Introduction

Previously, in class, we had created a Tic-Tac-Toe game, a player that lets us take in user input, and a player controlled by the computer. However, can we do better? With developments in artificial intelligence, computers have been able to defeat the best human players in complex games, such as chess, Go, and poker. In this project, we'll learn about an AI algorithm that we will implement to create a unbeatable AI Tic-Tac-Toe player.

Current Setup

At this point, we should already have an implementation of a Tic-Tac-Toe game, comprised of a `TicTacToeGame` class, a `Player` class, a `RandomPlayer` class and a `HumanPlayer` class. We will expand on our existing implementation in order to create a `AIPlayer`.

There is no starter code for this project. Instead, we will expand on the **tictactoe.py** file we have already created in class.

Let's Begin!

Game and Player Separation

Often, when we start creating more complex projects, we want to think about how to organize our files. Currently, we have all of our classes in one file, **tictactoe.py**, but as we add more to the player, it may be a better organizational choice to create additional files and partition our code. This is a good practice because it allows us to group similar functions together, while keeping our code length reasonable. Let's do this now.

> Create new **player.py** and **game.py** files. Move the `Player` class (and subclasses) into the **player.py** file, and move the `TicTacToeGame` class into the **game.py** file. In the **tictactoe.py** file, there should still be a `play` function for playing the game.

Try running the **game.py** file. What happens?

We notice that our game is broken! This is because we have not imported the code from the newly created files into the **game.py** file. Let's fix this using import statements. There are two ways to import something into a file.

Let's say we have a class called `Thing` in a file called **thing.py**. If we want to import this `Thing` into another file, I can use:

```
from thing import Thing

t = Thing()
```

This would successfully create an instance of `Thing` and assign it to `t`. Another way we can also import it is:

```
import thing

t = thing.Thing()
```

Here, we are doing the same thing as above, but just importing the entire file instead of the class in the file.

Stop and think: These are both two valid ways to import code from other files. What are the pros and cons of using each technique?

At the top of the **tictactoe.py** file, add the necessary imports so that the code recognizes objects in the **player.py** and **game.py** files. Run the code to verify that the imports are successful.

Creating the `AIComputerPlayer`

Now, let's start creating a class for `AIComputerPlayer`. We will fill it out later, after we learn about the AI algorithm, but we will set the basics first.

Stop and think: What are similarities and differences between the `AIComputerPlayer` and the existing players, `RandomComputerPlayer` and `HumanPlayer`?

Create another class called `AIComputerPlayer` in the **player.py** file and initialize it. Within the class, include the `get_move` method, but do not implement it yet. Let's just `pass` for now.

In order to implement the `get_move` method, we first need to learn how the AI algorithm works.

Recursion

The algorithm that we will be introducing uses a property known as *recursion*, which we need to understand in order to implement our AI bot. Therefore, we must take a quick detour and return to our Tic-Tac-Toe game later.

What is Recursion?

Recursion is when a function calls itself in its own implementation. This allows us to break down a problem into smaller subproblems that repeat. Sometimes, our problem might be too complex, too challenging, or too messy for `iterative` code. That is, code that uses loops (what we have learned so far). Therefore, we use recursion.

