Computer vs User Tic-Tac-Toe

Michael LaBarbera, M.S. Student, Department of Applied Math and Statistics
Shahzad Patel, M.S. Student, Department of Electrical and Computer Engineering

Objectives

Throughout the course of this project our objective was to make an interactive game of Tic-Tac-Toe where the user plays the computer at 3 levels of difficulty (Easy, Medium, and Hard). We implemented the different difficulties by coding specific Tic-Tac-Toe strategies for each level and making each one more intense and harder to beat than the previous level.

Contributions

As for the contributions to the project, we started by both researching various strategies and decided which direction we wanted to move in for the project. Shahzad was responsible for formulating the architecture and flow of the program and writing the code. Once the code was written, we ran basic tests to see if there were any glitches or run-time errors in the game. Once that was attended to, Michael tested each level of difficulty in depth. This involved trying every permutation, at every level and in situations of first and second user move. Michael then coded the results to be presented in multiple pie charts. We then worked as a team to fill in the holes and try and fix all the bugs that we could.

Techniques and Trials

Our approach to the interface of the game was to create a brute force Computer player of 3 levels of difficulty (Easy, Medium and Hard). Although there are algorithms (for e.g. the minimax algorithm) that we could've implemented, we chose the traditional approach of prediction (basic if-else statements) for the Computer move and to predict the User move.

Our first step was to define the general functions that would be required by all the difficulty levels. Following are the said functions that were defined along with their role in the program and the required inputs for each: -

1. displayBoard: - This function shows the layout of the board and the position of each index that would be entered by the user to play its move. As input, it takes the list containing the positions at which the user and computer have played their respective moves. Fig 1. shows the mentioned layout.

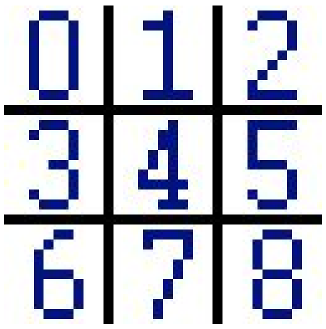

Figure 1

2. checkWin: - After every move this function is executed to check if either the user or the computer have won the game. As input, it takes the list containing the positions of ‘X’ and ‘O’ played by the user and computer and the mark (‘X’ or ‘O’) it is supposed to check for the win.

3. checkDraw: - After every move this function is executed to check if either the user or the computer have no possible moves to win the game. Just as the displayBoard function, it takes the position list as input.

4. getBoardCopy: - For the computer to win the game, it needs to know the positions at which a win would be possible for itself and the user. For that purpose, a duplicate board is created solely for the mentioned purpose so that no changes need to be made to the original board. Like the checkDraw function, it takes the position list as input.

5. testWinMove: - This function checks for a win move for the user and computer. The computer can benefit from it since this function checks for a win move for the user so that the computer can prevent the user from winning. Like the checkWin function, as input this function takes the list containing the positions of ‘X’ and ‘O’ played by the user and computer, the mark (‘X’ or ‘O’) it is supposed to check for the win and the position that would lead to a possible win for the User and the computer.

[**NOTE**- We first call the functions getBoardCopy and testWinMove for each square to check if the computer can win, then for each square to check if the player can win, so the computer can block the user.]

Our next step was to create our own algorithm for each level of difficulty and implement them in python.

The algorithm for the level Easy function (getComputerMoveEasy) is as follows: -

1. Check for a square that CPU can win on
2. Check for square that player can win on
3. Move on a free corner
4. Move on free center
5. Move on a free side

A computer player at this level should be easy to beat. Therefore it is created in its tutorial mode wherein it uses the **random** library of python for the Computer to place its marker at a random position. Due to its simplicity, this Computer player can outplay only a novice or a "random" player.

Now that we have a reasonable Computer player to play the game against, we ourselves played the game to see how the computer works and how it can be beatable by going first and second. So, we observe that if we play as an expert, we win majority of the games, but if we play as a random player, we see that the computer doesn't play the center position when it should so as to result in a draw or a win for the computer, indicating that a few tweaks are required to make it better.

Along with the above-mentioned tweaks we also create a function to predict a fork move for the computer and User. The concept of a fork move is when a player makes a move that gives them two opportunities at a three in a row. It's effectively a "checkmate in 1" situation that means the other player can't block both three in a row possibilities, and therefore loses. Fig 2. shows an example of a fork move wherein, either of the moves indicated by green crosses would create a fork for crosses.

