PYTHON AI

Build a noughts and crosses playing AI

Inspired by *WarGames*, **Andrew Smith** avoids thermonuclear war and proposes an AI solution for tic-tac-toe instead.

OUR EXPERT

Andrew Smith is a software developer at NHS Digital, has a Bachelor's degree in Software Engineering and an MSc in Computer Networks (Mobile and Distributed).

QUICK TIP

The source code for this project is located on the LXFDVD called tictactoe.py

The solution presented in this article was first inspired by the movie *WarGames* (1983) where strategy games such as poker, backgammon and noughts and crosses were featured in the movie that depicted AI opponents not only playing against human players but also having the ability to figure out how to beat them. The AI opponent built for this project mainly uses the minimax[1] AI algorithm to help beat a human opponent. In addition, various YouTube videos were viewed for common implementation techniques and approaches already attempted by others.

The aim of this project was to create an AI player that would prove very difficult to beat for a human player even though not completely impossible. In the end, to get this projected to a reasonably completed state, various strategies were implemented so that the AI player was very difficult to beat. In addition to using the minimax algorithm, various scenarios have been considered, such as where the human player would likely start the game from and how that starting position would affect the chances of the AI player winning or losing. We have also considered various aspects that were highlighted in several different YouTube videos on creating a tic-tac-toe game with an AI player using the minimax algorithm.

From the min to the max

The minimax algorithm is an algorithm (*you don't say – Ed*) that can be used in turn-based strategy games such as chess, draughts or in this case tic-tac-toe. A minimax algorithm can be used to predetermine possible outcomes before or after a move has been taken by a player. A score is given to each predicted outcome, usually a high or low score to identify a winning or losing move respectively. The minimax algorithm can be used to predict at least two move outcomes that may end in a game state (win, lose or draw).

The tic-tac-toe game has been set up in a way that allows a human player to play against an AI player, where by default the human player goes first when the program is run/executed. The human player will select their move using a mouse, which will then be displayed on the tic-tac-toe board. After that the AI player will do some processing and then perform its move on the tic-tac-toe board. This process will continue until there is an end-game state reached, for instance the AI player or human player has won the game or the game has ended in a draw state. The program can be quit at any time by closing the window the game is presented in.

Process of execution

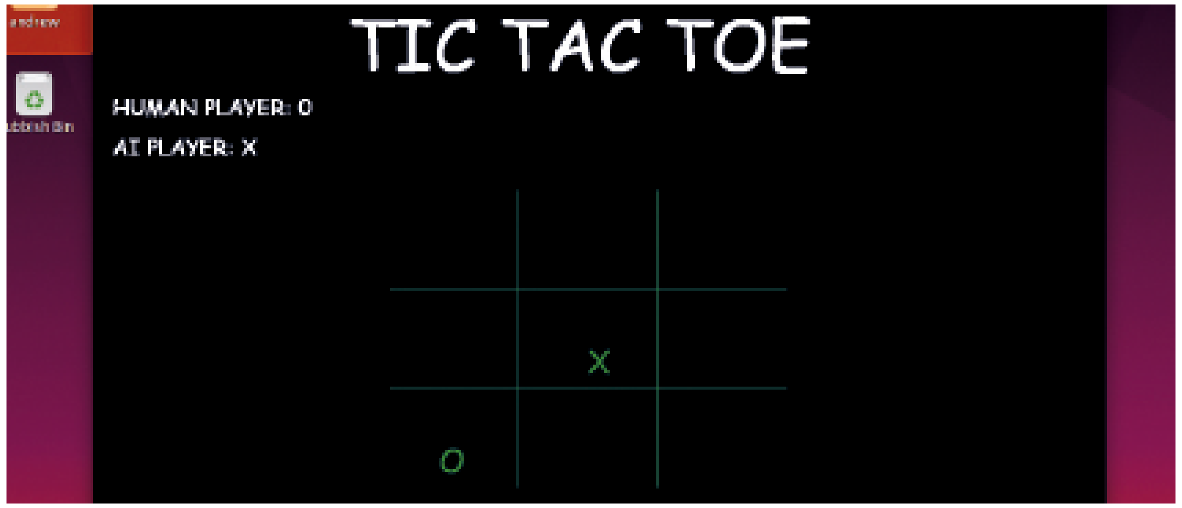
The first move of the AI player is dependent on the opening move of the human player. After many testing runs, it was found that the minimax algorithm didn't have much of an impact on the first two moves of the game – the minimax algorithm is only engaged after the human player undertakes their second move. It was also found that by undertaking this approach, the minimax algorithm didn't have to do as much processing, as there were fewer possibilities to process.

