Graphical Tic-Tac-Toe

Salaj Ganesh

Table of Contents

The Idea

Background

The original Tic-Tac-Toe program was written by me and a couple of my friends back in 2018 for a 24-hour code-thon competition. At these events, we always strove to push our knowledge of programming and try new and interesting things. This project was our first C++ program with a graphics and visuals. However due to our inexperience, this program had plenty of drawbacks. The computer opponent in our program would randomly choose moves and had no strategy. This result in the player easily winning every time. The graphics implementation in our code was also messy and very hard to follow. And while our program had graphics, there was no way for a user to directly interact with them. Instead, they had to rely on a terminal to make their moves. Because of these flaws, our results in the competition were not as impressive as we would have liked. For the next competitions we moved away from C++ and started to learn the Unity game engine build our own 3D games.

Update

A couple of years later in college, I started to get interested in artificial intelligence and machine learning. As a fun side project of mine, I decided to revisit this Tic-Tac-Toe program I wrote in high school. Before I could work on updating the basic computer opponent to an intelligent algorithm, I first had to update and fix the original program. I found the original graphics API to be very convoluted and lacking in the features I desired so I rewrote the entire program into Python and use the Pygame package to handle the graphics. Much of the game logic could be translated between these two languages quite easily and the graphics conventions were very similar as well. During this rewrite, I also split the program into many smaller subfunctions and classes, making it easier to use and more modular. Once I was satisfied with the rewrite and optimization, I began working on implementing the machine learning algorithm. For a simple game like Tic-Tac-Toe, a reinforcement learning algorithm would probably suit it best. While researching the algorithm, I learned something about the game of Tic-Tac-Toe. If both players always make optimal moves, the game always ends in a draw. When I learned this, I lost a lot of my motivation and eventually moved on to other projects.

The Code

Tic-Tac-Toe Class

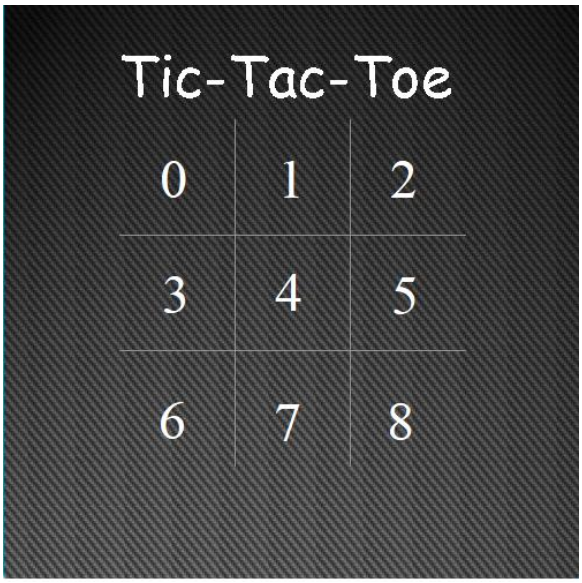
The Tic-Tac-Toe class is stored in the *game.py* file. This class handles creating the empty board, drawing the graphics, updating the board/graphics, and checking for winners.

```
import pygame

class tictactoe:
    def __init__(self, graphics = True):
        self.board = [0, 0, 0, 0, 0, 0, 0, 0, 0]
        self.player = 1
        self.graphics = graphics
        if (graphics):
            self.drawgraphics()
```

Code Snippet 1: Initializing Tic-Tac-Toe Game

When creating a new Tic-Tac-Toe class, it initializes a new empty board. The board is orientated in row major order with the smallest index representing the top left of the board and the last index representing the bottom right. (See Picture 1 below)



Picture 1: Board Layout

When initializing a new class, only one optional argument is requested. The *graphics* boolean determines whether to draw the GUI on the screen. When playing with human players a GUI will be needed for them to interact with the game. However, to run thousands of simulations between computer algorithms, it may be preferred to not draw a GUI. Depending on the value of the *graphics* boolean, the code may then initialize the Pygame package and draw the screen. Once the screen and board are drawn, it will then wait for one of its subfunctions to be called.

```
def move(self, idx):
    if (self.board[idx] == 0):
        self.board[idx] = self.player
        self.player = self.player%2+1
        if (self.graphics):
            self.update()
        return self.player
    else:
        return -1
```

Code Snippet 2: Making a move on the board

The *move* subfunction takes in an index on the board as its input argument. It checks if the spot is empty and places the mark of the current player. If the move is invalid or already taken it would return a -1. After marking the board, it will move to the next player and return the number representing this player (1 for player one and 2 for player two).