A simple example of recursion might be the factorial function.

```
def factorial(n):
    # assume n is a positive integer
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Here, we see that `factorial` actually calls itself from within the function! Rather than looping through to `n`, recursion lets us break down the factorial problem into `n*(n-1)!`, which then becomes `n*(n-1)*(n-2)!` and so on, until we hit `n*(n-1)*(n-2)*...* 1!`. Here, the `n==1` case is known as the *base case*. **Recursive functions will continue to recurse until it reaches a base case.**

Stop and think: How is recursion and iteration similar and different? What are the pros and cons of each?

Calculating Fibonacci

Recursion is a difficult concept to grasp. Just so we get some practice with recursion, we'll go through a small example completely unrelated to Tic-Tac-Toe. We will practice using Fibonacci numbers!

Create a file called **fibonacci.py**. Define a function called `fibonacci`, which takes a single input, representing the position in the sequence.

We want our function to calculate the Fibonacci value at that position. Suppose our Fibonacci sequence is defined as:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

The output of `fibonacci(0)` should be `0`. The output of `fibonacci(2)` should be `1`. The output of `fibonacci(7)` should be `13`.

Test out your code using a couple of different inputs and use Google to verify that your algorithm is giving the correct answer (be aware that there may be different definitions of "position"). *Do not use really large numbers to test the algorithm. It may be slow and you may get an error that your stack depth has been exceeded.*

AI Algorithm

Introducing Minimax

Minimax is a common algorithm used for *complete-information zero-sum two-player games*. This means a game with two players, where both players can see all the information, there is no "randomness" or "hidden information", and the gain of one player means the equal magnitude loss of another player. Minimax determines the optimal move for the hero given that the opponent is also playing optimally.

In Minimax, we assign values to various states of game, depending on how beneficial they are to helping you win. One basic example might be, if there is a state of the game where you win, that might get assigned to a value of +1, but the state where the opponent wins (and you lose) might get assigned to a value of -1. The goal of Minimax is to *maximize* the *minimum* gain. In other words, we want to **maximize the value of the worst-case scenario**.

Stop and think: What might be a reasonable value to assign to a tie game (assuming we assign +1 to a game where you win and -1 to a game where the opponent wins)?

Game Tree

A *game tree* is a powerful tool in game theory. The game tree is a tree representing all possible game states within a game. There are *nodes*, also known as *vertices*, representing points where players can take some action, connected by *edges*, representing the action taken at some node. An example of a game tree is shown below[1].