After the first two moves of the game have been taken, the minimax processing algorithm is engaged until an end terminal game state is reached (win, lose or draw). The processing the AI player does can be seen in the console window as shown in the screenshot that's on page 95:

From the processed outcomes of the AI Player, priority is given to outcomes that have a score of -1, then outcomes that have a score of 1, and then lastly outcomes that have a score of 0. The AI player was written to evaluate threats to success first and then evaluate possible advantage/win scenarios after. Even though threats are interpreted first, if it is evaluated that a move would result in a win for the AI player then the AI player will undertake that move.

The AI player plots its moved based on the opening move made by the human player.

1) https://en.wikipedia.org/wiki/Minimax



If the human player selects a corner start position, the AI player takes the centre square.

Preparation and setup

The script can be executed both on Windows and Linux machines. However, before the script is executed, Python 3.6+ to Python 3.7.3 needs to be installed, along with Pygame version 1.9.6. Python for both Linux and Windows can be downloaded from the following location: **www.python.org/downloads**. Pygame for both Linux and Windows can be installed by following the instructions located here: **www.pygame.org/wiki/GettingStarted**.

If you are executing the script on a Linux machine, which as you are reading this article you most likely are, then remember to add an executable permission to the file, for example: `chmod +x tictactoe.py`. After this, the script should be able to be executed with the command `./tictactoe.py`, providing that both Python and Pygame have been setup and installed correctly.

As with any other program or script, the library files/modules are included first.

```
import pygame
import random
import copy
from time import sleep
from pygame.locals import *
```

Some of the most relevant modules to include for this project listed above are *pygame*, *random*, *copy* and *sleep* from the time module. To those that are new to Python and Pygame in general, the *pygame* module must be included at the start of each Pygame project to enable the features and functions of Pygame, such as setting up screen resolutions, loading images, manipulating game objects, etc. There is a "random AI player" implemented into the code that has no real written AI code other than to pick any available random square. Details of the "random AI player" are covered later. The copy module is used to create various copies, or instances of states of the tic-tac-toe game board. For readers who are new to coding with Python, copying objects, as in other languages like Java or C++, is done differently. In a language such as Java, for example, to copy an object would just be a case of writing, **obj1 = obj2**. In Python, this line just references **obj2** to **obj1**. To copy an object in Python involves using the copy module; `obj1 = copy.deepcopy(obj2)` is an example of how an object can be copied. This is shown in a code example later.

You may notice that from looking at the source code in **tictactoe.py**, there are a lot of global variables declared. This was done to prevent hard coded values being used. Some of the most import global variables for the tic-tac-toe game we're making are shown and described below.

```
HORIZ_RESOLUTION = 1024
VERT_RESOLUTION = 768
```

The above variables are used to define the window size/gameplay area. This can be adjusted to any compatible screen resolution.

```
screen = pygame.display.set_mode((HORIZ_
RESOLUTION, VERT_RESOLUTION))
```

For the X and O emblems to appear in the correct position on the screen, all emblem positions for each square on the grid have been preset.

QUICK TIP

By using variables in place of hard coded values, the value only needs to be changed once, and not each time the value is used within the script.

» OBJECT-ORIENTATED PROGRAMMING

Object-orientated programming has been used to quite an extent in this project. For those readers who are not yet familiar with object-orientated programming, it is an approach of writing software in terms of classes and objects. A class can be seen as a template of an object. A class can contain both variables and functions (usually referred to as methods) that define behaviour and operation.