Game Result	checkwin() return value
No Winner	-1
Tie	0
Player 1 wins	1
Player 2 wins	2

Table 1: checkwin return values

```
def checkwin(self):
        winner = -1
        for i in range(3):
            if (self.board[i*3]!= 0 and self.board[i*3] == self.board[i*3+1] and
self.board[i*3+1] == self.board[i*3+2]):
                winner = self.board[i*3]
            if (self.board[i]!= 0 and self.board[i] == self.board[i+3] and
self.board[i+3] == self.board[i+6]):
                winner = self.board[i]
        if (self.board[0]!= 0 and self.board[0] == self.board[4] and
self.board[4] == self.board[8]):
            winner = self.board[0]
        if (self.board[2]!= 0 and self.board[2] == self.board[4] and
self.board[4] == self.board[6]):
            winner = self.board[2]
        message = ""
        if (0 not in self.board and winner == -1):
            message = "Tie"
            winner = 0
        if (winner == 1):
            message = "Player X Wins"
        elif (winner == 2):
            message = "Player O Wins"
        return winner
```

Code Snippet 3: Checking for Winners (Code simplified from file)

The final major subfunction in the Tic-Tac-Toe class is the winner checker. This function should be called manually after every valid move is made. It scans the board for a winning placement and returns a number accordingly (Consult Table 1 above). With *graphics* enabled, the check win function will also draw the winning line on the GUI and print the result of the match. This graphics code has been omitted from Code Snippet 3 above.

Human Class

The *human* class is stored in a file called *player.py*. This class is responsible for creating a new human player and translating mouse clicks on the GUI to inputs the Tic-Tac-Toe class can understand.

```
import pygame

class human:
    def __init__(self, player = 1):
        self.player = player
        self.type = 'human'
```

Code Snippet 4: Creating a New Human

When creating a new human player class, the *player* number is requested. When no player number is provided, it will default to player 1.

```
    def move(self, board):
        if (0 not in board):
            return -1
        pos = pygame.mouse.get_pos()
        i = self.poscalc(pos)
        if (board[i] != 0):
            return -1
        return i
```

Code Snippet 5: The Human Makes His Move

The *move* subfunction in the *human* class should be called when it's that players turn, and the GUI registered a click. It requires the current game board as an argument to check the validity of a potential move. This subfunction will rely on the Pygame package to get the position of the mouse click. The mouse click will be returned as a tuple of length 2 corresponding to the x and y position of the mouse click. We then use these coordinates to determine which square the player clicked on the board. If this area is invalid it will return -1 otherwise return the board index corresponding to the click.

CPU Class

The *CPU* class is stored in a file called *computer.py*. In many ways the *CPU* class is very similar to the *human* class. It is initialized almost identically and uses the game board to make a move. However, the way it calculates its move is very different.

```
import random
import numpy as np

class cpu:
    def __init__(self, player = 2, dumb = False):
        self.player = player
        self.type = "cpu"
        self.dumb = dumb
```

Code Snippet 6: Building a New Computer

Like the *human* class, the *CPU* class utilizes a *player* argument to decide if it's going first or second. By default, the code assumes it is going second. The *dumb* boolean is used the determine if the CPU shall use an algorithm to make a move or choose a move randomly. We can see this in the code snippet below for the *move* subfunction.

```
    def move(self, board):
        if (self.dumb):
            return self.moverandom(board)
        for i in np.arange(9):
            if (board[i] == 0):
                _ = game2.move(i)
                winner = game2.checkwin()
                if (winner == self.player):
                    return i
                for j in arr:
                    if (board[j] == 0):
                        _ = game2.move(j)
                        winner = game2.checkwin()
                        if (winner != other.player):
                            return j
        return self.moverandom(board)
```

Code Snippet 7: The Computer Moves (Code Simplified from File)

If the *dumb* boolean is set, the *move* subfunction would immediately call another subfunction to make a random move. However, if the *CPU* class is set to use the algorithm it would continue normally.

This algorithm is a very basic Tic-Tac-Toe strategy. Its choice is based on two factors, whether it can win on this turn or its opponent it about to win on the next turn. If there is a winning move for itself present on the board it will take that move and win the game. If it detects a winning move for its opponent, it will make a move to try and block it.