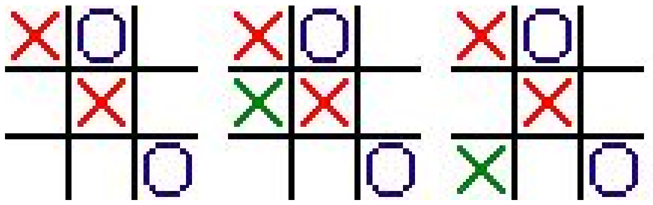

Figure 2

This concept inspired us to create an algorithm for the level "Medium" of this game. The algorithm for its function (getComputerMoveMedium) is as follows: -

1. Check for a square that CPU can win on
2. Check for square that player can win on
3. Check for a fork opportunity for the CPU
4. Check for a fork opportunity for the User
5. Move on a free corner
6. Move on free center
7. Move on a free side

When we test this level, we see that it gets slightly harder to beat the computer. Although after playing a little over 50 games, this level too becomes easy to win.

A final attempt at making the game unbeatable, "attempt" being the keyword here, was to create a Computer player that would make a move that gives itself a fork opportunity. If not, it should at least block the moves that give the player a fork opportunity.

Following is the algorithm for level "Hard" function (getComputerMoveHard) of the game: -

1. Check for a square that CPU can win on
2. Check for square that player can win on
3. Check for a fork opportunity for the CPU
4. Check for a fork opportunity for the User but choose to force a block if there are two potential forks before the computer creates a fork opportunity for itself.
5. Move on a free corner
6. Move on free center
7. Move on a free side

After the codes for the above-mentioned functions and algorithms are written, we write general game code which displays the basic interface of the game and calls the necessary functions as the code is executed.

The technique that was used to graph the results was creating a Pandas series, with the number of wins, losses, and draws that occured at each level of the game. We then indexed the series to label the numbers to their correct groupings. We then named the series to explain exactly where the proportions of wins, losses, and draws came from. Lastly, we used the plot function in order to plot the data.

Trials

As we conducted the trials we found that as the difficulty level increased the number of trials we were able to do decreased. This was due to the fact that the code was designed in a way that the computer played with a specific strategy each time and there were not many iterations that we could make with different possible games. To solve this problem, we implemented the positionally specific trials when going second and at least 30 different trials when going first in order to get a general idea of the win percentage at each level.

Results

Throughout this project we set out to create an interactive Tic-Tac-Toe game that had three levels of increasing difficulty. Once our code was complete, we conducted trial runs for each of the levels, going both first and second, in order to test if the code was successful.

Easy

We began my trials by choosing to go second on the first level, “Easy.” After exhausting all possible outcomes for the first two move combinations, we ended up with 54 wins, 6 losses, and 4 draws. This means that the player wins approximately 84% of the time on this level when they go second (Figure 3).

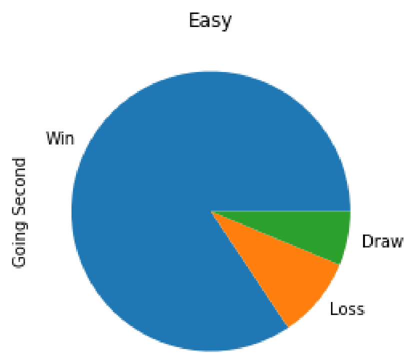

Figure 3

Next, we played through all possible outcomes for the first two moves when going first. The results of this were 69 wins, 1 loss, and 4 draws. This means that the player wins approximately 93% of the time on this level when they go first (Figure 4).

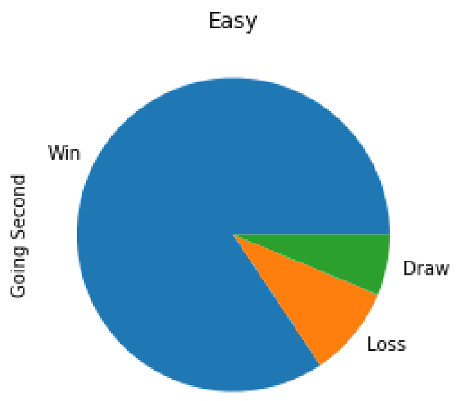

Figure 4

Lastly for this level, we determined the cumulative totals, which turned out to be 123 wins, 7 losses, and 8 draws. This means that the player wins approximately 90% of the time on this level (Figure 5).

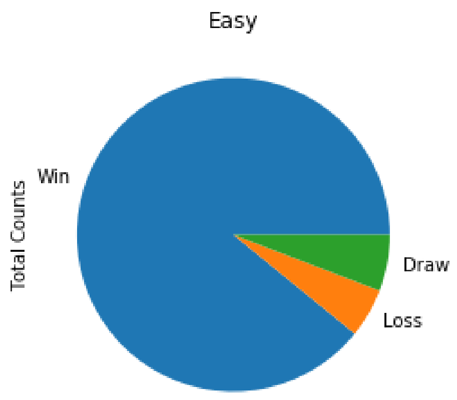

Figure 5

Medium

We began my trials for this level by choosing to go second on "Medium" After 25 trials, we ended up with 12 wins, 8 losses, and 5 draws. This means that the player wins approximately 48% of the time on this level when they go second (Figure 6).

