### Ultimate Tic Tac Toe

### WHAT THE PROJECT IS ABOUT

At some point in everyone’s life, the game of Tic-Tac-Toe has made an appearance. Typically this encounter occurs in the younger years of the individual. Before much exposure, players will play randomly or without much strategy at least. But it is not long before one realizes playing first is greatly advantageous and furthermore marking the middle spot on the board is best. With a little bit of concentration and trial and error, an individual can master the simple game of tic tac toe as the first or second player. At this point, unfortunately, the game is no longer interesting and a new intriguing game is sought after.

Due to the familiarity of Tic-Tac-Toe, our team decided to pursue a project focused on *ultimate* tic tac toe. “What is Ultimate Tic-Tac-Toe?”, you ask? Simply stated, it is an extension of regular Tic-Tac-Toe that is more challenging and complex. Instead of having just one game, we now have a **game board** that consists of 9 regular Tic-Tac-Toe games called **blocks**. Each block consists of 9 **cells** that either player can play in. Consider it Tic-Tac-Toe-ception, if you will. The rules are as follows: At the beginning of the game, player 1 has the option to play in any cell on the game board. That is, 9 blocks x 9 cells/block = 81 initial options. After the initial move, player 2 must play in the block that mirrors the cell from the previous player. Consider the following example for clarity-sake: Imagine a player is required to play in the top-right block of the game board and they select the center cell of that block. The next player is then forced to play in the center block of the game board, but that is the only restriction. If at any point a player makes a move that forces the next player to play on a block that is completed (a draw, player 1 has won, or player 2 has won), that player is given the ability to choose which block to play on. The goal is to win 3 blocks in-a-row on the game board.

After realizing the amount of work our project would entail, we decided to modify some of the rules to make the project more reasonable with only 1.5 months to work. One original rule we changed was how to win - instead of finishing a game by winning 3 blocks in-a-row of a given player’s piece, we restricted it to only requiring a player to win a single block. There was also a rule where if one player played in any corner cell of the center block, then the opponent had the option to play in any block that the previous cell was physically touching. For example, if player one played in the bottom-left cell of the center block, then the opponent could play in the center-left block, the bottom-left block, or the bottom-middle block. We removed this rule.

### WHAT PROBLEM OUR PROJECT IS SOLVING

The problem our project focused on is very simple conceptually, as is probably expected. It was to develop an artificial intelligent agent that is capable of winning Ultimate

Tic-Tac-Toe games. Our goal was for our AI engine to win more often than not, whether it be against a human player or a bot with predefined rules.

### HOW OUR PROJECT SOLVES THE PROBLEM

With little amount of time to start a project this large from scratch, we found pre-existing code online from a couple of different sources to give us a nice base for what we hoped to accomplish. In hindsight it may have been more beneficial to write code from scratch due to the convolutedness and bad programming practices that were inescapable. The first of our two files, **UTTT.py**, comes from <https://github.com/rohitsakala/ultimateTicTacToe>, which contains the main program and two computer bots that are not considered artificial intelligence. The second file, **PlayerAI.py**, which comes from <https://github.com/tarun018/AI-for-Ultimate-Tic-Tac-Toe> consists solely of the artifical intelligent agent that uses alpha-beta minimax search to choose its move. We integrated these 2 files to interact with each other so the former bots had the opportunity to face the more intelligent entity.

Here include what the 2 unartificial intelligent bots do:

* The Easy Bot would just play randomly.
* The Medium Bot would try to block, play in corners, or randomly.

The main modification we made in PlayerAI.py was changing the heuristic. The original heuristic was based on the original way of winning, which was claiming 3 blocks in-a-row of your specified piece. Our heuristic, described in our evaluation function, is defined as follows:

1. Iterate through each block on the game board and count the number of opponent’s pieces per block.
2. Iterate through each count (there will be 9 total) and find the minimum
3. Return the number inverted (more to come on this)

Let’s look at a simple example to clarify the heuristic. Say the opponent has 1 piece in block 0, 2 in block 1, 3 in block 2, 4 in block 3, 5 in block 4, 4 in block 5, 3 in block 6, 2 in block 7, and 1 in block 8. We store each of these numbers in an array of size 9 with indices representing the block it pertains to. Next we use a for loop to iterate over each array component and find the minimum. Then we take that minimum and make it a relatively large number if it is small and relatively small if it is a small number. The reasoning behind this is that we want to maximize our play, so a block with less amount of opponent pieces is more favorable in our eyes, thus we return a larger value.