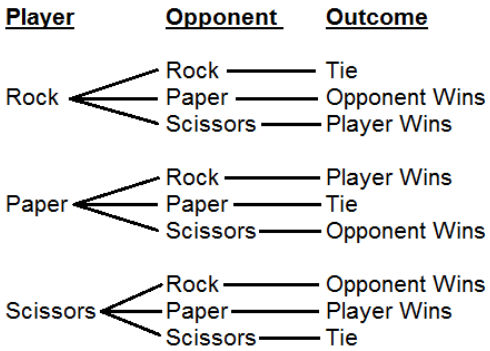
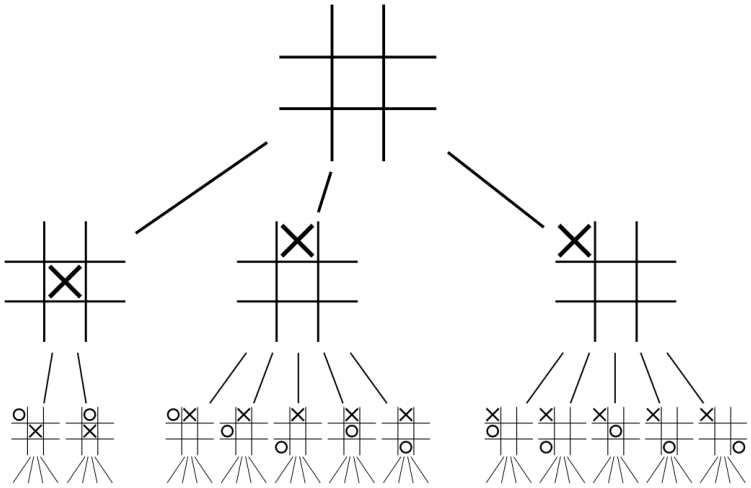


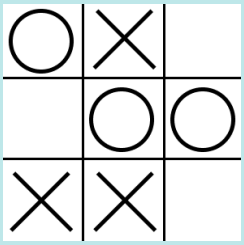
Figure 1: Rock, Paper, Scissors game tree

[1]Motta, Renan & Leite, Saul & Fonseca Neto, Raul. (2016). A String Kernel Density Estimation Algorithm for Repeated Games.

We can also create the same game tree for Tic-Tac-Toe. The *root*, or starting, node would be an empty game board. There would be 9 edges from the root node, representing each of the 9 positions where the first move can be places. A condensed version of the game tree is shown below[2]:



Draw the game tree starting from the state shown below (in other words, use the state below as the root node), assuming that **X** started first:



Take a photo/screenshot of your game tree!

You may notice that there are nodes where no additional actions can be taken. This is known as a *terminal node*. The game ends at this point, whether it means one of the players won or it ended as a draw. When a node is close to one of these terminal nodes, it can be easy to see what the best move is. These will be important!

Stop and think: Why do we want to have a game tree? How does this help our AI decide what action to take?

Assigning Values

In order for our game tree to be useful in decision making, we need to be able to make the **best** decision at any given decision node. Suppose the value of a winning node has value +1, a losing node has value -1, and a draw has value 0.

Look at the game tree that you drew in the previous section. Use the values `-1, 0, 1` to value the terminal nodes. Label these appropriately on your game tree. Take another picture/screenshot!

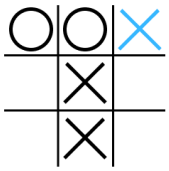
Now that we have the values of our terminal nodes, we can actually propagate these values up the tree and assign values to the intermediate nodes of the tree. In the next section, we will see how to do this.

[2] https://commons.wikimedia.org/wiki/File:Tic-tac-toe-game-tree.svg

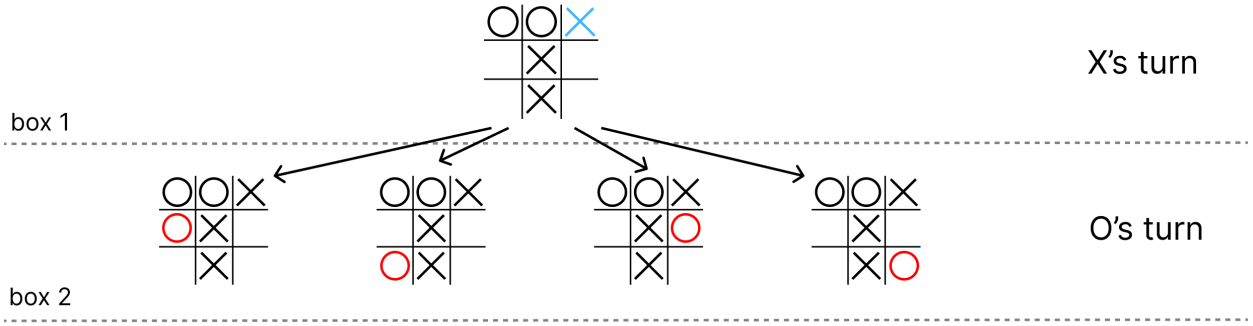
Stop and think: If we want to **maximize** the **minimum** gain, how do you think we should propagate the values? Should we use an average? Sum? Minimum? Maximum?

The game tree can become really large, so in order to gain some intuition about what we're trying to do, let's take a look at a larger section of a tic-tac-toe game tree.

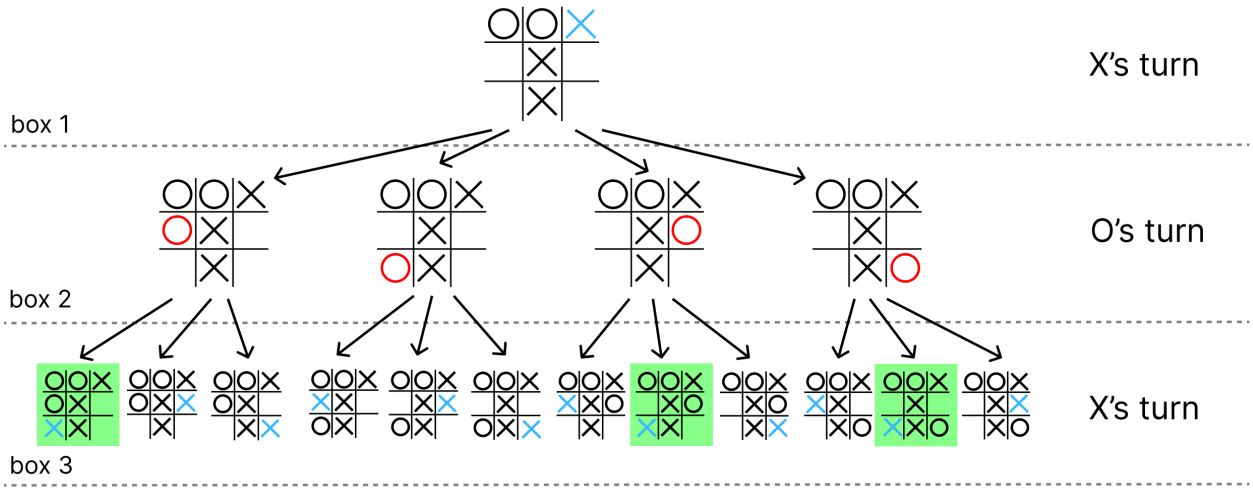
We'll start from this state. Suppose we are X and we just place our token:



After our turn, we alternate players, so it's O's turn:

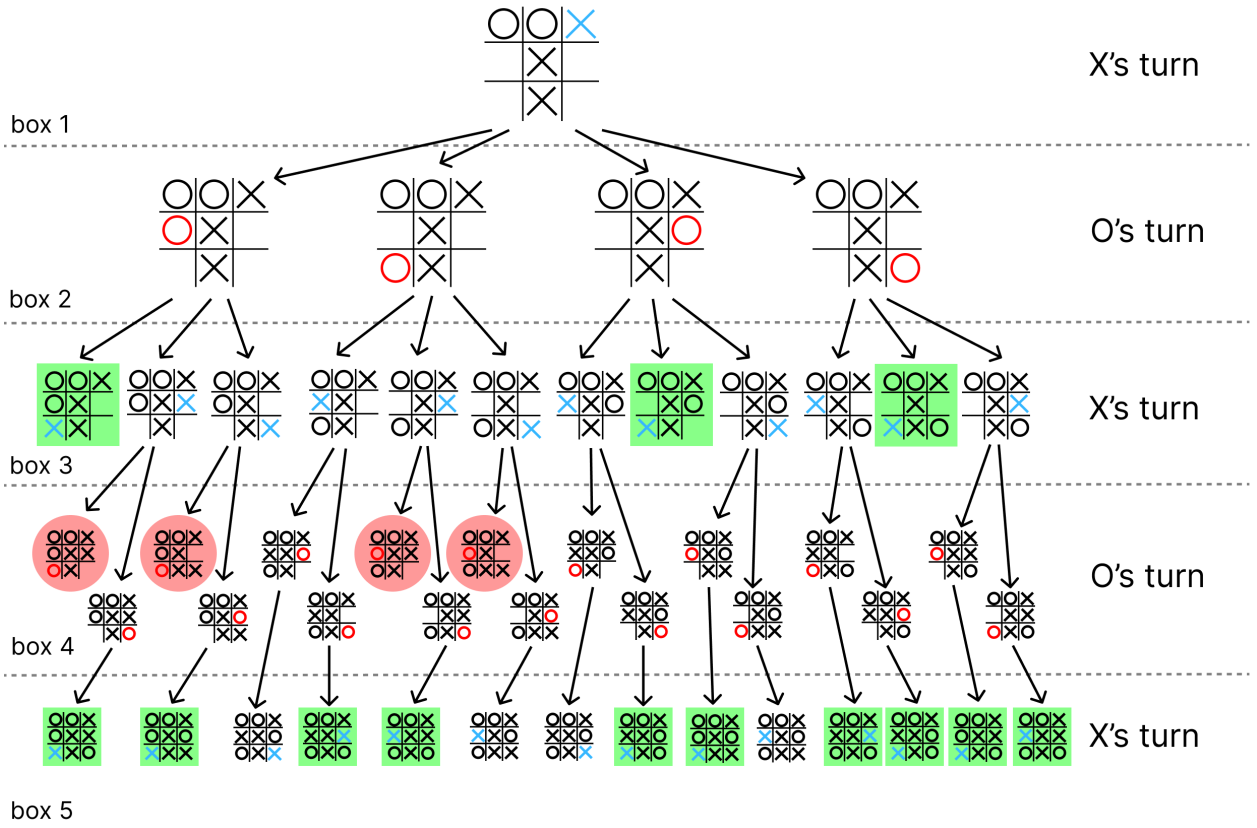


See how our game tree allows us to "see" all the different options in the game? Let's go again for X. At this point, there are some states where we have actually won, which are marked below in green boxes.