The Run

Playing a Game

The code for playing a quick match with a GUI is provided in the file *main.py*. With the necessary software installed on your computer, simply run the file through Python. As of the current version, changing any game settings requires editing this file in a text editor.

```
def run(game, player1 = 'human', player2 = 'cpu'):
    running = True
    turn = 0
    done = False
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            if event.type == pygame.MOUSEBUTTONUP:
                if (done):
                    game = tictactoe()
                    done = False
                    turn = 0
                else:
                    i = players[turn].move(game.board)
                    turn = game.move(i) - 1
                    winner = game.checkwin()
                    if (winner != -1):
                        done = True
        if (players[turn].type == 'cpu' and done == False):
            i = players[turn].move(game.board)
            turn = game.move(i) - 1
            winner = game.checkwin()
            if (winner != -1):
                done = True
```

Code Snippet 8: Running the Game (Code Simplified from File)

The *run* function takes in three arguments, with two of them being optional. The *game* argument is the Tic-Tac-Toe class that was initialized earlier. The other two optional arguments are strings representing the player types for player 1 and 2. These arguments can be changed as desired, human versus human, CPU versus CPU, or CPU versus human where the CPU goes first.

Simulating Games

The code for simulating multiple games is provided in *simulation.py*. By default, the code is set the run 1000 games between two computers, one using the win or block algorithm mentioned above and the other randomly choosing moves. Changing any of these parameters requires editing the file in a text editor. Once all the games are simulated it will print out a bar graph depicting the result of all the matches.

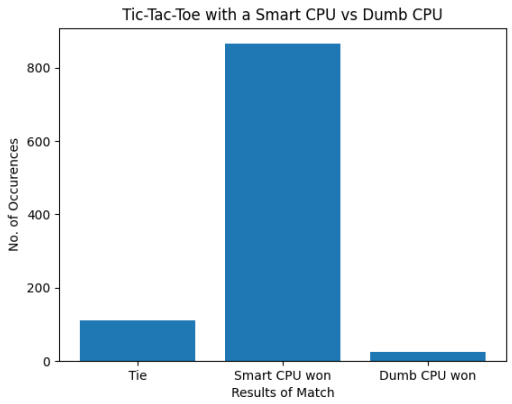
```
def main():
    results = [0,0,0]
    players = [cpu(player = 1, dumb = False), cpu(player = 2, dumb = True)]
    for i in tqdm.tqdm(range(epochs)):
        game = tictactoe(graphics = False)
        winner = run(game, players)
        results[winner] += 1
    print("Player 1 winrate: ", results[1]/epochs)
    tags = ['Tie', 'Player 1 won', 'Player 2 won']
    fig = plt.figure()
    plt.bar(tags, results)
    plt.show()
```

Code Snippet 9: Simulation Code

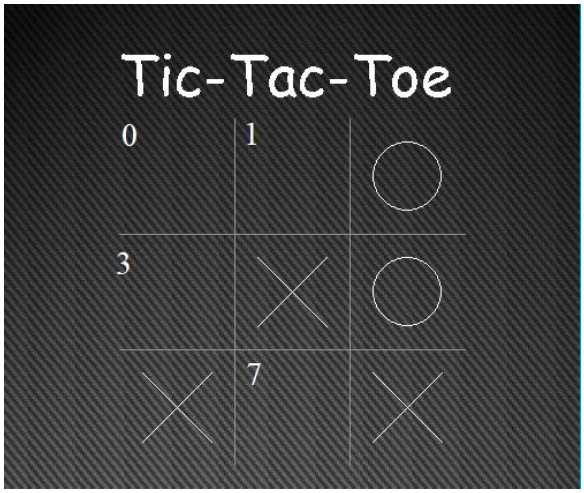
The variable *players* is a list of length 2 that contains the player classes for player 1 and player 2. Changing this variable will change the type of computer algorithms used in simulation. Since the simulation is expected to run for a thousand iterations or more, any *human* classes present in the *players* list will cause the simulation to return an error.

The Analysis

This win or block strategy is quite effective against a dumb computer player (win rate of 87.8%). The detailed result of this algorithm can be seen below in Picture 2.



Picture 2: Results Using Algorithm



Picture 3: Sample Game Board

However, it can be quite easy for a competent human player to outsmart this algorithm. Since the moves up to the block are random, it is quite easy to force a position with multiple paths of victory. Consider the scenario depicted by Picture 3 above. Player X can win in his next turn by playing either square 0 or square 7. If the computer was Player O, it would be forced to choose one these locations and surrender the win to Player X. A more optimal strategy is needed to prevent this happening in the first place.

This is where I believe a better algorithm is needed. Using a true reinforcement learning or min-max algorithm would greatly improve the performance of the CPU.