Tic-Tac-Toe with a Twist

Boye Akinwande, Andy Bergman, Lizzie Cross, Veronica Tan

Professor Stephen Kennedy

Carleton College Mathematics Department

June 1, 2011

1 The Game

Say there are two players, each with ten dollars. Both players simultaneously write down a bid of the amount they are willing to pay to make a move. Whoever bids more gets to take a spot on a standard tic-tac-toe board, but has to give the other player his bid. In case of equal bids, the player who has the tiebreaker gets to move on the board, gives up her bid and passes the tiebreaker to the other player.

Definition: *The tiebreaker* is the ability to win a tie in the case that both players bid the same amount to move. When a tie is made, *the tiebreaker* switches from one player to the other, along with the value of their shared bid.

From now on, we will refer to the player who begins with the tiebreaker as Xena. Her opponent is called Oscar. We want to know for which games a profitable strategy exists. The only difference between the players at the beginning is that one player begins with the tiebreaker. Perhaps starting with the tiebreaker will sway the outcome of the game.

Research Question: Does this tiebreaker guarantee a win for Xena?

Definition: n denotes the amount of money that each player starts with at the beginning of the game, and the resulting possibilities of gameplay. Each of the two players begins with the same amount of money n. An *$n game* is the game in which each player starts with n dollars.

The total amount of money in any game is $2n$. The game can end in a win, where one player gets three moves in a row, or a tie, where there are no longer any possible ways that either player can get three moves in a row.

1.1 Payoffs

If the game ends in a win, then the player who placed three moves in a row acquires all of the money, $2n$. If the game ends in a tie, then each player gets the amount that he/she ended with after the final move, which could be any integer between 0 and $2n$. We set out to find for which $\$n$ games a profitable strategy exists for Xena, who begins with the tiebreaker.

Profitable outcomes could include:

- A **win:** where Xena ends with three moves in a row and gains n.
- A **tie:** where once a tie is reached, Xena has at least $\$(n+1)$, meaning she gained at least $1.

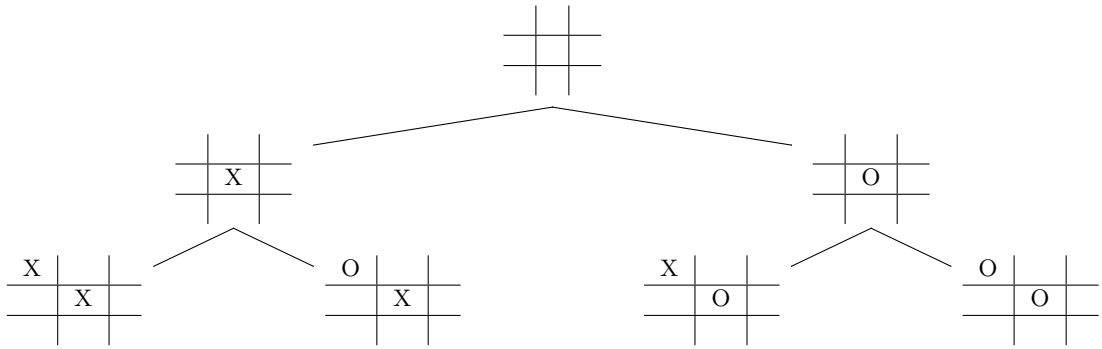
Unfortunately, defining outcomes in these terms puts the reader in the perspective of one of the players. To define an outcome as a loss, we imply that we are speaking from the perspective of the losing player. From now on, the players will each define their goals at the beginning of the game, for the sake of bidding and board play. There are a few different goals that each player can have.

Definition: A *tree of games* consists of

- nodes, representing stages of a tic-tac-toe board
- branches, representing when players put an additional move on a board.

Definition: A *leaf* is a terminal node on a tree. In our case, a leaf represents a finished game, where either a player has won or a tie has occured.

Below is an example of a tree. It shows the first two moves that can be chosen.



As we can see, the top of the tree is an empty board. This is always true. The initial node for our game trees will always be the empty board.

Below the empty board is a second level. Each node that is not a leaf on the tree will always have two daughters, just like this one. The two daughters correspond to the choice that Xena would make (on the left) and the choice that Oscar would make (on the right.) Soon, you will see how we narrowed down all of the choices to only the best choices for each player and his/her respective goal.

Below the level that signifies the first move is the level that signifies the second move. Each of the first move nodes, Xena's and Oscar's, has two daughters that signify a move for Xena and a move for Oscar. As we can see, these moves add a move to the previous board, in the level above.

2 The Computer Program

By performing any sort of combinatorial estimate how many ways there are to place moves on the board until the game is over, we see that these games are much too many to investigate by hand. Nonetheless, we must understand the most stategic way to place moves on the board, since all conjectures about the outcome of the game must involve this aspect of the game. We turn to the computer and create a Python3 program that

- **finds all the finished boards**
  and from these
- **finds the best finished boards**
  according to each player's goal.

We use object oriented programming, creating a class called "game". Instances of this class carry all the data specific to a certain way of placing moves on the board. In addition, there are functions that manipulate this data. *See Appendix 6.1 for the specifics of the program, i.e. code and comments.

2.1 Data

Each game object must track the board as moves are placed sequentially during gameplay. The board itself is a 3×3 matrix. Each position in the matrix can either be empty, filled by an X, or filled by an O. One of the most useful data types in the Python language is lists. They are mutable, meaning that you can change the data without creating an entire new copy of the data and destroying the old. A simple way to represent a matrix as a python data type is using a list of lists, where each item in the outermost list

corresponds to a row in the original matrix. The example below shows this isomorphism from the matrix to the list of lists. Note that each empty position in the matrix is represented by the symbol Ø.

$$\begin{array}{c|c|c} \mathrm{O} & \mathrm{O} & \\ \hline & \mathrm{X} & \\ \hline & \mathrm{X} & \mathrm{X} \end{array} \cong \Big[[\mathrm{O}, \mathrm{O}, \emptyset], [\emptyset, \mathrm{X}, \emptyset], [\emptyset, \mathrm{X}, \mathrm{X}]\Big]$$

It is important to note that the current state of any board does not indicate the order in which the players placed the moves on that board. So the second piece of data that will now be introduced is a sequence which represents the order of the moves. We will define a **move** as the player who is placing the move and the position the player chooses. This definition results from the fact that simply a sequence of Xs and Os (e.g. XOXX) does not uniquely define a way to play the game board. This is because two games (Game A and game B below) can have the same sequence of moves XOXX yet the locations chosen on the board or the order in which the locations are chosen can be different. The three distinct games below illustrate this reasoning. The subscripts denote the round in which the move was placed (e.g. X_3 would be the third move placed on the board).

Game A	$\neq$	**Game B**	$\neq$	**Game C**
XOXX	$=$	**XOXX**	$\neq$	**OXXX**
$\begin{array}{c\|c\|c} \mathrm{X}_1 & \mathrm{O}_2 & \\ \hline & \mathrm{X}_3 & \\ \hline & & \mathrm{X}_4 \end{array}$	$\neq$	$\begin{array}{c\|c\|c} \mathrm{X}_3 & \mathrm{O}_2 & \\ \hline & \mathrm{X}_1 & \\ \hline & & \mathrm{X}_4 \end{array}$	$=$	$\begin{array}{c\|c\|c} \mathrm{X}_3 & \mathrm{O}_2 & \\ \hline & \mathrm{X}_1 & \\ \hline & & \mathrm{X}_4 \end{array}$

Definition: A *move* is {a player + a position}. A *game* is defined by a sequence of *moves*.

Lastly, each game object will include a set of indicators, each of which corresponds to a certain outcome of the game. Is the game over? Was it a tie? If not, which player won? Each of these questions is answered by a variable being carried by the data in the game object.

2.2 Functions

The "game" class includes some trivial functions, like advancing the round after a move is placed. Here we will discuss the more important functions. When the program is given the game in some unfinished state, it finds all the next possible moves for each player. The simplest thing to do would be to let all empty spaces on the board denote a possible location for either player's next move. Howevever, doing this would slow computing time, and perhaps even fill all the available memory. Each of the functions below reduces the number of possible choices for each player's next move. The first and most widely used is the the function which checks for symmetry.

Goal: Given a game in some unfinished state, reduce number of possible next moves.

- Check for **symmetry**

  There are four lines of possible symmetry on the board, the vertical, the horizontal, and the two diagonals. When the state of the board is symmetrical across one of these lines, reflecting the board across that line results in an identical board. The four lines of symmetry are colored in gray on the boards below.

V **H** **D1** **D2**

By restricting the number of unique next moves by these lines of symmetry, we will significantly (multiplicatively) reduce the number of games that the computer will need to follow.

- Check for **winning move/block**

  For any rational player who wants to win the game, it is true that if a player has a winning move on the board (i.e. 2 moves in a row, the third space empty), the player should go in that location as his/her next move and subsequently win the game. However, if the player has some other goal, such as to tie, the other possible locations on the board in addition to the winning move are viable. This is because the player might need more of the money to win the game immediately than to go in a different location and eventually either win or tie. Since we cannot rule this out, we conclude that when a player has a winning move, no reductions (in addition to the reduction by symmetry) can be made on the number of possible next moves.

  However, for the player who does not have the winning move, reductions can be made. Let a block be defined as the opponent's response to a player's winning move. If a player has a winning move, and the opponent does not block the winning move in the next round, then in the next round, the player still has that winning move. If the opponent continues to not block, then the first time that the player is able to outbid the opponent and take a move, the game will be over. More generally, if the opponent does not block, the opponent must have enough money to lay down enough consecutive moves in a row in order to end the game in a win. The opponent can only do so when he/she still has a lane on the board to have three moves in a row. Let an **open lane** for the opponent be a row, column, or diagonal that meets this condition.

  Thus, if the player has a winning move, the opponent can either block the winning move or go consecutively in the open lane. Notice, however, that these two options may not be mutually exclusive.

  Game A below shows the case when Oscar has no open lane, and hence, he has only one next possible move: the block in the top right corner.

  Game B below shows the case when Oscar has an open lane (rightmost column), but the position of the block (top right corner) is in the open lane, so Oscar has only one viable next move: top right corner.

  Game C below shows the case when Oscar has an open lane (middle column), and the position of the block (left column, middle row) is *not* in the open lane, so Oscar has two viable next move: the block and the start of the open lane (note that Oscar doesn't have a third next move in the open lane because it would be redundant; Oscar will be going there next if he chooses to pursue the open lane).

Game A

O has no open lane

X	O	
O	X	X
X		O

1 next move for O

Game B

O has open lane

X	O	
O	X	
X		O

1 next move for O

Game C

O has open lane

X		O
	O	X
X		O

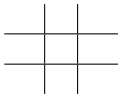
2 next moves for O

- Check for **game over**
  When a game is over, each player has zero possible next moves. Checking whether a game is over will be important when we use the program to find all possible ways to go from the empty board to finished boards.

2.3 Finding All Finished Games

We have now described the foundational data and functions that enable us to use the program to generate all possible ways to put moves on this board until the game is over, which is the same as finding all unique finished games because a game is defined by a sequence of moves. A basic outline of the algorithm used to do so is below.

Algorithm (Outline)

1. **Begin** with the empty board 

2. **Make all next moves** for each player

3. **Repeat step (2)** with these new boards.
   Stop at finished games

It is now necessary to translate this outline into more precise instructions for the computer to follow. To find all the finished games, we use a recursive function that puts all possible finished games into a list. A recursive function is a function that calls itself. This automatic looping process is preferred because it would be much more difficult to more explicitly direct the computer through the generation of all the finished games. Each game's index in the list of all finished games is referenced in the tree which is built by the recursion.

