External Documentation for David’s Terrible Tic-Tac-Toe Game

**Problem Definition**

The problem at was to build a tic tac toe game that could be played against the cpu that was fairly stable, following a detailed specification. Text input is through std::in as per specification, not using the C equivalents. The program had to compile on GNU/Linux to get full credit, which I have verified on my own machine as well as Grace.

**Analysis**

In the program, I decided to include the following header files:

#include <iostream> (for I/O)

#include <string> (for std::string)

#include <cmath> (for sqrt)

#include <cstdlib> (for an enormous number of features I could not live without)

#include <ctime> for srand(time(0))

Most variable data types were either chosen by the specification of the program or necessity in data complexity. Common sense dictates not to use floats for flat “states” of things, etc. There were several places I felt I should change ints to unsigned ints, and indeed I did in some places, but I’m feeling too tired to do that right now.

For variables in main() I will run down variable types and why I chose them, as well as explaining how they are used

*char board[][]* ***-*** Specification said to use char. This Array holds the state of every square on the board. There are three states: X, O, and space.

*int scoreBoard[][]* - Specification said to use int. This Array holds the win/loss for every turn- and if both sides tie, then it is a loss.

*unsigned int turns\_played* – It never has to be less than zero, so I made it unsigned. This counts how many turns have been played so far in the current game.

*unsigned int games\_played* – Ditto, this counts how many games have been played in the current set

*unsigned int iterator –* Ditto, this is a randomly-offset-by-1-or-2 version of turns\_played. Its only purpose is to allow for using a shifted-by-one value to allow the CPU to play first sometimes.

Formulas

The math in this assignment was not very heavy, as such it is difficult to figure out what should be mentioned and what isn’t worth mentioning. Stuff like this:

\* CPU Move position = rand()%BOARD\_WIDTH, rand()%BOARD\_HEIGHT if it is valid

\* Iterator before any turns have been played = rand()%2 + 1

A lot of complicated formulas were used for the CPU’s AI (Disabled by default, can be re-enabled by setting CPU\_USE\_AI to 1)

However, I used a considerable amount of (possibly pointless) math in designing the CPU’s AI. I will explain it very roughly here:

\* The CPU uses a 2D array of integers to store how much it “likes” the idea of moving to any particular spot. Various algorithms are used to add “preference” to a position. Factors such as ability to win on the turn, blocking the enemy’s streaks, increasing their own streaks, and proximity to other pieces are factored in. Number of spaces it is possible to move into is also factored in, which is why the AI will usually move into a square that is not very close to the edge with large board sizes.

All of these are done with formulas which are not so much carefully crafted as they area slapped together.

An optimal algorithm was not used, nor even quite my best (because I procrastinated until the very last day and didn’t have enough time) however I feel the CPU’s ability to accurately represent the skill of most Tic-Tac-Toe players, who are usually quite young.

Formulas in the AI:

if(checkWin(sim\_board, PLAYER\_USES\_X?'O':'X'))

board\_evaluation[i][j] += 1000000;

If the AI can win by moving to a place, 1 million is added to the preference to ensure the CPU will not move to a losing square.

if(checkWin(sim\_board, PLAYER\_USES\_X?'X':'O'))

board\_evaluation[i][j] += 100000;

If the Player can win by moving to a place, 100,000 is added to the preference. If the CPU cannot win, and the player can win by moving to a space, the CPU must move there. If there are multiple such spaces, it picks one.

sim\_board[i][j] = PLAYER\_USES\_X?'O':'X';

board\_evaluation[i][j] += (GetLongestStreakForPlayer(sim\_board, PLAYER\_USES\_X?'O':'X') + 1) \* 200;

The longest streak that will exist for the CPU if it chooses to move there is factored in. This has multiple effects: If the CPU can move to a place and thereby create a larger maximum streak, it will, but if the CPU can move to a place with a streak and increase it by 1, but will not increase the largest streak, it has no special preference.