Another modification we made was deepening the search of the tree for each turn.

### ADDITIONAL MODIFICATIONS WE MADE

With the original code we found, there was one rule that seemed quite peculiar to us as a team - the 3 corner rule. If a move was made in one of the corner blocks of the game

board, then the next player had the option to play in the same corner block or either of the two adjacent blocks. We believed making a heuristic for the artificial intelligence engine to abide

by this rule would be quite difficult, so we did away with it entirely. We replaced it with the more intuitive rule - when a player makes a move in a corner block, the next move is to also be played in that corner block.

Because our program is run using terminal commands, our GUI is, simply, a nicely formatted output of the game board in its current state sent to the terminal. It also invokes the user for input when it is that player’s turn along with exhibiting what move they chose to make. Originally the GUI displayed the current state of the game board along with the winner of each block directly below it. Due to our “win rule” change, we felt it unnecessary to list the winner of each block because the game would terminate after one block was taken. Therefore we modified it to instead display the current block that is to be played on along with valid coordinates (row, column) to input for their move. Before this refinement user experience was poor. Explicitly, the user had to calculate the row and column numbers and if done incorrectly, an invalid move would be made, which results in termination of the game with an automatic loss. We did not approve, so we made the appropriate changes that were needed.

With the integration of the AI agent into the main program we were required to add more game options so it could face opponents. Previously, the options existed:

* Human Player vs. Easy Bot
* Human Player vs. Medium Bot
* Easy Bot vs. Medium Bot.

We felt it necessary to test our AI player class against the other bots, as well as a Human Player, to understand how well it performed. Thus, we added three more options:

* Human Player vs. AI agent
* Easy Bot vs. AI agent.
* Medium Bot vs. AI agent.

We tested the AI and the bots by running them 20 times for each pair:

* With 0 - 6 return values for the heuristic:
  + AI & Easy: AI won 80%
  + AI & Medium: AI won 80%
  + Easy & Medium: Medium won 55%
* With 0 - 60,000 return values for the heuristic:
  + AI & Easy: AI won 95%
  + AI & Medium: AI won 75%
  + Easy & Medium: Medium won 55%

### 

### TEAM MEMBER CONTRIBUTIONS

As a team, we decided that pairs programming would be the most beneficial for this project, enabling us to bounce ideas off one another and limiting the chance for errors. We worked on the project during thirteen sessions.

* In the first session, the full team met to decide the basis of our project.
* Stephanie analyzed the original code in UTTT.py and made comments describing functionality.
* Shannon and Stephanie went through and made more comments in the original code while Jeremy disallowed the game from allowing a cluster of 3 boards to play on when previous play was in a corner cell, to match the new rules of our game.
* Sang and Jeremy deciphered the evaluation function for bot1.
* Stephanie and Shannon made the output readable for humans, displayed the current block to make a move, and made part of the code more object-oriented.
* Jeremy and Stephanie updated the first move’s output, implemented draw functionality, and changed the program to win by one block instead of three blocks.
* Shannon and Sang updated the output to show the winner’s block for the final move, which was the last step to finish for the output.
* As a full team, we found code that had artificial intelligence with alpha-beta minimax search and integrated it with our existing code. The AI played using the old rules, so we had to update it’s valid moves and heuristic. We also ran win statistics and saw that one of our bots would beat the artificial intelligence bot more than half of the time.
* Jeremy and Stephanie updated the heuristic to win by one block instead of three blocks, counting the number of opponent’s pieces on each block for our evaluation function.
* All team members worked with partners to comment our code. Sang and Shannon ran win statistics on the different bots while changing the alpha-beta depth, to see which depth was preferred.
* Jeremy and Stephanie completed the project proposal, Sang and Shannon did the progress reports, and all of us contributed to this final paper. All of us met to work on the presentation.