Below we rewrite the algorithm above in more technical terms. Also, you can find the corresponding python code for this algorithm in Appendix 6.1.

Algorithm (Technical)

1. Begin with an empty board.
2. **If** game over or board full
   then add game object to list and return game's index for tree.
3. **Else**
   (a) Find all viable next moves.
   (b) Create a **copy** of the game for each next move.
   (c) On each copy (subgame), **do** one of the moves.
   (d) **Run** the algorithm [2-3] on all the subgames.

The total number of finished games that the computer finds depends on the the functions that check for viable next moves. When using the functions defined earlier to decrease the number of next moves, the number of games found is

$$\mathbf{268,618}$$

This set of finished games is the one which we analyzed further. Other sets may be smaller but have forced another rule on the players when they are finding their possible next moves. One such rule would be if a player has a winning move, the opponent strictly blocks, thus ignoring the possibility of an open lane. We now reduce the set of all finished games to the set of the "best" finished games, which is built upon each player's system of values for the different outcomes of the game.

2.4 Finding the "Best" Finished Games

Once we had constructed an algorithm to generate all possible ways of placing moves on the board, we boiled these down to only the most feasible and strategic boards at each level of the tree. A major part of our algorithm for doing so involved p-prices.

Definition: The set of p-prices P for leaves of a tree of games are the proportions of total money that each player needs given a certain outcome at the leaf (win, loss, or tie).

For a player:

- after a win, proportion needed is 0, none of the money.
- after a loss, proportion needed is 1, all of the money.
- after a tie, proportion needed is either 0, 1, or $\frac{1}{2}$.

If we think of the p-price as the minimum cost for a player to achieve her objective, winning included, she should not have to expend any more money once she has achieved it. For this reason, we priced all wins at 0. Losses, at the opposite end of the spectrum, are priced at 1. We think of them as the least favorable outcomes, their existence being the highest cost to the players chance at success. We therefore require a player have all the money in the event of a loss. Ties can be priced at 0, 1, or $\frac{1}{2}$ depending on the objective. If a player is equally happy winning or tying, we price them at 0. If she wants to win and sees no distinction between tying and losing, we price them at 1. If she is content winning or tying with at least half of the total money, we price them at $\frac{1}{2}$.

Utilizing this methodology of assigning p-prices to all the finished boards, we were able to calculate p-prices for earlier stages of the game.

Definition: Function $f_p()$ finds the proportion of total money needed at a parent node from its two "best" daughter nodes.

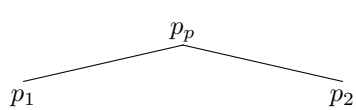
Theorem 1. *The proportion of total money needed at the parent node is the average of the proportions needed at the two daughter nodes.*

If p_1 and p_2 are the p-prices at the two daughter nodes, then

$$f_p(p_1, p_2) = \frac{p_1 + p_2}{2}$$

is the p-price at the parent node.

Proof. Let p_1 and p_2 and p_p be the proportions Xena needs of the total money at the two daughter nodes and the parent node respectively.



Let b be the proportion amount Xena bids at the parent level for the next move. When Xena goes, she gives away b. When Oscar goes, the minimum Xena gets is b. Thus,

$$2b = p_2 - p_1$$
$$b = \frac{p_2 - p_1}{2}$$
$$p_p = p_1 + b = p_1 + \frac{p_2 - p_1}{2} = \frac{p_1 + p_2}{2}$$

□

After laying out this necessary groundwork, we can outline the algorithm.

Algorithm for Xena (Outline)

1. **Begin** at leaves (bottom) of tree of all games.
2. **Assign value** from P to each leaf.

X wins

X	X	O
	X	
	X	O

$p = 0$

O wins

X	X	O
	X	O
		O

$p = 1$

Tie

X	X	O
O	X	X
X	O	O

$p = 1$

3. Of all Xena's choices that share a parent, **choose** the daughter node with smallest p as "best".

Daughters Where Xena Moves

Choice 1

X	O	O
	X	O
		X

$p_1 = 0$

$\leq$

Choice 2

X	O	O
	X	O
X		

$p_2 = .25$

$\leq$

Choice 3

X	O	O
X	X	O

$p_3 = .375$

$\leq$

Choice 4

X	O	O
	X	O
	X	

$p_4 = .4375$

Wins are conveniently priced at 0, so Xena would choose this option over all others. It is important to note that finished and unfinished boards can be daughters to the same parent as seen above.

We subsequently repeated the same for Oscar, examining all the boards from the same parent in which he had the last move and selecting that with the smallest p-price for him. In order to calculate the p-price at the parent node for Xena, we run the f_p function using the p-price of her optimal choice and her corresponding p-price at Oscar's optimal choice (not Oscar's p-price).

4. $p_{parent} = f_p(p_1, p_{\text{oscar's choice}})$
5. **Repeat steps (3-4)** with parent as new daughter. **Stop** at top of tree, the empty board.

   We have officially filtered out all unfeasible boards from the tree of all games and essentially shopped for the cheapest way for both players to achieve their objective given that they are fighting over the same fixed money supply.

Again, it is now necessary to translate this outline into more precise computer instructions. To find the best finished games, we use another recursive function. We pass into the function the tree of all finished games and it outputs the tree of "best" finished games according to each node's price for each player.

Below we rewrite the algorithm above in more technical terms. The corresponding python code for this algorithm is in Appendix 6.1.

Algorithm (Technical)

1. Begin at top level in tree of all games.
2. **If** node is a leaf, then add price from P to node.
3. **Else** run algorithm (2-5) on subnodes.
4. Of all sister nodes, **choose** a node for each player as "best" according to their prices. **Remove** all other sister nodes from tree.
5. **Add price** for each player to parent node of the two remaining daughter nodes. These prices are given by $f_p()$.

In practice, this game is played with a whole number of dollars, and each player must bid an integer amount of dollars for the right to make a move. Up to this point, we have been considering the amount of money each player needs and the amount of money each player bids as a proportions of the total money available. However, these proportions are not always integer amounts of the total money avaible $2n$. Thus, converting from p-prices to integers will involve rounding.

3 Converting to Integers

The integer equivalent of p-prices we call i-prices. i-prices differ from p-prices in two ways. First, i-prices are an amount of money rather than a proportion of the total amount of money. Secondly, i-prices involve rounding to ensure their integer status.

Definition: An *i-price* is the minimum integer amount needed to succeed at a certain stage of the game. Given that each player starts off with n dollars, for any particular player, the *i-price* is:

- 0 after a win;
- $2n + 1$ after a loss; and
- variable after a tie, depending on the player's initial objective. If the player was trying:
  - not to lose, the *i-price* is 0;
  - to tie, the *i-price* is n;
  - to tie favorably, the *i-price* is $n + 1$;
  - to win, the *i-price* is $2n + 1$.

An undesired outcome by an player is valued at $2n + 1$ at a terminal node. Because the total money supply in any game is $2n$, as the i-price $2n+1$ is averaged up the tree, the player will not have enough money to go down branches of a path which end up at a terminal node with an i-price $2n + 1$.

Definition: The function $f_i()$ finds the minimum integer amount of money needed at a parent node from its two "best" daughter nodes.

A set of prices I for terminal nodes of a tree of games is composed of three i-prices, one for each possible outcome at the terminal node.

For i-price games, we made two changes to our computer algorithm in section 2.4. We replaced function $f_p()$ with function $f_i()$ and we replaced set P with set I.

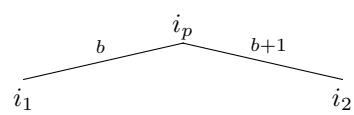
Theorem 2. *If i_1 and i_2 are the minimum integer amounts of money a player needs at the two daughter nodes, then*

$$f_i(i_1, i_2) = \begin{cases} \left\lfloor \dfrac{i_1 + i_2}{2} \right\rfloor & \textit{if player has tiebreaker} \\ \left\lceil \dfrac{i_1 + i_2}{2} \right\rceil & \textit{otherwise} \end{cases}$$

is i_p, the minimum integer amount of money the player needs at the corresponding parent node.

Proof. **Let:**

- i_p be the minimum integer amount Xena needs at some parent node;
- i_1 be the minimum integer amount Xena needs after she made the next move;
- i_2 be the minimum integer amount Xena needs after Oscar made the next move; and



- b be the integer amount Xena bids at the parent level for the next move.
  - Thus if Xena makes the move, she gives away b
  - But if Oscar makes the move instead, Xena gets at least $(b+1)$ when she has the tiebreaker, and b when she doesn't have the tiebreaker.

We therefore consider if Xena has the tiebreaker or if she does not have the tiebreaker.

Case 1: Xena has the tiebreaker

$$\begin{aligned} i_p = i_1 + b &= i_2 - (b+1) \\ b &= \frac{i_2 - i_1 - 1}{2} \\ i_p = \lceil i_1 + b \rceil &= \left\lceil i_1 + \frac{i_2 - i_1 - 1}{2} \right\rceil = \left\lceil \frac{i_1 + i_2 - 1}{2} \right\rceil = \left\lfloor \frac{i_1 + i_2}{2} \right\rfloor \end{aligned}$$

Case 2: Xena does not have the tiebreaker

$$\begin{aligned} i_p = i_1 + b &= i_2 - b \\ b &= \frac{i_2 - i_1}{2} \\ i_p = \lceil i_1 + b \rceil &= \left\lceil i_1 + \frac{i_2 - i_1}{2} \right\rceil = \left\lceil \frac{i_1 + i_2}{2} \right\rceil \end{aligned}$$

□

3.1 Ways to Play

In p and i-price games, players either aim to win, tie or not lose. In i-price games, players can also aim to tie favorably. We do not consider the strategy of tying favorably in the p-price games, because if the player's goal is to tie with more than $\frac{1}{2}$, it is not clear how much "more" would be considered a profit. In a p-price game, it is possible to tie with an amount infinitely close to $\frac{1}{2}$, so we only consider the T strategy defined below.

Definitions:

W: Player aims strictly to win.

TF: Player aims either to win or to tie the game with more than half the total amount of money.

T: Player aims either to win or to tie the game with at least half the total amount of money.

DL: Player aims either to win or to tie the game with any amount of money i.e. to not lose the game.

By the stars and bars combinatorial theorem, with 4 possible strategies and 2 participants in each game, this makes a total of

$$\binom{4+2-1}{2} = \binom{5}{2} = 10$$

possible player-opponent strategies.

3.2 Potential Outcomes

We thought about what the finished games in which Xena achieves her goal and/or Oscar achieves his goal look like. If Xena and Oscar were both aiming to win (W vs. W player-opponent strategy), only one of them will achieve his/his goal in any game, because only one player can win in any game. Thus, finished games in which both Xena and Oscar achieve their goal (if their goal is to win) is the empty set, and games in which either Xena or Oscar achieve their goal is all games that end in a win, which is also the complement of all games that end in a tie.

Following the same logic, in addition to the W vs. W player-opponent strategies, it is also not possible for both players to achieve their goals for the W vs. TF, TF vs. TF, W vs. T, TF vs. T, and W vs. DL player-opponent strategies.

Continuing our analysis on the W vs. TF, TF vs. TF and W vs. T player-opponent strategies, we look at finished games in which either Xena or Oscar achieves his/her goal. To do that, we have to first consider finished games in which neither Xena nor Oscar achieves his/her goal. The player-opponent strategy and respective situations in which neither Xena nor Oscar achieves his/her goal are the following:

- For W vs. TF, games that end in ties in which Xena ends up with more than n.
- For TF vs. TF, games that end in ties in which Xena ends up with n.
- For W vs. T, games that end in ties in which Xena ends up with more than n.