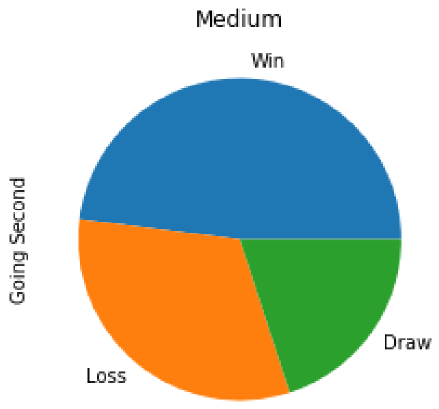

Figure 6

Next, we played through 33 trials while choosing to go first and the results of this were 10 wins, 3 loss, and 20 draws. This means that the player wins approximately 30% of the time on this level when they go first (Figure 7).

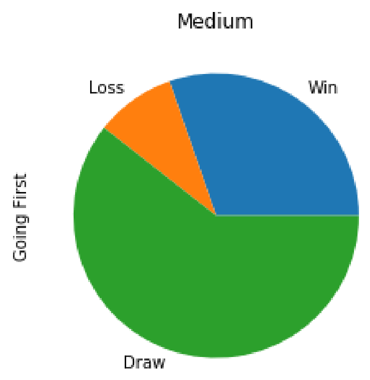

Figure 7

Lastly for this level, we determined the cumulative totals which turned out to be 22 wins, 11 losses, and 25 draws. This means that the player wins approximately 37% of the time on this level (Figure 8).

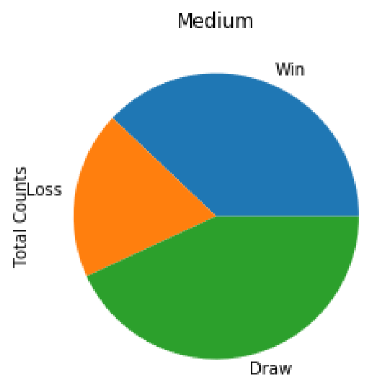

Figure 8

Hard

We began my trials for this by choosing to go second on the first level, “Hard” After exhausting all possible outcomes for the first two move combinations with our code, we ended up with 0 wins, 6 losses, and 2 draws. This means that the player wins approximately 0% of the time on this level when they go second (Figure 9).

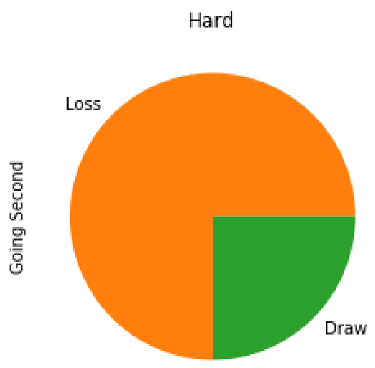

Figure 9

Next, we played through 33 trials while choosing to go first and the results of this were of this were 5 wins, 10 loss, and 18 draws. This means that the player wins approximately 15% of the time on this level when they go first (Figure 10).

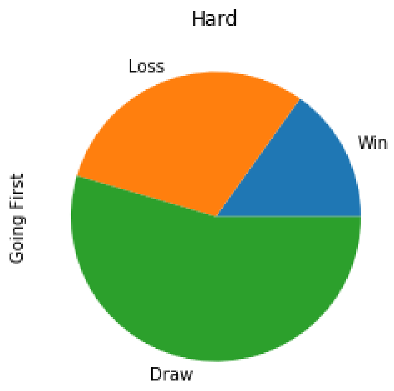

Figure 10

Lastly for this level, we determined the cumulative totals, which turned out to be 5 wins, 16 losses, and 20 draws. This means that the player wins approximately 12% of the time on this level (Figure 11).

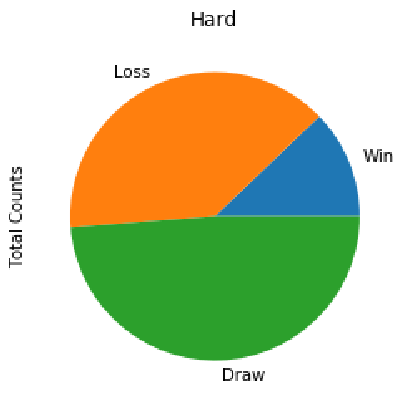

Figure 11

Conclusion

After the numerous trials, we were able to determine that our code was successful in completing the task we set out to complete, which was to create a User vs. Computer Tic-Tac-Toe game with three levels of difficulty. As shown in the results, as the levels get harder the win percentages for the User decrease. Also, as the difficulty increased, the number of trials decreased because the Computer was playing with a strategy rather than going in a random, open spot. This led to not as many possible trails to complete, which in turn, makes the game harder to play and win.