Philron Hozier
COMPSCI383-SEC01
SP16 Final project
29233316

Introduction and Discussion

Ultimate tic tac toe is played within a three by three matrix (Figure 1 and Figure 2). Within each square or cell of the matrix is a normal three by three tic tac toe board. Players can be either x or o. The game is won by winning three “cells” or squares in a row or on the diagonal. A brief overview of features within the game enables a clearer understanding of how the provided implementation chooses to characterize (and/or define) the state of a board. According to ArtInt-ch9, “[…I]nstead of reasoning explicitly in terms of states, it is often better to describe states in terms of features and then to reason in terms of these features. Often these features are not independent and there are hard constraints that specify legal combinations of assignments of values to variables.” The generalization, or abstraction of the state space allows us to break apart the problem into more manageable units from which a solution can be derived. In our case, there exists a feature for each "cell" of the three by three tic tac toe board that species whether the "cell" is an x or an o. There also exists a feature for each square of the larger, main three by matrix within which the tic tac toe boards reside. The feature for this square is also either an x or an o. As a result of this characterization, our bot models the whole board.

The persistence of states throughout the game’s life allows us to peek and observe the next best move beyond the next move before having our bot choose it’s optimized next play. The definition of features within the game provides an easy backdrop for the game’s algorithm. The approach of the implemented strategy focuses heavily on the reduction (minimization) of the opponent’s chance at a next, successful move. More pointedly, the bot is written with heuristic evaluation functions which are often “used by game-playing programs to estimate the value or goodness of a position in the minimax and related algorithms minimax algorithm.” The strategy is such that weights are assigned to all possible remaining legal moves and later optimized for the bot’s turn. We should should note that for all intents and purposes, we define “move” to be the current (x, y) position within our tic tac toe expanded nine by nine grid – where the overall state space is computed as eighty one distinct positions. All possible moves which are legal and give the opponent advantage are weighted less as a consequence of the algorithm. The result is such that our bot always makes the more advantageous play—all the while leaving the less intelligent move (the less weighted moves within our legal moves vector) to our opponent.

The extension of the bot implementation can be seen as an execution in three parts. The Heuristics class within algorithm.py evaluates and assigns weights accordingly to legal next moves through search and optimization methods. The Position class within position.py analyzes the player’s current position and next moves to determine winning, losing and drawing states. There also exist helper methods within this class that which describe with greater certainty a possible move’s success based on it’s immediate neighbors within the square. The Bot class within stewie.py represents the engine in a sense by which the bot runs. The Bot class’s main responsibility is return the next move which we deem as best through our written algorithm.

Methods

The entry point of our bot in execution begins through the Bot class with a call to get_move(self, pos, tleft). Using heuristic evaluation functions as a basis by which optimal moves for the bot are

determined, we first create a new instance of our algorithm class, poll for all next available legal moves and instantiate a key, value pair collection (payoff) which will hold the weighted value for each remaining legal move.

Beyond the initial set up, our Bot class proceeds to break apart the problem of finding the next optimal move within six main steps. Because heuristic evaluation function estimates the cost of an optimal next step, I believe using iterative steps as a searching solution would prioritize speed over accuracy.

Although the Minimax algorithm traditionally falls more squarely inline with tic tac toe solutions, it's recursive strategy can be seen as a pitfall when time is a factor which plays into an optimal solution. In the end, I believe that the use of heuristics coupled with searching allowed the algorithm to not only be efficient, but also extensible for further optimization. The algorithm which follows further expresses how the Heuristics class to search for the problems solution.

For each move within the key, value pair legal move collection

1. Build a three by three board with respect to the current position of the observable move

(This board can be thought of as a cell within the larger three by three ultimate tic tac toe matrix.)

More pointedly, for each each tuple, send its x,y coordinates in and build a microboard from it, i.e.:

Given move: (0, 1), the board returned corresponding to that move will be since (0, 1) sits within that particular "cell."