Thus for the W vs. TF, TF vs. TF, and W vs. T player-opponent strategies, the games in which either Xena or Oscar achieves his/her goal is the complement of all games of the conditions outlined for the respective player-opponent strategies.

For the TF vs. T and W vs. DL player-opponent strategies, the games in which either Xena or Oscar achieves his/her goal are the games in which either Xena achieves her goal or the games in which Oscar does.

And the rest of the player-opponent strategies and respective situations in which both Xena and Oscar achieve their goals are the following:

- For TF vs. DL, games that end in ties in which Xena ends up with more than n.
- For T vs. T, games that end in ties in which Xena ends up with n.
- For T vs. DL, games that end in ties where Xena ends up with more than or equal to n.
- For DL vs. DL, simply all games that end in ties.

It follows that in these four player-opponent strategies, the finished games in which either Xena or Oscar achieves his/her goal is all possible finished games.

Our theoretical predictions are summarized below in a table. It shows the finished games corresponding to Xena's and/or Oscar's goals. The intersection represents the finished games in which both Xena and Oscar achieve their goals, and the union represents the finished games in which either Xena or Oscar achieves his/her goal.

Potential Outcomes Table

		Finished Games by O's Goal							
		W		**TF**		**T**		**DL**	
Finished Games by X's Goal	**W**	$\emptyset$	$\{\text{all ties}\}^c$	$\emptyset$	$\{\text{all ties where X} \leftarrow\geq n\}^c$	$\emptyset$	$\{\text{all ties where X} \leftarrow> n\}^c$	$\boldsymbol{\emptyset}$	$\boldsymbol{G}$
	TF			$\emptyset$	$\{\text{all ties where X} \leftarrow= n\}^c$	$\boldsymbol{\emptyset}$	$\boldsymbol{G}$	$\{\text{all ties where X} \leftarrow> n\}$	G
	T					$\{\text{all ties where X} \leftarrow= n\}$	G	$\{\text{all ties where X} \leftarrow\geq n\}$	G
	DL							$\{\text{all ties}\}$	G
		$\cap$	$\cup$	$\cap$	$\cup$	$\cap$	$\cup$	$\cap$	$\cup$
		Intersection or Union							

G = set of all finished games

In summary,

- In the W vs. W, W vs. T, W vs. TF, and TF vs. TF strategies, it is possible for just one player to succeed or for both players to not succeed. This is due to the strict criteria of success for both players.
- Only in W vs. DL and TF vs. T strategies is the intersection empty and the union the set of all games; thus only in those two situations will one player necessarily succeed and the other necessarily fail in meeting their objectives. In the games where players are playing these strategies, it is guaranteed that exactly one player will succeed.
- In the T vs. T, T vs. DL, DL vs. DL and TF vs. DL strategies, it is possible for just one player to succeed or for both players to succeed. This is due to the low criteria of success for both players.

The 7 strategies we tested on our computer program were the W vs. DL, TF vs. T, W vs. W, W vs. T, T vs. T, T vs. DL and DL vs. DL player-opponent strategies. We tested the program with our theoretical predictions in mind, and our actual results match our theoretical predictions. The outcomes table shows the outcomes for n 1-30.

Outcomes Table

n	W vs DL	T vs T	W vs W	W vs T	T vs DL	DL vs DL	TF vs T
1	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X
3	O	B	F	O	B	B	O
4	X	X	X	X	X	X	X
5	X	X	X	X	X	B	X
6	O	B	F	O	B	B	O
7	O	X	X	F	B	B	X
8	O	X	F	F	B	B	X
9	O	X	X	F	B	B	X
10	O	B	F	O	B	B	O
11	O	X	X	F	B	B	X
12	O	X	X	O	B	B	X
13	O	X	X	F	B	B	X
14	O	X	X	F	B	B	X
15	O	X	F	F	B	B	X
16	O	X	X	F	B	B	X
17	O	X	X	F	B	B	X
18	O	B	F	O	B	B	O
19	O	X	X	O	B	B	X
20	O	X	X	F	B	B	X
21	O	X	F	O	B	B	X
22	O	B	X	O	B	B	O
23	O	X	X	F	B	B	X
24	O	X	X	F	B	B	X
25	O	X	X	F	B	B	X
26	O	X	X	F	B	B	X
27	O	X	X	F	B	B	X
28	O	X	X	O	B	B	X
29	O	X	X	F	B	B	X
30	O	B	F	O	B	B	O

Key

B	Both players achieve goal.
O	Player O achieves goal, player X does not
X	Player X achieves goal, player O does not
F	Neither player achieves goal

4 Theoretical Results

4.1 Do We Play The Same Way?

To begin to think theoretically about this game, we must reduce the variablity when the game is played with integer dollar amounts. If we can't control this variablity, a product of the rounding when going from p-prices to i-prices, we cannot generalize this game for certain n dollar amounts. So the question now becomes at what threshold do i-price trees become indistinguishable from p-price trees?

The difference between the p-price trees and the i-price trees is the potential rounding that could happen at any level of the tree for all the non-terminal nodes. It is convenient to keep the rounding term separate for each i-price in the tree. The function f_i can compute the averages of the non-rounded term and the rounding term seperately. This idea results in the following fact, which can be easily deduced from the definition of f_i when compared to the definition of f_p.

Fact: An i-price at a node has two components: a fraction p of $2n$, where p is the corresponding p-price, and r, where r is the rounding accumulated from below the node.

$$i = p \times 2n + r$$

Definition: Let $q = p \times 2$, the fraction of n needed in order to succeeed in real numbered games.

$$i = q \times n + r$$

We can now restrict p and q by using the p-price algorithm and the function f_p.

Lemma 1. *A completely reduced p-price p can have a maximum denominator of 1024, which implies that a completely reduced q can have a maximum denominator of 512. Smaller denominators can occur depending on the level of the tree in which p is found and the other p-prices in the tree.*

Proof. Each factor of two in the denomiator of a completely reduced p-price p for some node is the result of the averaging of the p-prices below the node, i.e. the composition $f_p\left(f_p(...), f_p(...)\right)$. There are at most 10 possible levels of the game. The bottom most level contains all finished games, or terminal nodes, which are priced from P rather than f_p and hence do not add a factor of 2 in the denominator of p with f_p. However, in the T strategy, $\frac{1}{2} \in P$, so the tenth level can also add a factor of 2 in the denominator of p. Thus, the maximum denominator for any p-price is $2^{10} = 1024$. It follows from the definition of q that the maximum denominator for the corresponding q of any p-price is $2^9 = 512$.

□

When we are given a p-price tree, the actual maximum denominator can be found by looking at each of the p-prices in the tree. The lemma sidesteps the issue that we don't have the time/computing power to find and check each p-price in the tree of all finished games, while we do have the time/computing power to find and check each p-price in a tree of "best" games.

Since i-prices differ from p-prices only in the additional r term, we proceed by finding how much effect this rounding term can possibly have.

Lemma 2. $|r| \leq 9$

Proof. At any node, $r_p = \frac{r_1+r_2}{2} + \delta$, where r_1 and r_2 are from the daughter nodes and δ is the additional rounding at the parent node.

Since $|\delta| \leq 1$, we know

$$|r_p| \leq \left|\frac{r_1 + r_2}{2}\right| + 1 \leq max[|r_1|, |r_2|] + 1$$

$\therefore$ $|r|$ increases at most by 1 every round.

□

With Lemmas 1 and 2, we can find a N dollar amount for which all $n > N$, according to some pattern players will choose the same tree as "best" when reducing the tree of all games. Another way of putting this is that players will make the same choice for their next move as they did in some n dollar game that we already understand. The result achieved is the following.

Theorem 3. *For all $n > 9216$, players will make the same board decisions mod 512.*

Proof. If $n > 9216$, at any parent node, we need ordering of i-prices for all next moves to be the same in the $\$(n + 512)$ game as in the $\$n$ game.

Let $i_1 \leq i_2 \leq \cdots$ be the known ordering of i-prices for Xena's next moves $\{1, 2, 3, \cdots\}$ in the $\$n$ game.

Let $\tilde{i_1}, \tilde{i_2}, \cdots$ be the i-prices for these moves in the $\$(n + 512)$ game.

Show: $\tilde{i_1} \leq \tilde{i_2} \leq \cdots$

We know:

$$i_1 \leq i_2 \tag{1}$$
$$q_1 \times n + r_1 \leq q_2 \times n + r_2 \tag{2}$$

Now,

$$\begin{aligned} \tilde{i_1} &= q_1(n + 512) + r_1 \\ &\leq q_2 n + q_1 512 + r_2 \\ \\ &\Downarrow \text{ if } q_1 \leq q_2 \\ \\ &\leq q_2(n + 512) + r_2 \\ &= \tilde{i_2} \end{aligned}$$

so if $q_1 \leq q_2$, then

$$\tilde{i_1} \leq \tilde{i_2}$$

How large must n be to force $q_1 \leq q_2$?
Suppose $q_1 > q_2$. What could n possibly be?

$$i_1 \leq i_2$$

$$q_1 \times n + r_1 \leq q_2 \times n + r_2$$

$$(q_1 - q_2)n \leq r_2 - r_1$$

$$min[q_1 - q_2]n \leq max[r_2 - r_1]$$

$$\Downarrow \text{ by Lemmas 1 and 2}$$

$$\frac{1}{512}n \leq 9 - (-9)$$

$$n \leq 9216$$

Therefore, if $n > 9216$, then $q_1 \leq q_2$, which implies that the best boards in the $\$(n + 512)$ game are the best boards in the $\$n$ game.

□

Now that we have a theorem about playing the board the same way for all n after a certain N, we can draw connections between p-price trees and i-price trees. For these n, the i-price tree consists of the same finished games as the p-price tree.

4.2 The Importance of the p-price Game and W vs. DL

After discussing both p-prices and i-prices we needed a way to reconcile the two with one another. How does a p-price dictate the corresponding proportion $\frac{i}{2n}$ that a player must have at a particular stage of the game? One assumption we could make is that $\frac{i}{2n}$ should exceed the p-price at every stage in the p-price tree.

- The W vs. DL game indicates that Xena needs $\frac{133}{256}$. (*See Appendix 6.2)
- Both players begin game with $\frac{n}{2n}$, yet in the \$1, \$2, \$4, and \$5 games, Xena wins.
- In these games, for every node after the empty board, $\frac{i}{2n} > p$.

Given this information, the assumption does not hold true although it seems that as long as $\frac{i}{2n}$ matches or exceeds the corresponding p-price at each finished board, Xena should be able to achieve her objective. To see how this is possible, we examined bidding in the p-price game as opposed to the i-price game. Because players can bid in real number amounts in the p-price tree, Oscar only needs to bid infinitesimally more than Xena to be able to move. In the worst case, Xena gives away the same amount when she moves as she receives when Oscar moves. For this reason, we do not assume b_x and b_o are any different.

In the i-price tree the player with the tiebreaker can buy the next move for the price of his or her bid, but the opponent is forced to buy for a whole dollar more than that bid. In the event that Xena does not possess the tiebreaker, she forfeits $b_x = b_o + 1$ in order to move. In the event that Xena does possess the tiebreaker when, she obtains $b_o + 1$ when Oscar moves.

Changes to Xena's money supply in p-price game as opposed to i-price game where M represents Xena's current money state and b represents the amount gained or given up in the p-price game.

p-price tree:

- When she moves $\implies M - b_x + (b_o - b_x) = M - b$.
- When Oscar moves $\implies M + b_o + (b_o - b_x) = M + b$.

i-price tree:

- When she moves **with tiebreaker:** $\implies M - b_x + (b_o - b_x) = M - b$.
- When she moves **without tiebreaker:** $\implies M - b_x + (b_o - b_x) = M - b - 1$.
- When Oscar moves **with tiebreaker:** $\implies M + b_o + (b_o - b_x) = M + b$.
- When Oscar moves **without tiebreaker:** $\implies M + b_o + (b_o - b_x) = M + b + 1$.