Now, if we finish the game tree, it would look something like this figure below. In box 4 (O's turn), our opponent (O) wins in certain parts of the game tree, shown in red circles.

On the game tree above, label the terminal winning states at +1, the terminal losing states as -1, and the terminal draw states at 0. *Remember that a terminal state means that the game has ended and there are no further moves. This should mean there are no exiting arrows from that state!*



Now, let's propagate the values up the tree. In box 4 (O's turn), you should already have some values for when O wins. However, there are many states with a single arrow (denoting a single action) leading to a next state in box 5.

Because there is only one single path from these states in box 4, there is a definite trajectory the game will take! X only has one option of where to move. Therefore, these states should be worth the same value as the following game state.

> On the same game tree above, give values to the game states in box 4 (O's turn) that do not have values yet.

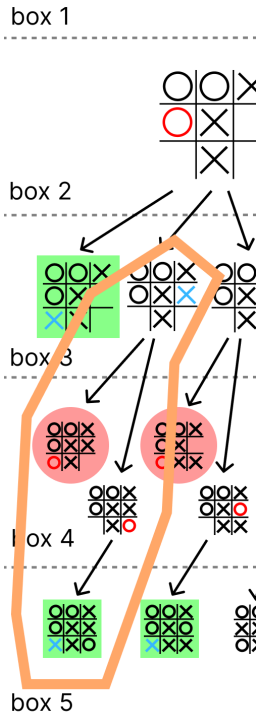
Now, let's think about box 3 (X's turn). This is less straightforward than box 4, because some states have 2 possibilities. In box 3's second state from the left (also shown on the next page), we even see that if O moves to the bottom left corner, O wins, but if O moves to the bottom right corner, X wins.

> *Stop and think*: If you are playing O, where would you want to go?

Each player wants to win, right? Each player wants to make the move that will put them in the best position. That means player O would want to play moves that get them closer to -1, and player X would want to play moves that get us closer to +1. In minimax, **we assume that the opponent plays optimally**. In other words, the opponent always takes the action that will lead to the most negative state. Because of this, we can basically assume that player O will play the bottom left corner in the example below.

> *Stop and think*: If we know O is going to go bottom left corner in the example above, then what should the value of the state in box 3 (the state at the top of the orange blob) be?

If you really stopped and thought about the above answer, you might realize that the state in box 3 (within the orange blob) is a dead end! If we play X there, our opponent has an easy way to make 3-in-a-row and win.



That means the value of this state should be assigned -1.

> Using this logic, let's assign values to the states in box 3 in the figure on page 6. These values should be the **minimum** of its following game states in box 4.

Now, we can move on to box 2. In order to get from box 2 to box 3, we want to make the best move possible, which means we are going to want to make the play that **maximizes** the value of our game state. Therefore, the value of the game state should be the **maximum** of its following game states (in box 3).

> Assign values to the states in box 2.

