Jacob Tucker
CS 482 Artificial Intelligence
Project 1
AI for tic tac toe using Minimax algorithm

Task: Given a tic tac toe board, write a program that will determine the optimal move for the next player.

Board Setup: The tic tac toe boards are a 3 x 3 matrix with 0's representing empty spaces, 1's representing x's and -1's representing o's. The program loads the board files by reading them into a 3x3 int matrix. The program saves the board state to the given output file path after making a move on the matrix.

All of the functionality of the decision logic is in the file tictac_turnin.cc. The only other files modified is the tictac_support.h which I added a struct of two integers; row and col. These help to keep track of a specific board location.

The main function for making a move is the make_move function. Another big function is the miniMaxMove function. There are also utility functions; checkWin, movesLeft and makeWinningMove.

Int checkWin(int player, int board[][3])
checkWin takes a player, which can be -1, 1 or 0. checkWin scans the board for winning conditions for the player. It returns scores based on which player was passed to it and which winning condition exists. It will return 10 x the player passed to it (10 or -10). If no win condition exists, it will return zero. This is used in the main minimax function for generating heuristics for the possible game states.

Bool movesLeft(int board[][3])
movesLeft is a simple function that takes a board matrix and iterates through each position checking for zeros. It will return true if one is found indicating that there are available moves left on that board state. If no zeros are found on the board it will return false

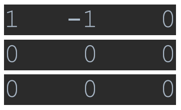
Bool makeWinningMove(int player, int board[][3])
This function scans the board state passed to it for any moves that would result in an immediate win for the player passed to it. This function ensures that swift victory will be taken in such a situation. Because tic tac toe does not have a large number of endgame states, it is more efficient to hardcode each of these states instead of running the recursive algorithm.

Int miniMaxMove(int board[][3], bool isMaximizer, int player, int &totalScore, int turns)
This function is the core of the decision logic for the program. This function determines a minimax score for an associated move passed to it in the board state. It recursively calls itself to check every possible outcome from the given board state. When it reaches terminal nodes, it will return a score of 10, if it is a win state for the passed player, -10 if it is a win state for the

opposing player, or zero if it is a draw. As the recursive functions return, it will add or subtract the number of turns it has taken to reach that state. It will add turns to the score if it is a minimizing turn, or subtract turns from the score if it is a maximizing turn. This causes the AI to factor in the number of turns it will take to achieve a winning condition or a losing condition. It will choose the fastest path to a win condition or the slowest path to a losing condition. totalScore is passed by reference so that the scores can be accumulated and not lost when out of scope.

Int make_move(int board[][3])
Make_move puts it all together. The function takes a board state and implements the game logic by utilizing the other functions. First it determines which player’s turn it is. Then it runs checks to make sure the game board state is playable. It does this by checking that the player’s have had the correct number of relative turns, checking for pre existing win conditions, and making sure there are available moves left. Then it will check to see if there are any immediately winning moves available for the player.
After this, it will iterate through the board and get a minimax score for each available move. If the score is better than the previous score, it will update that as the best move. I have also included a separate matrix to hold the values of these scores so it is easy to see how the available moves were scored.
Once the best move has been determined, it will make that move. It will also print the game state after the move was made along with the board of minimax scores. Finally, it re-checks for win conditions or draws.

The program is working correctly with many different input states. The program will attempt to make the move that will lead the fastest to a win condition. Or, if there are no win conditions predicted, it will choose the move with the slowest path to a losing condition.
For example, given this board. It determined that the greatest chance of winning what the bottom left corner, with a minimax value of 5. This was found by finding a terminal node with a win condition, which returned 10, then subtracted the number of moves it took to get to that game state, which was 5. Essentially it believes that it can win with this move in about 5 moves.



```
1    -1     0
0     0     0
0     0     0
```

From the minimax value matrix, it can be seen that there were other options for win conditions with a value of 2. The -999s are there to make it obvious that they were not available moves. However this likely meant that the win condition took 8 turns to reach and was therefore not the optimal solution,

```
-999    -999    -9
2        2      -9
5       -5      -8
```

In the next example, we can see the opposite approach. The AI is choosing the path of slowest loss when there are no good moves. From this state, the AI made the move to the bottom right, which blocked the 1 player from winning in the next turn, even though it determined that it was likely going to lose the match eventually.

```
1    -1     0
0     1     0
0     0     0
```

You can see from the minimax values that any other move would have resulted in an immediate loss condition. That is -10 + 0 turns.

```
-999    -999    -10
-10    -999    -10
-10    -10    -9
```