Because we assume b_x and b_o to be equivalent in the p-price tree, neither Xena nor Oscar has an advantage when bidding to move. This is reflected in the marginal difference between b_x and b_o in deciding who moves. The same holds true when the player with the tiebreaker moves. The fundamental difference between the p-price and i-price trees appears to lie in the second and fourth highlighted bullet points under **i-price tree**. Proportionally Xena gains $\frac{1}{2n}$ more than she would have in the p-price tree when Oscar moves without the tiebreaker. On the other hand, she loses $\frac{1}{2n}$ more than she would have in the p-price tree when she herself moves without the tiebreaker. Because Xena starts with the tiebreaker, the p-price tree indicates

that she has more opportunities with the tiebreaker. When quantified this means she obtains a surplus following the final summation of the gains and losses of $\frac{1}{2n}$ at every level in the tree. This pushes her true proportion of money possessed to over $\frac{133}{256}$ in the W vs. DL game.

To discern exactly how much of a surplus Xena has over Oscar, we initially kept track of the unfinished boards of the p-price tree where each player had the tiebreaker. At the stages in which Xena held the tiebreaker, we marked branches stemming to the left with $=$ and branches stemming to the right with $>$. These were meant to signify Oscar's bid in relation to Xena's bid: $=$ represents an equal bid and $>$ represents a greater bid. At stages in which Oscar possessed the tiebreaker, we marked branches stemming to the left with $<$ and branches stemming to the right with $=$, $<$ representing a smaller bid by Oscar in relation to Xena. $>$ and $<$ branches denote the gain of $\frac{1}{2n}$ and loss of $\frac{1}{2n}$ respectively that Xena experiences in the i-price tree as distinct from the p-price tree. With this information at hand, we formulated an equation that would ensure that by each finished board, Xena would have at least the amount of money specified by the p-price tree.

We track the accumulation of Xena's money at each level and node all the way down to the finished boards. We begin at the top of the tree where Xena has $\frac{n}{2n}$. At the next level

When Xena goes first she has

$$\frac{n-b}{2n} > \frac{23}{64}$$

When Oscar goes first Xena has

$$\frac{n+b+1}{2n} > \frac{87}{128}$$

When we sum these together we have

$$2\left(\frac{n}{2n}\right) + \frac{1}{2n} > 2\left(\frac{133}{256}\right)$$

The fractions to the right represent the corresponding p-prices at these levels of the tree, although the left-hand expressions need not exceed the p-prices at these stages of the game. We instead use the inequalities as an elementary template to be added together for the necessary p-price constraint that should be in place by the end of the game. As a corollary of Theorem 1, the sum of p-prices at the two daughter nodes will always equal twice the p-price at the parent node. This pattern persisted as we continued this process. The p-price bids b also cancel, so the only true excess is in the summation and subtraction of the $\frac{1}{2n}$ terms.

Resulting sum at third level

$$4\left(\frac{n}{2n}\right) + \frac{2}{2n} > 4\left(\frac{133}{256}\right)$$

Xena did not gain or lose any money at this level of the tree since there were the same number of $>$ and $=$ branches. The initial $\frac{1}{2n}$ has also doubled in size, but not in influence. It will have more of an effect on Xena's exhaustive money supply because of all the p-price daughter node summations and its presence in half the game's finished boards. Our final equation is as follows:

$$512\left(\frac{n}{2n}\right) + \frac{261}{2n} > 512\left(\frac{133}{256}\right)$$

This results in the following theorem describing Xena's outcome in the W vs. DL game.

Theorem 4. *Given that the "best" boards chosen are the same as the p-price tree suggests, for any $n \geq 14$ Xena will not be able to accomplish her objective to win.*

The same methodology can be applied to any of the other p-price games. Because the advantage from the Tiebreaker that Xena receives is finite, it is safe to say that in any of the games in which the p-price at the top of the tree indicates that Xena needs more than half the money to guarantee that she reaches her objective, she will not be able to win once n becomes sufficiently large. As far as explaining why some of the other numbers less than thirteen do not work, a lot could be attributed to the rigidity that comes with integer bids. This certainly was the case with the \$3 game.

4.3 TF vs. T

Analyzing the computer output for TF-T for the first 640 n, we find that the pattern of Xena's successes and failures to TF for certain n repeats mod 128 (*see Appendix 6.3). We also know that the maximum denominator for q in any of these games is 128. We find this by inspecting the different trees found for n mod 128. It makes intuitive sense that the pattern of successes and failures depends on this denominator, which limits the number of different ways rounding occurs in the i-prices. We can now state the following theorem.

Theorem 5. *Given that the best tree chosen is the same for n mod 128, for all $n > 1152$, the TF-T outcome will be the same mod 128.*

Proof. Recall:

$$i = q \times n + r \tag{3}$$

$$\tilde{i} = q(n + 128) + r \tag{4}$$

Show:

$$\text{If } i \sim n, \text{ then } \tilde{i} \sim n + 128 \qquad \text{where } \sim \text{ is some relationship } >, =, \text{ or } <.$$

We'll consider two cases: when $q = 1$ and when $q \neq 1$

Case 1: $q = 1$

Plug q into equations (3) and (4)

$$\begin{aligned} & i = n + r && \sim n \\ \implies \quad & \tilde{i} = n + 128 + r && \sim n + 128 \end{aligned}$$

Case 2: $q \neq 1$

Apply what we know about the denominator of q to the difference between q and 1.

$$\begin{aligned} |q - 1| &\geq \frac{1}{128} \\ |q - 1|n &\geq \frac{n}{128} \\ \text{set} \quad |q - 1|n &\geq \frac{n}{128} > |r| \\ n &> 128|r| \\ n &> 128 \times 9 \\ n &> 1152 \end{aligned}$$

$$\text{So if } n > 1152, \text{ then } |q-1|n > |r|,$$
$$\text{and if } |q-1|n \geq |r|, \text{ then } |q-1|(n+128) \geq |r|$$

Since we have limited the effect of r, we can now make the following statements:

$$\begin{aligned} i &\sim n \\ q \times n &\sim n \\ q \times n + r &\sim n \\ q(n+128) &\sim n+128 \\ q(n+128) + r &\sim n+128 \\ \tilde{i} &\sim n+128 \end{aligned}$$

$\therefore$ for all $n > 1152$ in a TF-T game on the same tree mod 128,

$$i \sim n \quad \Longleftrightarrow \quad \tilde{i} \sim n + 128$$

or the outcome is the same mod 128.

□

We can now put these theorems together to produce some meaningful corollaries.

4.4 Corollaries

From Theorem 3, we can draw a corollary to Theorem 4 which has fewer necessary conditions.

Corollary: Win-Don't Lose
For $n > 9216$, Xena playing W will fail to Oscar playing DL.

Also from Theorem 3, we can draw a corollary to Theorem 5.

Corollary: Tie Favorably-Tie
There are an infinite number of $\$n$ games where Xena can make a profit of $\$1$.

This accounts for about 80% of all positive integers. In the other 20%, she gains nothing, loses nothing.

5 Conclusion

After extensive analysis involving theoretical analysis as well as computer programming, we have an answer to our research question. Yes, a profitable strategy exists for most $\$n$ games. And we can make a conjecture of how large the profit is.

Final Conjecture: Given all strategies, the maximum amount of money that a player can guarantee to make is probably \$5.

Reasoning. Because we know that when Xena begins with the tiebreaker in the W vs. DL game, the largest n dollar amount for which she can guarantee a win is \$5. Otherwise, there are an infinite number of games for which there is a profitable strategy, but we assume that in the tie, she gains no more than \$1. This is based on our algorithm that she only needs to tie with a gain of \$1 in order to tie favorably.

Of course, this is only when we consider games where the bids must be integer amounts. If we specified a game in which players could bid pennies, a player could only win a maximum of \$.05. However, if players could only bid \$100 amounts in the game, then the player who begins with the tiebreaker could walk away with \$500.

For future research, the algorithm could be edited to redefine the Tie Favorably goal as one where she ties with at least $\$(n+6)$, and the researchers could run the algorithm for very high n-values to see if she ever achieves the new goal. Of course, her opponent's goal would have to be redefined as well so that he would be the most difficult opponent. His new goal would be specified as attempting to tie with at least $\$(n+5)$.

The new research could lead to examples where the player with the tiebreaker gains more than \$5, which would disprove our final conjecture. Nonetheless, we believe that a counter-example is unlikely to be found. Like in many competitive games, only if the player wins does the Tiebreaker, or marginal advantage, pay out big.