Ok, we should have all the game states labeled with values in our tree. *At this point, if you have any hesitation about your game tree values, please ask Kylie to check your work.*

Making Decisions

Now that we have all of our values, we can use these to guide our decisions. Remember that O wants to minimize the values and X wants to maximize the values.

> To go from box 1 to box 2, O must make a move. Select a path to take for O based on the values (O wants to minimize!). Draw something on the tree to indicate this choice.
>
> Now to go from box 2 to box 3, X makes a move. Select a path to take for X based on the values (X wants to maximize!) Draw something on the tree to indicate this choice.

If you have done this properly, you will see that we are not finished yet.

Then, to go from box 3 to box 4, O must make a move. Select a path to take for O based on the values (O wants to minimize!). Draw something on the tree to indicate this choice.

Finally, to go from box 4 to box 5, there should only be one path. Draw something on the tree to indicate this path.

If you have done this correctly, you'll see that this ends in a draw. The reason is because we are assuming that O is **playing optimally**, or in other words, constantly making the best possible move. However, against an opponent who is not as good, this opponent will likely make mistakes and play some sub-optimal moves. That means we may be able to catch our opponent at a game state that actually favors us, and win!!!

Stop and think: Why does our bot want to assume the opponent is playing optimally?

Implementation

Hopefully, at this point, you've gained some intuition about how the algorithm works. If not, please check in with a staff member (probably Kylie, since she created this project doc) to make sure you are on the right track. We are now going to start turning this algorithm into code. We can go back to the `get_move` method in our `AIPlayer` class.

Create a `minimax` method under the `AIPlayer` and let it do nothing for now (you should `pass`). The method should have 4 parameters: the board, the last move made, the current player, and whether the player should be minimizing or maximizing the score (in other words, if the player is you or your opponent).

Implement the `get_move` method such that if it is the first move on the board, we select the middle position on the tic-tac-toe board. If you would like, you can also implement this so that we select a random position on the board. If it is not the first move, then the move should take the position determined by the `minimax` method.

Now the `minimax` method is where we will implement the recursive AI algorithm. As we learned previously, a recursive algorithm needs a *base case*. In the minimax overview above, we propagate the values from the terminal nodes of the game tree. Here, the terminal nodes will be our base cases.