For example, in **tictactoe.py**, there is a class implemented called **Player**. This class contains methods for what actions are player is able to perform during the game. In the case of the **Player** class implemented, two methods exist: **performMove** and **getAvailableMoves**. You may notice that there is a method called __init__, which is used to initialise variables with values if needed. This is also sometimes referred to as the class constructor, as in other languages such as C#, Java and C++.

There is also a class implemented called **AIPlayer**, which is an inherited class of the **Player** class. This means the class is able to access methods and variables of the **Player** class, which saves having to replicate the same code needed in that class (code reuse).

```
class AIPlayer(Player):
…
```

By taking this approach, the **AIPlayer** class only contains methods and variables relevant to what an AI Player can do and just call on the generic Player methods if needed to perform the same operation.

QUICK TIP

Two kinds of AI player have been implemented; one that uses the minimax algorithm and one that acts as a dumb AI player that just selects any available move at random. By default the minimax AI player is activated but to change to the dumb AI player, change the value of minimaxAI to False.

```
GRIDCELL_1X = 350
GRIDCELL_1Y = 250
GRIDCELL_2X = 500
GRIDCELL_2Y = 250
```

The above code, as you may be able to tell, relates to cell 1 and 2 of the grid, starting from the top left. See the diagram at the bottom of the page:

You may also notice from looking further into the source code that the range values are specified for each square on the grid. These are specified so that when the human player clicks on a square, the AI can detect which square the user has clicked on and selected to place their emblem.

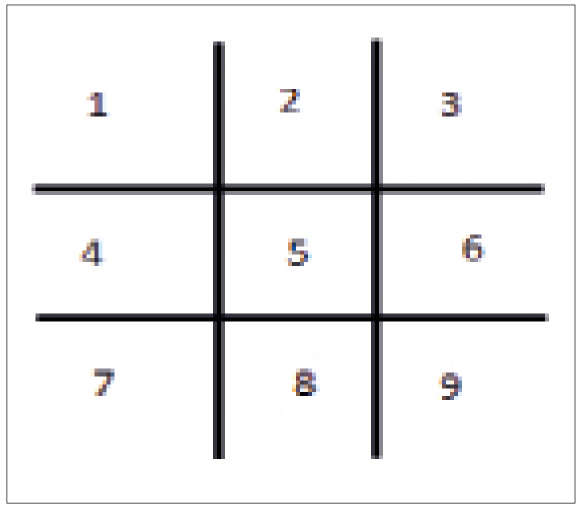
In the script, there is a class called **GridGameBoard**, which is the class used to set up the tic-tac-toe grid and is also used for game management purposes. For example, the class contains a method called **createGrid**, to create the grid for tic-tac-toe:

```
def createGrid(self):
 # Row 1 of 3 Cells
 self.theGrid.append(Cell(1, EMBLEM_BLANK,
GRIDCELL_1X, GRIDCELL_1Y))
 self.theGrid.append(Cell(2, EMBLEM_BLANK,
GRIDCELL_2X, GRIDCELL_2Y))
 self.theGrid.append(Cell(3, EMBLEM_BLANK,
GRIDCELL_3X, GRIDCELL_3Y))
 # Row 2 of 3 Cells
 self.theGrid.append(Cell(4, EMBLEM_BLANK,
GRIDCELL_4X, GRIDCELL_4Y))
 self.theGrid.append(Cell(5, EMBLEM_BLANK,
GRIDCELL_5X, GRIDCELL_5Y))
 self.theGrid.append(Cell(6, EMBLEM_BLANK,
GRIDCELL_6X, GRIDCELL_6Y))
 # Row 3 of 3 Cells
 self.theGrid.append(Cell(7, EMBLEM_BLANK,
GRIDCELL_7X, GRIDCELL_7Y))
 self.theGrid.append(Cell(8, EMBLEM_BLANK,
GRIDCELL_8X, GRIDCELL_8Y))
 self.theGrid.append(Cell(9, EMBLEM_BLANK,
GRIDCELL_9X, GRIDCELL_9Y))
```

As can be seen from the above code, a clear grid is created in the structure **theGrid**.

```
class GridGameBoard:
 # Stores a grid of cells for Tic Tac Toe
 theGrid = [ ]

...
```



The grid cells of tic-tac-toe have been numbered 1 to 9 from top left to bottom right.