6 Appendix

6.1 Code for Computer Program

```
'''
Tic Tac Toe With a Twist
By Boye Akinwande, Andy Bergman, Lizzie Cross, Veronica Tan

This program can be used to study the version of tic tac toe where each player bids for the right
to move. It finds all the ways to play the game and from these finds the way each player would play
if the players are assumed to be playing strategically and according to some goal.

last modified: 6/1/11

Some aspects of recursion and pickling were aided by computer science department faculty and lab
assistants at Carleton College, Northfield MN.



if you want to use the interface built into the program, place this file in your working terminal
directory and use the following code:

> python3 tttTwist.py


if you want to access functions and data individually from terminal use the following code:

> python3
> from tttTwist import *
> main()     #example
> games    #prints all the games in "games"

'''


from copy import deepcopy
from math import ceil
from math import floor
import time
import pickle
import sys

class Game:
    def __init__(self, n = 10, simulation = False):
        self.simulation = simulation
        self.chips = [n, n]
        self.adv = 0
        self.advHistory = [0, None, None, None, None, None, None, None, None,None]
        self.board = [[None,None,None], [None,None,None], [None,None,None]]
        self.round = 0

        self.bids0 = [None, None, None, None, None, None, None, None, None]
        self.bids1 = [None, None, None, None, None, None, None, None, None]
        self.hasMove = 0
        self.history = []   #for sequence of moves
        self.placeHistory = []
```

```
        self.winningMove = [False, [0, 0], [[],[]] ]
        self.gameOver = False
        self.winner = None
        self.tie = False
        self.cantWin = [True,True]

        self.pHistory = [[None,None,None,None,None,None,None,None,None,None],[None,None,None,None,\
        None,None,None,None,None,None]]

        self.iHistory = [[None,None,None,None,None,None,None,None,None,None],[None,None,None,None,\
        None,None,None,None,None,None]]

        self.bidpHistory = [[None,None,None,None,None,None,None,None,None],[None,None,None,None,\
        None,None,None,None,None]]

        self.bidiHistory = [[None,None,None,None,None,None,None,None,None],[None,None,None,None,\
        None,None,None,None,None]]

        self.winning3 = []  #for all possible ways to get three moves in a line
        for i in [0,1,2]:
            self.winning3.append([self.board[i][0], self.board[i][1], self.board[i][2]])
        for i in [0,1,2]:
            self.winning3.append([self.board[0][i], self.board[1][i], self.board[2][i]])
        self.winning3.append([self.board[0][0], self.board[1][1], self.board[2][2]])
        self.winning3.append([self.board[0][2], self.board[1][1], self.board[2][0]])

        self.choices = []
        self.subgames = []


        self.hsymm = False
        self.vsymm = False
        self.d1symm = False
        self.d2symm = False


#_________________________________CLASS FUNCTIONS_______________________________________________

    def nextRound(self):
        self.round += 1

    def updateBids(self, a, b):
        self.bids0[self.round-1] = a
        self.bids1[self.round-1] = b


    def compareBids(self):
        del(self.hasMove)
        if self.bids0[self.round-1] > self.bids1[self.round-1]:
            self.hasMove = 0

        elif self.bids0[self.round-1] < self.bids1[self.round-1]:
            self.hasMove = 1

        else:
            self.hasMove = self.adv
            self.adv = (self.adv + 1) % 2
```

```
    if self.simulation == True:
        if self.hasMove == self.adv:
            self.adv = (self.adv + 1) % 2
            self.advHistory[self.round] = self.adv
        else:
            self.advHistory[self.round] = self.adv


def updateChips(self):
    hasNoMove = (self.hasMove + 1) % 2
    if self.hasMove == 0:
        bid = self.bids0[self.round-1]
    else:
        bid = self.bids1[self.round-1]

    self.chips[self.hasMove] = self.chips[self.hasMove] - bid
    self.chips[hasNoMove] = self.chips[hasNoMove] + bid


def updateBoard(self, y, x):
    self.board[y][x] = self.hasMove
    self.history.append(self.hasMove)
    self.placeHistory.append((y,x))


def updateWinning3(self):
    del(self.winning3)
    self.winning3 = []
    for i in [0,1,2]:
        self.winning3.append([self.board[i][0], self.board[i][1], self.board[i][2]])
    for i in [0,1,2]:
        self.winning3.append([self.board[0][i], self.board[1][i], self.board[2][i]])
    self.winning3.append([self.board[0][0], self.board[1][1], self.board[2][2]])
    self.winning3.append([self.board[0][2], self.board[1][1], self.board[2][0]])
    del(i)

# functions for checking symmetry and changings bools accordingly
def hsym(self):
    for row in [0,1,2]:
            if self.board[row][0] != self.board[row][2]:
                self.hsymm = False
                return
    self.hsymm = True
def vsym(self):
    for col in [0,1,2]:
            if self.board[0][col] != self.board[2][col]:
                self.vsymm = False
                return
    self.vsymm = True
def d1sym(self):
    for cord in [[2,0],[1,0],[2,1]]:
        if self.board[cord[0]][cord[1]] != self.board[cord[1]][cord[0]]:
            self.d1symm = False
            return
    self.d1symm = True
def d2sym(self):
```

```
    for cord in [[0,0],[1,0],[0,1]]:
        if self.board[cord[0]][cord[1]] != self.board[2-cord[1]][2-cord[0]]:
            self.d2symm = False
            return
    self.d2symm = True


def updateChoices(self):
    del(self.hsymm,self.vsymm,self.d1symm,self.d2symm)
    self.hsym()
    self.vsym()
    self.d1sym()
    self.d2sym()

    rows = [0,1,2]
    cols = [0,1,2]
    if self.hsymm:
        cols = [0,1]
    if self.vsymm:
        rows = [0,1]

    self.choices = []
    for row in rows:
        for col in cols:
            nosymm = True
            if self.d1symm:
                cord = [row,col]
                if cord == [2,0] or cord == [1,0] or cord == [2,1]:
                    nosymm = False
            if self.d2symm:
                cord = [row,col]
                if cord == [2,2] or cord == [2,1] or cord == [1,2]:
                    nosymm = False


            if self.board[row][col] == None and nosymm:
                self.choices.append((row,col,1,0))
                self.choices.append((row,col,0,1))

def checkWinAndChoices(self):

    self.winningMove = [False, [0, 0], [[],[]] ] #[,[# 0s wms, #1s wms], [[0cords], [1s cords]]
    checkTie = True
    openlane = [[],[]]  #a list for each player

    for i in range(len(self.winning3)):
        awin = self.winning3[i]
        Nones = awin.count(None)
        movesinarow = [awin.count(0), awin.count(1)]

        # check to see if this winning3 proves not yet a tie
        if checkTie == True:
            if movesinarow[0] == 0 or movesinarow[1] == 0:
                checkTie = False
        # for each lane (above), loop through both players
        for player in [0,1]:
```

```
        #if player has won in this winning 3, update variables and return
        if movesinarow[player] == 3:
            del(self.choices,self.winningMove,self.gameOver,self.winner)
            self.choices = []
            self.winningMove = [False, [0, 0], [[],[]] ]
            self.gameOver = True
            self.winner = player
            return

        # if player has a winning move in this winning 3
        elif movesinarow[player] == 2 and Nones == 1:
            #get board pos of the empty space None
            relativepos = awin.index(None)
            if i in [0,1,2]:
                pos = (i, relativepos)
            elif i in [3,4,5]:
                pos = (relativepos, i-3)
            elif i == 6:
                pos = (relativepos, relativepos)
            else:
                pos = (relativepos, 2-relativepos)
            self.winningMove[0] = True
            self.winningMove[1][player] += 1
            self.winningMove[2][player].append(pos)
            #self.blockHistory[self.round - 1] = True
            del(i,pos,relativepos)
        # if open lane
        elif (movesinarow[player] == 1 and Nones == 2) or Nones == 3:
            #get board pos of the empty space None
            relativepos = awin.index(None)
            if i in [0,1,2]:
                pos = (i, relativepos)
            elif i in [3,4,5]:
                pos = (relativepos, i-3)
            elif i == 6:
                pos = (relativepos, relativepos)
            else:
                pos = (relativepos, 2-relativepos)
            #could increase efficiency by choosing lane that includes block
            #adjust openlane according to what's in it.
            #means we haven't found a open lane for this player yet
            if not openlane[player]:
                openlane[player] = [int(i), pos, Nones]
            elif openlane[player][2] > Nones:
                openlane[player] = [int(i), pos, Nones]
    del(awin,Nones,movesinarow)

# if it's a tie
if checkTie == True:
    self.tie = True
    self.choices = []
    self.winningMove = [False, [0, 0], [[],[]] ]
    self.gameOver = True
    return


# if there's a winning move, go for a win (if you have an open lane
```

```
    # or a winning move), as well as block
    if self.winningMove[0] == True:
        del(self.choices)
        self.choices = []

        for player in [0,1]:
            oppon = (player-1)%2
            # if player has a winning move, go in it:
            if self.winningMove[1][player] > 0:
                poswin = self.winningMove[2][player][0]  #(row , col)
                self.choices.append((poswin[0],poswin[1],oppon,player))

            # if player is blocking, block, and go for win if possible
            else:
                for posblock in self.winningMove[2][oppon]:
                    self.choices.append((posblock[0],posblock[1], oppon, player))

                # if player blocking has an open lane,
                if openlane[player]:
                    #get all positions of Nones in openlane.
                    positionNones = []
                    i = openlane[player][0]
                    for relativepos in [0,1,2]:
                        if self.winning3[i][relativepos] == None:
                            if i in [0,1,2]:
                                pos = (i, relativepos)
                            elif i in [3,4,5]:
                                pos = (relativepos, i-3)
                            elif i == 6:
                                pos = (relativepos, relativepos)
                            else:
                                pos = (relativepos, 2-relativepos)
                            positionNones.append(pos)

                     # if all positions of Nones are different than the places where
                     # player already is blocking, add a new choice from the openlane
                    newChoice = True
                    for pos in positionNones:
                        if pos in self.winningMove[2][oppon]:
                            newChoice = False
                    if newChoice:
                        pos = positionNones[0]
                        self.choices.append((pos[0],pos[1],oppon,player))


def fillSubGames(self):
    templist=[]
    for k in range(len(self.choices)):
        templist.append(deepcopy(self))
    self.subgames=templist
    del(templist)

def clearSubGames(self):
    del(self.subgames)

def doChoices(self):
    for index in range(len(self.choices)):
```

```
        self.subgames[index].nextRound()
        self.subgames[index].updateBids(self.choices[index][2], self.choices[index][3])
        self.subgames[index].compareBids()
        self.subgames[index].updateBoard(self.choices[index][0], self.choices[index][1])


#functions for printing data to terminal
def getHistory(self):
    print()
    print("history of moves: ", self.history)
def getAdvHistory(self):
    print('advHistory ', self.advHistory)
def getPlaceHistory(self):
    print()
    print("history of cordinates chosen: ", self.placeHistory)
def getBids(self):
    print("player 0's bids: ", self.bids0)
    print("player 1's bids: ", self.bids1)
def getHasMove(self):
    print()
    print("player", self.hasMove, "gets to move.")
def getWinner(self):
    print()
    if self.winner == None:
        if self.tie:
            print('Game is a tie')
        else:
            print('Game is not complete')
    else:
        print('Winner is player', self.winner)
def getChoices(self):
    print("choices: ", self.choices)
def getBoard(self):
    print()
    for j in [0,1,2]:
        for i in [0,1,2]:
            print(' ', end = '')
            if self.board[j][i] != None:
                print(self.board[j][i], end = '')
            else:
                print(" ", end = '')
            print(' ', end = '')
            if i != 2:
                    print('|', end = '')
        print()
        if j != 2:
            print('-----------')
    print()
def getGame(self):
    print('**************************************************')
    self.getWinner()
    self.getHistory()
    self.getAdvHistory()
    self.getBoard()
    self.getPlaceHistory()
    print()
    print('**************************************************')
```

```
    # this function re-writes python's print() function for the class game.
    def __repr__(self):
        rv="\n\n"
        rv+=str(self.board[0])+"\n"
        rv+=str(self.board[1])+"\n"
        rv+=str(self.board[2])+"\n"
        return  rv

#-------------------------------------Class Functions end----------------------------------------


#-----------------------------------Functions on outermost level--------------------------------

def regularTTT(game):
    '''for playing regular tic tac toe in terminal'''
    game.bids1 = [0,0,0,0,0,0,0,0,0]
    game.bids0 = [0,0,0,0,0,0,0,0,0]

    while game.round < 9 and game.gameOver == False:
        game.getHistory()
        game.getBoard()
        game.compareBids()
        game.updateChips()
        game.getHasMove()
        y = int(input('input row: ')) - 1
        x = int(input('input column: ')) - 1
        game.updateBoard(y,x)
        game.updateWinning3()
        game.checkWinAndChoices()
        game.nextRound()
    game.getBoard()
    print()
    if game.winner != None:
        print("Winner is player ", game.winner, '!!!')
    else:
        print("It's a tie.")
    print()


#----------------------------Find All Games-----------------------------------------------------

def recursionGames(game):
    '''this function finds all possible ways to play tic tac toe and puts them in "games"
    returns a tree of indices, which correspond to unique games in "games". games found = 268618.'''
    global games

    game.updateChoices()
    game.updateWinning3()
    game.checkWinAndChoices()

    # base case
```

```
    if game.round == 9 or game.gameOver == True:
        games.append(game)
        l = len(games)
        if l%10000 == 0:
            print('finished game', l)
        return l-1    #or return game (if you want tree of class instances, not indices.)

    else:
        game.updateChips
        game.fillSubGames()
        game.doChoices()

        # call recusionGames() for each subgame
        node = []
        for subgame in game.subgames:
            node.append(recursionGames(subgame))

        del(subgame)
        game.clearSubGames()
        return node


#----------------------------------Find Best Games ---------------------------------------------
def findBest(games, tree,n,id,speed):
    '''this function finds best games in tree, which corresponds with the games list.
    returns best tree, btree'''

    #get some variables that we will use in the functions
    n2 = n*2
    defaultp = n**2 +10

    global gameplays
    endvalues = gameplays[id-1]
    if id == 7:
        extra = 1
    else:
        extra = 0


    def addP(needsP,r,n2=n2,endvalues=endvalues, extra=extra,speed=speed):
        '''this function adds p-prices at level r to games in needsP'''

        #don't add p-price when r < 0
        if r < 0:
            return

        #fraction tree
        if n2 == -2 or speed==0:
            #case for leaf (win, loss, or tie)
            if len(needsP) == 1:
                g = games[needsP[0]]
                if g.winner == 0:
                    g.pHistory[0][r] = endvalues[0][0]
                    g.pHistory[1][r] = endvalues[1][0]
                elif g.winner == 1:
                    g.pHistory[0][r] = endvalues[0][1]
                    g.pHistory[1][r] = endvalues[1][1]
```

```
        else:   #its a tie
            g.pHistory[0][r] = endvalues[0][2]
            g.pHistory[1][r] = endvalues[1][2]
    else:
        for a in [0,1]:
            o = (a+1)%2

            #average to get new p
            p0 = games[needsP[0]].pHistory[a][r+1]
            p1 = games[needsP[1]].pHistory[a][r+1]
            p = (p0 + p1)/2
            for k in needsP:
                games[k].pHistory[a][r] = p
                games[k].bidpHistory[a][r] = games[needsP[o]].pHistory[a][r+1]-p

# if integer tree
if n2 > 0:
    #case for leaf (win, loss, or tie)
    if len(needsP) == 1:
        g = games[needsP[0]]
        ivals = [None,None]
        g.cantWin = [True,True]
        if g.winner == 0:
            ivals[0] = int(endvalues[0][0]*n2)
            ivals[1] = int(endvalues[1][0]*n2)
        elif g.winner == 1:
            ivals[0] = int(endvalues[0][1]*n2)
            ivals[1] = int(endvalues[1][1]*n2)
        else:   #its a tie
            ivals[0] = int(endvalues[0][2]*n2) + extra
            ivals[1] = int(endvalues[1][2]*n2)

        for a in [0,1]:
            if ivals[a] == n2 and (extra == 0 or g.tie==False):
                ivals[a] += 1
            g.iHistory[a][r] = ivals[a]

    else:
        # for each player
        for a in [0,1]:
            o = (a+1)%2   # a's opponent

            #if a has adv
            if a == games[needsP[a]].advHistory[r]:
                adv = True
            else: #a doesn't have adv
                adv = False


            #average i, round up or down according to wheter player a has advantage

            i0 = games[needsP[0]].iHistory[a][r+1]
