -> "| " + cell

      output -> (newline)
      for (every cell in row + every instance of "|" used for row):
            output -> "-"
}

output -> (newline)
```

Player's Move

```
void playerMove(char board[MAX_SIZE][MAX_SIZE], int size, char player);
```

This function prompts the player to enter the coordinates of a cell to which they place their character on the board. If the coordinates are out of bounds from the grid, or result in an occupied cell, the player is prompted again until they give a valid answer.

Pseudocode

```
character parameter: player
integer variables: x, y
boolean variable: isValid == false

while (isValid == false): {
      x, y <- received input
      if (x <= 0 or x > size) output -> row is out of bounds
      else if (y <= 0 or y > size) output -> column is out of bounds
      else if (cell at board position row x, column y != ' ')
            output -> cell already occupied
      else isValid = true
}
cell at board position row x, column y = player
```

AI's Move

```
void aiMove(char [MAX_SIZE][MAX_SIZE] board, int size);
```

This function calculates where to place a character that opposes the user player. The function first checks all the rows, columns, then diagonals for any moves prioritized in the following order:

1. Playing winning moves for the AI
2. Blocking winning moves for the player

If none of the above are present, then the AI plays a random available position. The algorithm discerns that a cell has priority if an entire row, column or diagram is filled with the same character except for one cell.

The algorithm relies on the fact that the user's character will always be 'O'.

Pseudocode

```
integer variables: blocking position blockX = -1, blockY = -1
check for (every row, for every cell in row), (every column, every cell in
column), (every cell in corner-to-corner diagonal): {
      if (row contains two empty cells) break
      else if (row contains cells with both characters) break
      else if (characters in row are 'X') {
            place 'X' in missing cell
            return
      }
      else if (characters in row are 'O') store positions in blockX, blockY

if (blockX != -1) {
      place 'X' in missing cell's position
      return
}

integer array: available
```

Conclusion

In this mini-project, Jeremiah and I learned to divvy up sections of the project between ourselves based on what functions we would be writing. The functions served to be a useful tool to avoid repetition in writing code or being more modular in our approach to this project. The use of functions will serve to make our tasks more divisible from now on.