int distance\_from\_wh\_to\_ij = sqrt((w - i) \* (w - i) + (h - j) \* (h - j));

board\_evaluation[i][j] += 30/(distance\_from\_wh\_to\_ij + 1);

The CPU will prefer to move to places close to other pieces rather than randomly on the board somewhere. A similar version of this formula is used for blank spaces to make the cpu prefer spaces toward the middle on its first move.

if(board\_evaluation[\_i][\_j] > largest\_evaluation){

largest\_evaluation = board\_evaluation[\_i][\_j];

largest\_eval\_X = \_i;

largest\_eval\_Y = \_j;

} else if(board\_evaluation[\_i][\_j] == largest\_evaluation){

//Make the CPU ever-so-slightly random

int n = rand()%(100);

if(n >= 90){

largest\_evaluation = board\_evaluation[\_i][\_j];

largest\_eval\_X = \_i;

largest\_eval\_Y = \_j;

}

}

The CPU will always move to higher evaluations/preferences over lower ones, but when there is a tie it will randomly pick one of the 2.

**Design**

The main algorithm for the tic tac toe game was practically written in the assignment.

While there are stile games left

Let player 1 move,

check if he won

Let player 2 move,

check if he won,

rinse and repeat until either there are no moves left to make or someone has won.

My program is quite long and has a number of functions, I will cover only the most interesting:

CheckWin:

Check for horizontal wins

Check for vertical wins

Check for negative diagonal wins

Check for Positive diagonal wins

If there was a win anywhere, return true for winning when you find it

GetLongestStreakForPlayer:

Check Horizontal streaks

Check Vertical streaks

Negative Diagonal streaks

Positive Diagonal streaks

computerTurn:

If the CPU is not going to use the AI:

Generate random moves until a valid one is made, then move there

If the CPU IS going to use the AI:

Clear preferences

Check for winning moves

Check for blocking wins

Check for longest streak

Check for Ending Streaks

Check for Proximity to other pieces

Make the move with the highest preference

If the move with the highest preference was invalid somehow, move randomly.

Return control to main

DrawBoard:

For every row:

Draw the seperator above

Draw the lines between the Xs and Os whilst drawing the Xs and Os making sure to place the line at the end

Draw the bottom line

PlayerTurn:

While no valid inputs have been made:

if the user has failed twice, close the program

Ask for input, and if the user has failed once, teach them how to write inputs

if the input is valid, the move will be made and control is returned to main.

Main:

Declare a bunch of variables

While the set isn’t over

While the match isn’t over

let each player play one after the other

announce victory, loss, or tie

Give the score and overall result

End Program

**Implementation**

The program was coded entirely on my hand-built Siduction (Debian Buster/Sid GNU/Linux) x64 PC with an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 8 virtual cores and 4 physical cores. 32 Gigabytes of RAM. It has an ASUS Motherboard so I guess you could say it is an ASUS computer.

Program was tested on this machine but was also tested on grace. Proof that the program was compiled on grace exists (See attached file “Proof of Compiling on Grace”).

Program was compiled and linked by g++ (Debian 8.2.0-16) 8.2.0 as reported by g++ --version.

Unusual situations encountered while coding were usually my own fault in poorly typing things in. I had forgotten how to pass arrays as arguments to functions and idiotically typed out int[][SCORE\_NUM\_GAMES] scoreBoard and the like a few times (I’m still very displeased that I cannot use double pointers for 2d arrays on the stack). There is indeed a quirk with typing in a letter instead of a number, which I had originally planned to capture for, but decided that using such code would probably end up using functions I wasn’t allowed such as std::string.c\_str and atoi, or would be incredibly tedious. I had planned to do error handling however I am not sure if I would lose points so I didn’t use it. Currently, typing in a letter and pressing enter will cause the playerTurn function to believe that you want to quit the program, as it repeatedly goes to the same cin statements over the while loop causing the sentinel functionality I designed to trigger.

**Tips**

1) Try fiddling with the options at the top defined with #define

2) check out the AI

3) check out the AI on large boards

Thank you for reading.