            i1 = games[needsP[1]].iHistory[a][r+1]


            if adv:
                i = floor((i0 + i1)/2)
```

```
                else:
                    i = ceil((i0 + i1)/2)

                for k in needsP:
                    games[k].iHistory[a][r] = i
                    games[k].bidiHistory[a][r] = i-games[needsP[a]].iHistory[a][r+1]

def recursionProp(node, round, defaultp=defaultp, n2=n2):
    '''this recursive function finds sister games, compares p-values, and removes games
    accordingly from a tree to create best tree. Calls function addP(). Edits "node" in
    place, returns nothing.'''

    for obj in node:
        #*************BASE CASE************************************
        if type(obj) == int:    #finished game
            # by cases, assign p-prices
            addP([obj],round)

        #**********RECURSIVE CALL************************************
        else:
            recursionProp(obj, round+1)   #recursive call


    #now all games are p-filled (and i-filled) to same level, so compare and choose
    # this will reduce # of choices.

    #if a fraction tree
    if n2 == -2:
        pmin0 = [defaultp,None,None, None, None]
        pmin1 = [defaultp,None,None, None, None]
        # remove all bad objects from node at round, and remove old and new max and mins
        for obj in list(node):
            if type(obj) == int:
                if games[obj].history[round-1] == 0:  #PLAYER 0'S MOVE
                    if games[obj].pHistory[0][round] > pmin0[0]:
                        node.remove(obj)

                    elif games[obj].pHistory[0][round] == pmin0[0] and games[obj].pHistory[1]\
                    [round] < pmin0[4]:
                        node.remove(obj)

                    elif games[obj].pHistory[0][round] == pmin0[0] and games[obj].pHistory[1]\
                    [round] == pmin0[4] and games[obj].round >= pmin0[3]:
                        node.remove(obj)

                    else:
                        # there's a new min for player 0, a new best choice
                        # if there's a previous object in pmin, now it must be removed
                        if pmin0[1] != None:
                            node.remove(pmin0[1])
                        pmin0[0] = games[obj].pHistory[0][round]
                        pmin0[1] = obj
                        pmin0[2] = obj
                        pmin0[3] = games[obj].round
                        pmin0[4] = games[obj].pHistory[1][round]
```

```
    else:                                   #PLAYER 1'S MOVE
        if games[obj].pHistory[1][round] > pmin1[0]:
            node.remove(obj)

        elif games[obj].pHistory[1][round] == pmin1[0] and games[obj].pHistory[0]\
        [round] < pmin1[4]:
            node.remove(obj)

        elif games[obj].pHistory[1][round] == pmin1[0] and games[obj].pHistory[0]\
        [round] == pmin1[4] and games[obj].round >= pmin1[3]:
            node.remove(obj)

        else:
            #remove old max
            if pmin1[1] != None:
                node.remove(pmin1[1])
            pmin1[0] = games[obj].pHistory[1][round]
            pmin1[1] = obj
            pmin1[2] = obj
            pmin1[3] = games[obj].round
            pmin1[4] = games[obj].pHistory[0][round]


#else dig deeper in obj for representative at current round:
else:
    temp = deepcopy(obj)
    goDeeper = True
    while goDeeper:
        ob = temp[0]
        if type(ob) == int:
            goDeeper = False
            if games[ob].history[round-1] == 0:   #0's move
                if games[ob].pHistory[0][round] > pmin0[0]:
                    node.remove(obj)

                elif games[ob].pHistory[0][round] == pmin0[0] and \
                games[ob].pHistory[1][round] < pmin0[4]:
                    node.remove(obj)

                elif games[ob].pHistory[0][round] == pmin0[0] and \
                games[ob].pHistory[1][round] == pmin0[4] and \
                games[ob].round >= pmin0[3]:
                    node.remove(obj)

                else:
                    if pmin0[1] != None:
                        node.remove(pmin0[1])
                    pmin0[0] = games[ob].pHistory[0][round]
                    pmin0[1] = obj
                    pmin0[2] = ob        #our representative
                    pmin0[3] = games[ob].round
                    pmin0[4] = games[ob].pHistory[1][round]

            else:    #1's move
                if games[ob].pHistory[1][round] > pmin1[0]:
                    node.remove(obj)
```

```
                        elif games[ob].pHistory[1][round] == pmin1[0] and\
                        games[ob].pHistory[0][round] < pmin1[4]:
                            node.remove(obj)

                        elif games[ob].pHistory[1][round] == pmin1[0] and\
                        games[ob].pHistory[0][round] == pmin1[4] and\
                        games[ob].round >= pmin1[3]:
                            node.remove(obj)

                        else:
                            if pmin1[1] != None:
                                node.remove(pmin1[1])
                            pmin1[0] = games[ob].pHistory[1][round]
                            pmin1[1] = obj
                            pmin1[2] = ob
                            pmin1[3] = games[ob].round
                            pmin1[4] = games[ob].pHistory[0][round]
                else:
                    temp = ob


# else an integer tree
else:
    pmin0 = [defaultp,None,None, None, None]
    pmin1 = [defaultp,None,None, None, None]
    # remove all bad objects from node, and remove old and new max and mins
    for obj in list(node):
        if type(obj) == int:
            if games[obj].history[round-1] == 0:  #PLAYER 0'S MOVE
                if games[obj].iHistory[0][round] > pmin0[0]:
                    node.remove(obj)

                elif games[obj].iHistory[0][round] == pmin0[0] and\
                games[obj].iHistory[1][round] < pmin0[4]:
                    node.remove(obj)

                elif games[obj].iHistory[0][round] == pmin0[0] and\
                games[obj].iHistory[1][round] == pmin0[4] and games[obj].round >= pmin0[3]:
                    node.remove(obj)

                else:
                    # if there's a previous object in pmin, now it must be removed
                    if pmin0[1] != None:
                        node.remove(pmin0[1])
                    pmin0[0] = games[obj].iHistory[0][round]
                    pmin0[1] = obj
                    pmin0[2] = obj
                    pmin0[3] = games[obj].round
                    pmin0[4] = games[obj].iHistory[1][round]


            else:                                #PLAYER 1'S MOVE
                if games[obj].iHistory[1][round] > pmin1[0]:
                    node.remove(obj)

                elif games[obj].iHistory[1][round] == pmin1[0] and\
                games[obj].iHistory[0][round] < pmin1[4]:
```

```
            node.remove(obj)

        elif games[obj].iHistory[1][round] == pmin1[0] and\
        games[obj].iHistory[0][round] == pmin1[4] and games[obj].round >= pmin1[3]:
            node.remove(obj)

        else:
            #remove old max
            if pmin1[1] != None:
                node.remove(pmin1[1])
            pmin1[0] = games[obj].iHistory[1][round]
            pmin1[1] = obj
            pmin1[2] = obj
            pmin1[3] = games[obj].round
            pmin1[4] = games[obj].iHistory[0][round]


#else dig deeper in obj for game:
else:
    temp = deepcopy(obj)
    goDeeper = True
    while goDeeper:
        ob = temp[0]
        if type(ob) == int:
            goDeeper = False
            if games[ob].history[round-1] == 0:     #0's move
                if games[ob].iHistory[0][round] > pmin0[0]:
                    node.remove(obj)

                elif games[ob].iHistory[0][round] == pmin0[0] and\
                games[ob].iHistory[1][round] < pmin0[4]:
                    node.remove(obj)

                elif games[ob].iHistory[0][round] == pmin0[0] and\
                games[ob].iHistory[1][round] == pmin0[4] and\
                games[ob].round >= pmin0[3]:
                    node.remove(obj)

                else:
                    if pmin0[1] != None:
                        node.remove(pmin0[1])
                    pmin0[0] = games[ob].iHistory[0][round]
                    pmin0[1] = obj
                    pmin0[2] = ob        #our representative
                    pmin0[3] = games[ob].round
                    pmin0[4] = games[ob].iHistory[1][round]

            else:                                   #1's move
                if games[ob].iHistory[1][round] > pmin1[0]:
                    node.remove(obj)
                elif games[ob].iHistory[1][round] == pmin1[0] and\
                games[ob].iHistory[0][round] < pmin1[4]:
                    node.remove(obj)
                elif games[ob].iHistory[1][round] == pmin1[0] and\
                games[ob].iHistory[0][round] == pmin1[4] and\
                games[ob].round >= pmin1[3]:
                    node.remove(obj)
```

```
                            else:
                                if pmin1[1] != None:
                                    node.remove(pmin1[1])
                                pmin1[0] = games[ob].iHistory[1][round]
                                pmin1[1] = obj
                                pmin1[2] = ob
                                pmin1[3] = games[ob].round
                                pmin1[4] = games[ob].iHistory[0][round]
                        else:
                            temp = ob


        #add p's at round above current round for the two games chosen above.
        needsP = [pmin0[2], pmin1[2]]
        addP(needsP, round-1)



    #make a copy of tree, and call recursionProp on the copy at round 1 (first move).
    btree = deepcopy(tree)

    recursionProp(btree, 1)

    #btree now contains only the games that are as good or better than sister games
    return btree
#---------------------------------------Print Functions-------------------------------------------
def printResults(btree,games,n,tfile):
    '''this function takes a tree and prints each game in the tree in the order games were
    constructed by recursionGames()'''

    x = str(btree)
    i = 0
    count = 0
    s = ''
    while i < len(x):
        if x[i] in '0123456789':
            s += x[i]
            if x[i+1] == ',' or x[i+1] ==']':
                count += 1
                s = ''
                i += 2
            else:
                i += 1
        else:
            i += 1

    obj0 = btree[0]
    while type(obj0) != int:
        obj0 = obj0[0]

    g0 = games[obj0]


    if g0.iHistory[0][0] < g0.iHistory[1][0] and g0.iHistory[0][0] <= n:
        code = '0'
    elif g0.iHistory[0][0] == g0.iHistory[1][0]:
```

```
        code = '5'
    elif g0.iHistory[0][0] > g0.iHistory[1][0] and g0.iHistory[1][0] <= n:
        code = '1'
    else:
        code = '?'
    print('n:',n,'--',code,'wins since','(',g0.iHistory[0][0],'*,',
    g0.iHistory[1][0],')','# of games:',count)

    if tfile != 0:
        print('n:',n,'--',code,'wins since','(',g0.iHistory[0][0],'*,',g0.iHistory[1][0],
        ')','# of games:',count,file=tfile)

def printTree(btree, games, n):
    '''this function takes a tree and prints each game in the tree in the order games were
    constructed by recursionGames()'''

    x = str(btree)
    i = 0
    count = 0
    s = ''
    while i < len(x):
        if x[i] in '0123456789':
            s += x[i]
            if x[i+1] == ',' or x[i+1] ==']':
                count += 1
                print('game', s)
                print('pHistory: ',games[int(s)].pHistory)
                print('bidpHistory: ',games[int(s)].bidpHistory)
                if n > 0:
                    print('iHistory: ',games[int(s)].iHistory)
                    print('bidiHistory: ',games[int(s)].bidiHistory)
                games[int(s)].getGame()
                s = ''
                i += 2
            else:
                i += 1
        else:
            i += 1
    print()
    print()
    print('total number of games in besttree is', count)
    print()

    obj0 = btree[0]
    while type(obj0) != int:
        obj0 = obj0[0]

    g0 = games[obj0]

    obj1 = btree[1]
    while type(obj1) != int:
        obj1 = obj1[0]
    g1 = games[obj1]

    print('player 0 needs p = (', g0.pHistory[0][1], "+", g1.pHistory[0][1], ") /2  = ",
    g0.pHistory[0][0] )
    print('player 1 needs p = (', g0.pHistory[1][1], "+", g1.pHistory[1][1], ") /2  = ",
```

```
g0.pHistory[1][0] )
print()

if n>0:
    print('player 0 needs i = floor[ (', g0.iHistory[0][1], "+", g1.iHistory[0][1],
    ") /2 ] = ", g0.iHistory[0][0] )
    print('player 1 needs i = ceil[ (', g0.iHistory[1][1], "+", g1.iHistory[1][1],
    ") /2 ] = ", g0.iHistory[1][0] )

print()
ans = input('Would you like to write this to a file?')
print()
if ans[0] == 'Y' or ans[0] == 'y':
    name = input('filename: ')
    tfile = open(name, 'w')
    print(file=tfile)
    print('THE TREE',file=tfile)
    print(file=tfile)
    print(btree,file=tfile)
    print(file=tfile)
    print(file=tfile)

    i = 0
    count = 0
    s = ''
    while i < len(x):
        if x[i] in '0123456789':
            s += x[i]
            if x[i+1] == ',' or x[i+1] ==']':
                count += 1
                print('game', s,file=tfile)
                print('pHistory: ',games[int(s)].pHistory,file=tfile)
                print('bidpHistory: ',games[int(s)].bidpHistory,file=tfile)
                if n>0:
                    print('iHistory: ',games[int(s)].iHistory,file=tfile)
                    print('bidiHistory: ',games[int(s)].bidiHistory,file=tfile)
                print('**************************************************',file=tfile)
                print(file=tfile)
                self = games[int(s)]
                if self.winner == None:
                    if self.tie:
                        print('Game is a tie',file=tfile)
                    else:
                        print('Game is not complete',file=tfile)
                else:
                    print('Winner is player', self.winner,file=tfile)

                print(file=tfile)
                print("history of moves: ", self.history,file=tfile)
                print('advHistory ', self.advHistory,file=tfile)

                print(file=tfile)
                for j in [0,1,2]:
                    for k in [0,1,2]:
                        print(' ', end = '',file=tfile)
                        if self.board[j][k] != None:
                            print(self.board[j][k], end = '',file=tfile)
```

```
                        else:
                            print(" ", end = '',file=tfile)
                        print(' ', end = '',file=tfile)
                        if k != 2:
                                print('|', end = '',file=tfile)
                    print(file=tfile)
                    if j != 2:
                        print('-----------',file=tfile)
                print(file=tfile)

                print(file=tfile)
                print("history of cordinates chosen: ", self.placeHistory,file=tfile)
                print(file=tfile)
                print('**************************************************',file=tfile)
                s = ''
                i += 2
            else:
                i += 1

        else:
            i += 1

    print(file=tfile)
    print(file=tfile)
    print('total number of games in besttree is', count,file=tfile)
    print(file=tfile)
    print('player 0 needs p = (', g0.pHistory[0][1], "+", g1.pHistory[0][1], ") /2  = ",
    g0.pHistory[0][0],file=tfile )
    print('player 1 needs p = (', g0.pHistory[1][1], "+", g1.pHistory[1][1], ") /2  = ",
    g0.pHistory[1][0],file=tfile )
    print(file=tfile)

    if n>0:
        print('player 0 needs i = floor[ (', g0.iHistory[0][1], "+",
        g1.iHistory[0][1], ") /2 ] = ", g0.iHistory[0][0],file=tfile )
        print('player 1 needs i = ceil[ (', g0.iHistory[1][1], "+",
        g1.iHistory[1][1], ") /2 ] = ", g0.iHistory[1][0],file=tfile )

    print(file=tfile)


    print("data written to file")
    print()
    print()


#-----------------------------------SAVE and LOAD-----------------------------------------------

# these next two functions are for saving/loading games data by using python's pickling functions
def load():
    gamesdata1 = open('gamesData1.data', 'rb')
    gamesdata2 = open('gamesData2.data', 'rb')


    games1=pickle.load(gamesdata1)
    games2=pickle.load(gamesdata2)
```

```
    games = list(games1 + games2)
    gamesdata1.close()
    gamesdata2.close()
    return games

def save():
    global games
    games = tuple(games)
    games1 = games[:len(games)//2]
    games2 = games[len(games)//2:]
    gamesdata1 = open('gamesData1.data', 'wb')
    gamesdata2 = open('gamesData2.data', 'wb')
    pickle.dump(games1, gamesdata1, 3)
    pickle.dump(games2, gamesdata2, 3)

    gamesdata1.close()
    gamesdata2.close()

#-----------------------------------------------Research Functions----------------------------
def byprompt():
    '''this function allows the user to specifiy player-opponenent strategy and an n dollar amount.
    the function outputs the best tree with these inputs.'''
    global games
    global tree
    interested = True
    while interested == True:
        print()
        print()
        print()
        print()
        print("ID's for gameplays: \n 1. W vs DL \n 2. T vs T \n 3. W vs W \n 4. W vs T")
        print(" 5. T vs DL \n 6. DL vs DL \n 7. TF vs T")
        print()
        id = input("enter gameplay ID for finding best way to play, q to quit: ")
        if id == 'q':
            print()
            print("bye!")
            print()
            print()
            break
        else:
            id = int(id)
        print()
        mode = input("type 1 for fractions, 2 for integers:" )
        if mode == '1':
            n = -1
        elif mode == '2':
            n = int(input("enter each player's starting amount of money (integer n): "))
        print()
        print('calculating...')
        print()


        #call best games with tree, and catch best tree.
        btree = findBest(games, tree,n,id,0)

        print('besttree', btree)
```