Stop and think: Why are the terminal nodes the base cases of the recursion? What are the conditions for determining whether a game state is in a terminal state?

In a recursive function, we want to always check first for whether we have achieved a base case of the recursion. Let's do that below.

Under the `minimax` method, implement the base case conditionals. Return a dictionary in the format `{'position': last_move, 'score': score}`, where `last_move` is the last move (as passed into the method) and `score` is the appropriate score assigned to the terminal node.

If the base case has not been achieved, then we need to continue with the algorithm. Since we want to maximize each score, we want to set the initial score equal to $-\inf$.

Stop and think: Why do we use $-\inf$ as our initial score?

Initialize a variable `best` and set it equal to a dictionary: `{'position': None, 'score': -float(''inf'')}`.

Then, we are going to test all the possible moves. In order to do so, we need to iterate through all the possible moves on the board, make the move, find the simulated score after that move, then undo the move. We will be keeping track of the best score through this process.

> Iterate through all the possible moves on the board. For each move, you should make the move, then call the `minimax` method using the new state, and save the resulting score.
>
> Next, undo the move by setting the `current_winner` to `None` and erasing the letter from the game's board.
>
> If the resulting score from the move is better than the best score so far, replace the score/position placeholder variable with the result, which should be a score/position dictionary, from the `minimax` call.
>
> Finally, return the score/position dictionary that would maximize the score.

We are almost done. Now, we just have to slightly modify the `get_move` method to make sure that we are returning a position on the board (and not a dictionary).

> Edit the `get_move` method knowing that `minimax` method returns a dictionary of the position and score.

That's it! You should have implemented a minimax bot to play tic-tac-toe.

> Play against your AI algorithm a few times. What happens when you play? If the AI is unbeatable, what happens when you play the AI against an AI opponent?

Conclusion

Congratulations! You have created an Tic-Tac-Toe AI that is unbeatable.

You may notice that the AI is sometimes slow. This is because the game tree can get really large. In fact, there are somewhere along the lines of 9! (or 362,880) states that the computer might be trying to evaluate!

There are ways to speed up the algorithm that we will not cover in the problem set. However, if you're interested, you can continue looking into this algorithm. There is a handy technique known as *alpha-beta pruning*, which basically "discards" branches in the tree as we explore the tree. This way, we can decrease the number of nodes that we need to evaluate, which speeds up the operation.