Each cell on the grid has an ID number, x and y location of where the emblem is to be displayed and the value of that cell, which may be blank (**EMBLEM_BLANK**), taken by a human player (**EMBLEM_O**) or taken by the AI player (**EMBLEM_X**).

The **GridGameBoard** class also contains a method called **displayGrid**, which is used to display the current state of the game:

```
 def displayGrid(self):
  for gridcell in range(len(self.theGrid)):
   if self.theGrid[gridcell].getCellValue() ==
EMBLEM_X:
    screen.blit(xchar, (self.theGrid[gridcell].
getCellXPos(), self.theGrid[gridcell].getCellYPos()))
   if self.theGrid[gridcell].getCellValue() ==
EMBLEM_O:
    screen.blit(xchar, (self.theGrid[gridcell].
getCellXPos(), self.theGrid[gridcell].getCellYPos()))
```

As can be seen, a *for* loop is used to scroll through each square on the grid to see if it has been occupied by an X or a O. If it has, it is output on the grid.

AI player implementation

The AI player stores various information from the human player and game board so that it can decide on what move to take on the grid. Let's look at how the AI player started off being implemented.

```
class AIPlayer (Player):
gameBoardState = [ ] # Stores a collection of
gameboard states
enemyValidMoves = [ ] # Stores a collection of enemy
valid moves
firstValidMoves = [ ] # Collection of first valid moves
scoreCollection = [ ] # Score collection
firstMove = True # Identify that they are choosing for
the initial move
minimaxAI = True # Identifies to use the miniMax AI
algorithms
...
```

As can be seen from the above code, the AI player stores information on several aspects including various game board states, enemy valid moves, valid first moves for the AI player to take and score collection. As explained earlier in this article, the minimax algorithm is not initiated until after the human player's second move, so a variable (or property in OO terminology) is used, **firstMove** to identify that it is the AI Player's first move. After this has been done, it is then marked as **False**.

The top-level method overall that processes the minimax algorithm for the AI Player is called **getMiniMaxAIMove**. This method is used to return the selected move by the minimax AI player to a method of the AIPlayer class called **decideMove**, which in turn gets called from a top-level method **performAIMove**, which gets called from the main game processing loop.

```
...
moveResult = playerCollection[turn_indicator].
performAIMove()
...
```

Notice that both players are stored in a collection, **playerCollection**. The second player added to the collection is the AI player. The first player that is added to the collection is the human player.

Back to focusing on the minimax AI algorithm implementation itself by **getMiniMaxAIMove**, let's look at the implementation of the method as part of the **AIPlayer** class.

```
def getMiniMaxAIMove(self, validMovesIn):
indexCounter = 0 # Index counter for loop
gameBoardInstance = [ ] # Collection of gameboard
instances
chosenMove = -1 # No move selected by default
...
```

At the start of the method, a collection of valid moves is passed into the method for evaluation if there are any valid moves to process. A *while* loop is implemented to go through the collection of valid moves and is put into a game board instance:

```
def getMiniMaxAIMove(self, validMovesIn):
...
while indexCounter < len(validMovesIn):
 # Get a gameboard copy
 gboardInstance = copy.deepcopy(GridGameBoard.
theGrid)
 # Implement a valid move onto the game board
instance
 cellIndex = validMovesIn[indexCounter]
 # Mark AI Player Emblem In
 gboardInstance[cellIndex].setCellValue(self.
playerEmblem)
 # Score gameboard instance
 instanceScore = self.scoreInstance(gboardInstance,
cellIndex)
 # Add to score collection
 self.scoreCollection.append([instanceScore,
indexCounter, cellIndex])
...
# Add current instance of GridGameBoard.theGrid
gameBoardInstance.append(gboardInstance)
# Increment index counter
indexCounter = indexCounter + 1
```

As you can see from the above code, **copy.deepcopy** is used to copy an object, as we covered earlier on in the article.

» WHY PYTHON?