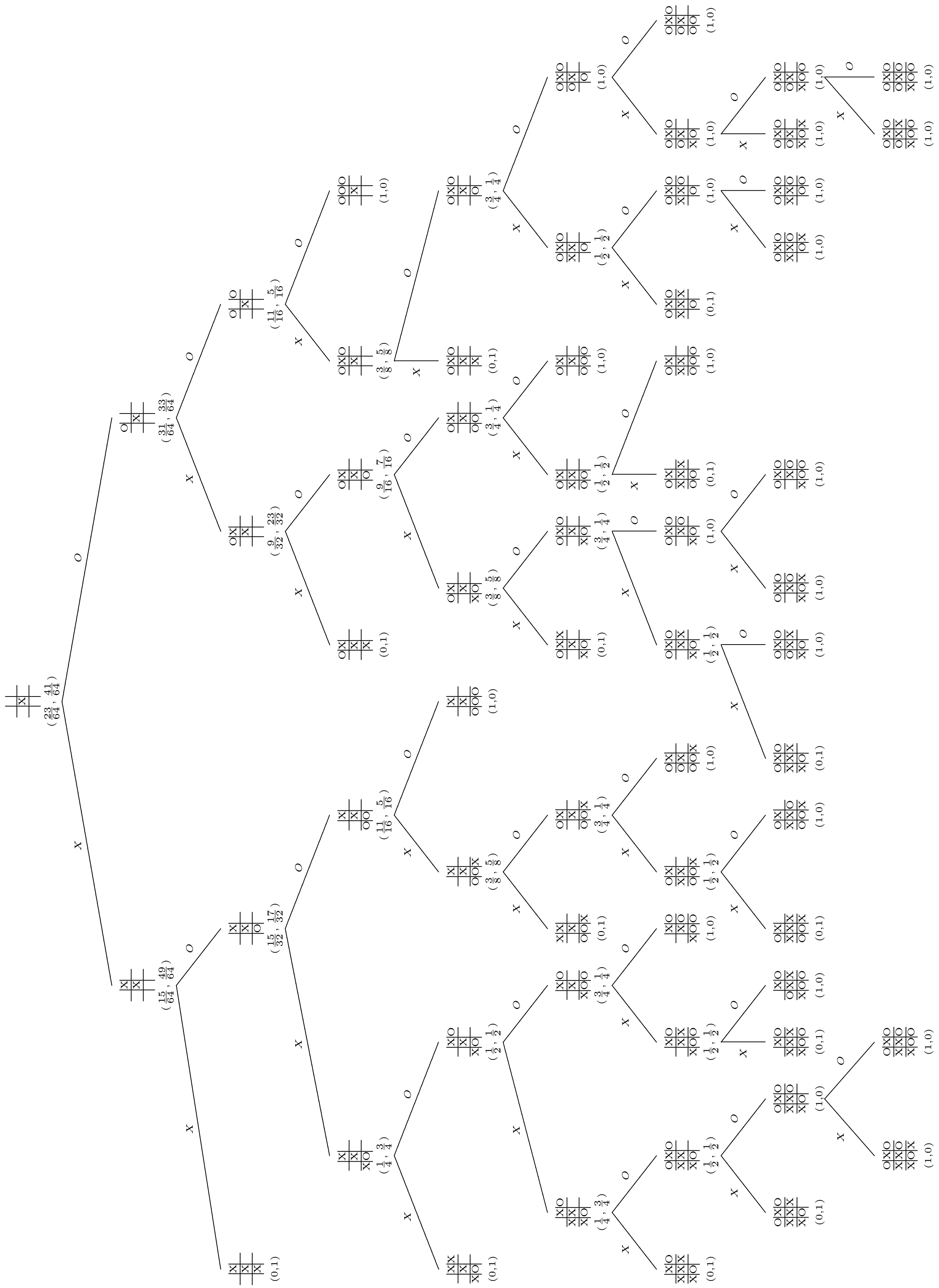
```
        print()

        printTree(btree, games,n)


def research():
    '''This function allows user to specify player-opponent stategy and a sequence of n values for
    which the outcome is desired. Results are printed to terminal and available to be written
    to a file.'''
    global games
    global tree
    interested = True
    while interested == True:
        print()
        ans = input('Would you like to write the research results to a file?')
        print()
        if ans[0] == 'Y' or ans[0] == 'y':
            name = input('filename: ')
            tfile = open(name, 'w')
            print('*****************************************************************',file=tfile)
        else:
            tfile=0
        print()
        print("ID's for gameplays: \n 1. W vs DL \n 2. T vs T \n 3. W vs W \n 4. W vs T")
        print(" 5. T vs DL \n 6. DL vs DL \n 7. TF vs T")
        print()
        ids = input("enter gameplay IDs for finding what's possible, q to quit: ")
        if ids == 'q':
            print()
            print("bye!")
            print()
            print()
            break

        print()
        minN = int(input('enter minN where integers minN through maxN will be calculated: '))
        maxN = int(input('enter maxN: '))
        print()
        print()
        print()

        for id in '1234567':
            if id in ids:
                print('CURRENT ID:', id)
                if tfile != 0:
                    print('CURRENT ID:', id, file=tfile)

                id = int(id)
                ndollar = minN
                while ndollar <= maxN:
                    #call best games with tree, and catch best tree.
                    btree = findBest(games, tree, ndollar, id, 1)

                    printResults(btree, games,ndollar,tfile)
                    ndollar += 1
        if tfile != 0:
            tfile.close()
```

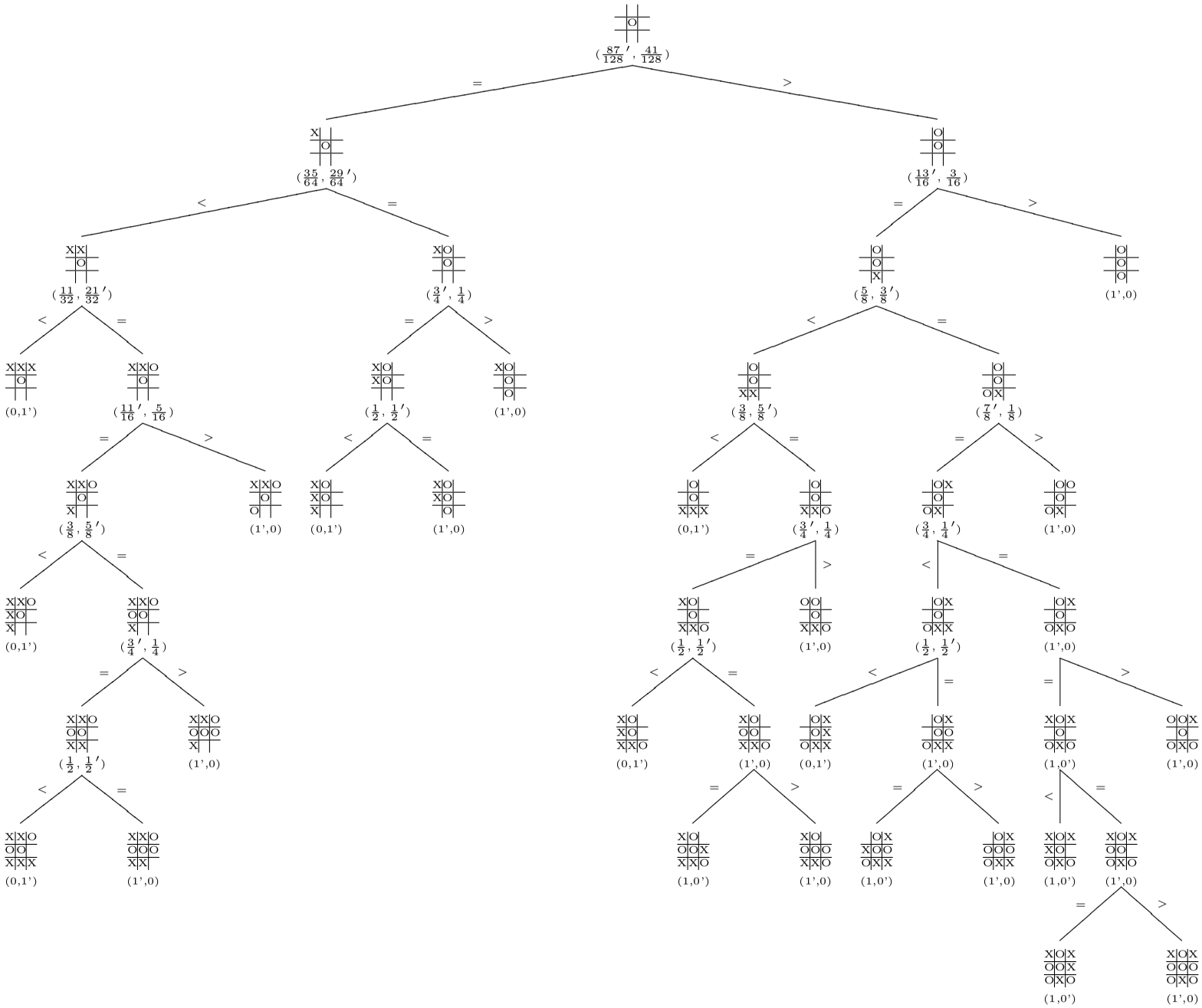
```
#----------------------------------------MAIN function-----------------------------------------
#put these variables on the outermost level, for use in all the functions.
tree = []
games = []

# p-prices for 0's win, 1's win, tie: by player, by gameplay
gameplays = [[[0.0,1.0,1.0],[1.0,0.0,0.0]],[[0.0,1.0,0.5],[1.0,0.0,0.5]],[[0.0,1.0,1.0],
[1.0,0.0,1.0]],[[0.0,1.0,1.0],[1.0,0.0,0.5]],[[0.0,1.0,0.5],[1.0,0.0,0.0]],[[0.0,1.0,0.0],
[1.0,0.0,0.0]],[[0.0,1.0,0.5],[1.0,0.0,0.5]]]


def main():
    global games
    global tree
    game = Game(simulation = True)
    game.bids1 = [0,0,0,0,0,0,0,0,0]
    game.bids0 = [0,0,0,0,0,0,0,0,0]
    #regularTTT(game)


    #call recursion games to get all games, and catch the finished tree
    print()
    print('finding all ways to play...')
    tree = recursionGames(game)
    print()
    print('Recusion finished. Games found: ', len(games))
    print('Game indices placed in tree of all games.')
    print()
    print()
    ans = input("Enter 1 for individual n dollar output, 2 for a set of n: ")
    if ans == "1":
        byprompt()
    elif ans=="2":
        research()
    #-----------------------Code for saving and loading--------------------------------
    '''save()
    print("SAVED")
    games=None
    games=load()
    print("LOADED")

    print('length of games', len(games))'''




    #_____________________________________________________________________________________


if __name__ == '__main__':
    #games=load()
    #print("LOADED")
    #print('length of games', len(games))
```

```
    main()

else:
    print("functions loaded")
```

6.2 Example W vs. DL p-price Tree

Attached is the W vs. DL p-price tree, split into two sides, one where X goes first, and one where O goes first.


($\frac{23}{64}$, $\frac{41}{64}$)
X
O
($\frac{15}{64}$, $\frac{49}{64}$)
($\frac{31}{64}$, $\frac{33}{64}$)
(0,1)
(1,0)


(87/128′, 41/128)
(35/64, 29/64′)
(13/16′, 3/16)
(11/32, 21/32′)
(3/4′, 1/4)
(5/8, 3/8′)
(1′,0)
(0,1′)
(11/16′, 5/16)
(1/2, 1/2′)
(3/8′, 5/8)
(7/8′, 1/8)
(3/8, 5/8′)
(3/4′, 1/4)
(3/4, 1/4′)
(1/2, 1/2′)
(1,0′)

6.3 TF vs. T Outcomes Table

Attached is the TF vs. T outcomes table obtained through the computer program.

n	TF vs T	n: 129-256	n: 257-384	n: 385-512	n: 513-640
1	0	0	0	0	0
2	0	0	0	0	0
3	1	1	1	1	1
4	0	0	0	0	0
5	0	0	0	0	0
6	1	1	1	1	1
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0
10	1	1	1	1	1
11	0	0	0	0	0
12	0	0	0	0	0
13	0	0	0	0	0
14	0	0	0	0	0
15	0	0	0	0	0
16	0	0	0	0	0
17	0	0	0	0	0
18	1	1	1	1	1
19	0	0	0	0	0
20	0	0	0	0	0
21	0	0	0	0	0
22	1	1	1	1	1
23	0	0	0	0	0
24	0	0	0	0	0
25	0	0	0	0	0
26	0	0	0	0	0
27	0	0	0	0	0
28	0	0	0	0	0
29	0	0	0	0	0
30	1	1	1	1	1
31	0	0	0	0	0
32	0	0	0	0	0
33	0	0	0	0	0
34	1	1	1	1	1
35	0	0	0	0	0
36	0	0	0	0	0
37	0	0	0	0	0
38	0	0	0	0	0
39	0	0	0	0	0
40	1	1	1	1	1
41	0	0	0	0	0
42	1	1	1	1	1
43	0	0	0	0	0
44	0	0	0	0	0
45	0	0	0	0	0
46	1	1	1	1	1
47	0	0	0	0	0
48	0	0	0	0	0
49	0	0	0	0	0
50	0	0	0	0	0
51	0	0	0	0	0
52	0	0	0	0	0
53	0	0	0	0	0
54	1	1	1	1	1
55	0	0	0	0	0
56	0	0	0	0	0

KEY	
1	player 1 achieves goal, player 0 does not
0	player 0 achieves goal, player 1 does not

n	TF vs T	n: 129-256	n: 257-384	n: 385-512	n: 513-640
57	0	0	0	0	0
58	1	1	1	1	1
59	0	0	0	0	0
60	0	0	0	0	0
61	0	0	0	0	0
62	0	0	0	0	0
63	0	0	0	0	0
64	1	1	1	1	1
65	0	0	0	0	0
66	0	0	0	0	0
67	0	0	0	0	0
68	1	1	1	1	1
69	0	0	0	0	0
70	0	0	0	0	0
71	0	0	0	0	0
72	0	0	0	0	0
73	0	0	0	0	0
74	0	0	0	0	0
75	0	0	0	0	0
76	1	1	1	1	1
77	0	0	0	0	0
78	0	0	0	0	0
79	0	0	0	0	0
80	1	1	1	1	1
81	0	0	0	0	0
82	0	0	0	0	0
83	0	0	0	0	0
84	0	0	0	0	0
85	0	0	0	0	0
86	0	0	0	0	0
87	0	0	0	0	0
88	1	1	1	1	1
89	0	0	0	0	0
90	0	0	0	0	0
91	0	0	0	0	0
92	1	1	1	1	1
93	0	0	0	0	0
94	0	0	0	0	0
95	0	0	0	0	0
96	0	0	0	0	0
97	0	0	0	0	0
98	0	0	0	0	0
99	0	0	0	0	0
100	1	1	1	1	1
101	0	0	0	0	0
102	0	0	0	0	0
103	0	0	0	0	0
104	1	1	1	1	1
105	0	0	0	0	0
106	0	0	0	0	0
107	0	0	0	0	0
108	0	0	0	0	0
109	0	0	0	0	0
110	0	0	0	0	0
111	0	0	0	0	0

n	TF vs T	n: 129-256	n: 257-384	n: 385-512	n: 513-640			
112	1	1	1	1	1			
113	0	0	0	0	0			
114	0	0	0	0	0			
115	0	0	0	0	0			
116	1	1	1	1	1			
117	0	0	0	0	0			
118	0	0	0	0	0			
119	0	0	0	0	0			
120	0	0	0	0	0			
121	0	0	0	0	0			
122	1	1	1	1	1			
123	0	0	0	0	0			
124	0	0	0	0	0			
125	1	1	1	1	1			
126	1	1	1	1	1			
127	0	0	0	0	0			
128	1	1	1	1	1			