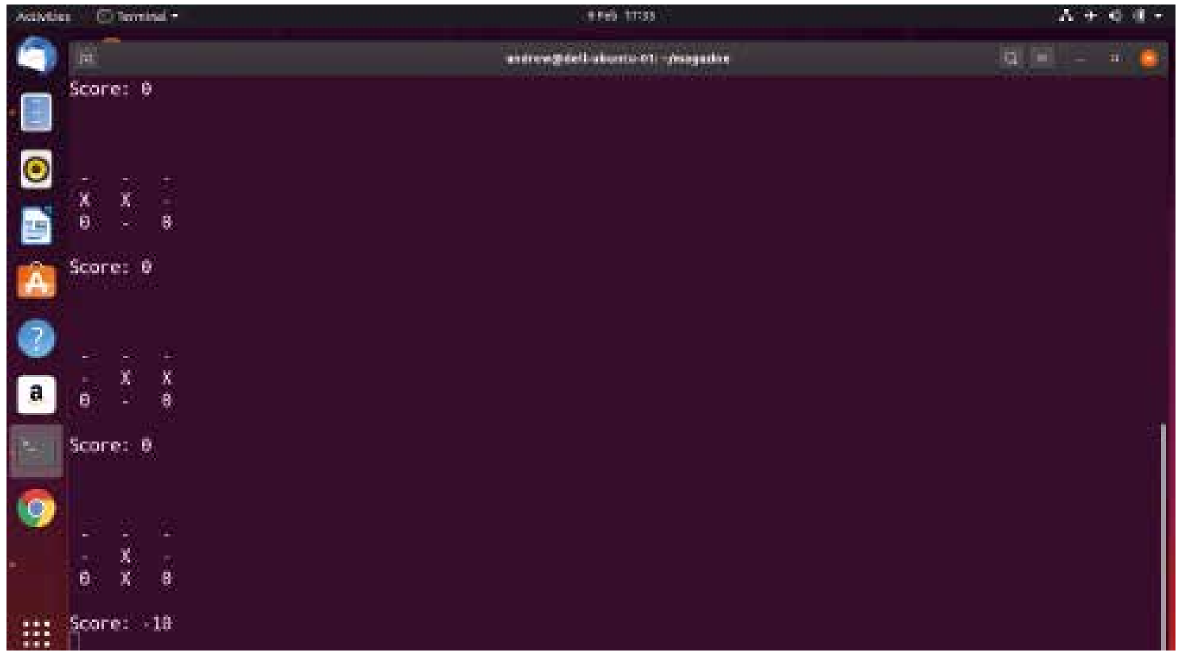
Python is an interpreted, high-level, general-purpose programming language that was first released in 1991 by its creator, Guido van Rossum. Very similar in programming construct to how BASIC (Beginners All-purpose Sybollic Instruction Code) was used to teach logic, program control and flow. However, In this day and age, Python is much more powerful than BASIC ever was by providing libraries to include functionality for areas such as mathematics and science, computer communications, video games and media (Pygame), file data manipulation/conversion and so much more.

Even though Python can be used as a learning tool to learn how to code, it can also be used in professional and commercial environments. In a lot of data-centred environments, Python can be used to retrieve data from files of different formats (*Excel*, *Access*, etc.) and then can be used to convert into another format entirely if there is a need to do so. As well as AI and data-modelling applications, Python can be used in correlation with 3D modelling, tool building, data security, and task automation, just to name a few. Python coding can be done on virtually any platform including Windows, Linux and Mac OS, and the code is portable between systems.

After the score collection of outcomes has been compiled, a move is chosen out of the score collection. This takes place after the *while* loop has processed:

```
...
# Choose the move based on scored game board
outcomes
chosenMove = self.selectFromCollection(self.
scoreCollection)
# Reset the score collection
self.scoreCollection = [ ]
return chosenMove
```

With this, the **AIPlayer** class will only contain methods and variables relevant to what an AI player can do and just call on the generic **Player** methods if needed to perform the same operation as a standard player.



Each processed outcome of the minimax process is given a score: 0 if there is a no win or no advantage outcome, -1 to identify a loss situation or disadvantage, or a 1 if the AI Player can win by performing that move.