[(0,0), (0, 1), (0, 2)
(1,0), (1, 1), (1, 2)
(2,0), (2, 1), (2, 2)

2. Return built board as well as the index to which the move is mapped to.
3. Call search_square(self, legal_moves, payoff, microboard, index, move, pos, bot) using the built board. search_square() calls six distinctive methods of the Position class which not only check against our bot's next successful move, but also against the possibility of our opponent having a successful move of their own. The two main cases are as follows:

   If our bot's next possible move is a good or winning move, have the heuristic evaluation function *increase* the current weight on that move. Not all good moves are weighted the same, however all evaluation of good moves result in a higher chance that our bot will choose that coordinate as the best choice to play.

   If our opponent's next possible move is a good or winning move, have the heuristic evaluation function *decrease* the current weight on that move.
4. Call is_opponent_playing_in_neighborhood(index, microboard, pos, payoff, move, self) to provide deeper insight into the state of the current "cell" where the move resides. if there exists a position within the subset (three by three board) that is currently near the move being explored from the legal_moves list AND that position is also our previously played move increase the weight of the current "move" by X. Returns a new, untethered copy of the Position instance.

5. Now that we have a new Position instance with slightly more augmented/optimized heuristics from the previous step, proceed to simulate making a next move with the current “move” copy.
6. Calling observe_next_next_moves(index, next_pos, payoff, move, pos, self) provides even greater insights into what would occur, given that we made a second move beyond the first move made in step five. We again run our heuristic evaluation function observing all the available next legal moves and then proceeding to increase/decrease weights accordingly for all legal moves observed.

Lastly we invoke select(pos, payoff, self) to select and return the best possible move for our bot within our payoff key, value vector.

Code snippet of the main bot algorithm
The above algorithm explains the code below:

```
    ''' @pos: an instance of the Position class
        @tleft: time remaining

        other relevant args
        @payoff: a key,value collection of moves associated with thier heuristic
        @move: a coordinate on the tic tac toe board
        @legal_moves: all the available next legal moves remaining on the board
        @self: an instance of the Bot class
    '''
    def get_move(self, pos, tleft):
        algorithm = a.Heuristics()
        legal_moves = pos.legal_moves()

        # overview: if a list is not returned, return the empty string.
        if not legal_moves:
            return ''

        # overview: create a dictionary named payoff and for each tuple
        # within the legal_moves list, assign the value 0 to it.
        payoff = self.new_dict(legal_moves)

        for move in legal_moves:
          microboard, index = pos.get_microboard(move[0], move[1])
            algorithm.search_square(legal_moves, payoff, microboard, index, move, pos, self)

            next_pos = algorithm.is_opponent_playing_in_neighborhood(index, microboard, pos, payoff,
move, self)
            next_pos.make_move(move[0], move[1], self.myid)
            algorithm.observe_next_next_moves(index, next_pos, payoff, move, pos, self)
        return algorithm.select( pos, payoff, self)
```

Results and Conclusions
My initial implementation considered the Minimax algorithm. The algorithm “is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario.” It seemed like a natural fit at first given that Minimax is oftentimes used as a strategy for derived best possible chances in general two-player games.

The algorithm itself is implemented recursively and as such I ran into many issues when attempting to write the solution with respect to the ultimate tic tac toe board. Ultimately, I settled on the more heuristics based approached which derived next best moves based on in depth calls to the heuristic evaluation functions. The results were as desired and – as seen – always attempted to favor our bot’s success over our opponent’s.

The expressiveness of the heuristic evaluation functions enabled clarity within the implementation when weighting observed states. As seen in the following code snippet, if an opponent’s bot would perform better under the observed next move, our evaluation function attempts to avert such success by minimizing the weight of the move.

Note: bot.oppid is the opponent’s ID.

```
    ''' can the opponent win? adjust.'''
    if pos.is_winner(next_move, bot.oppid):
        payoff[move] -= 10
```

The success of the heuristic evaluation function proved to be a good solution, however, future possibilities for improving the bot would definitely include a more unified, conjunctive implementation which views heuristics as a supplement and not solution. Having the Minimax algorithm play a more central role with respect to deriving the next best optimal moves is a future enhancement which I do believe is possible given how closely the algorithms can work alongside one another.

Please note that the link to the discussion forum posting is not on Moodle and/or could not be found.

Additional Resources
Github: https://github.com/hozier/final.cmpsci383/
GitHub username: hozier
AI Games username: phozier_Ultimate

Screenshots

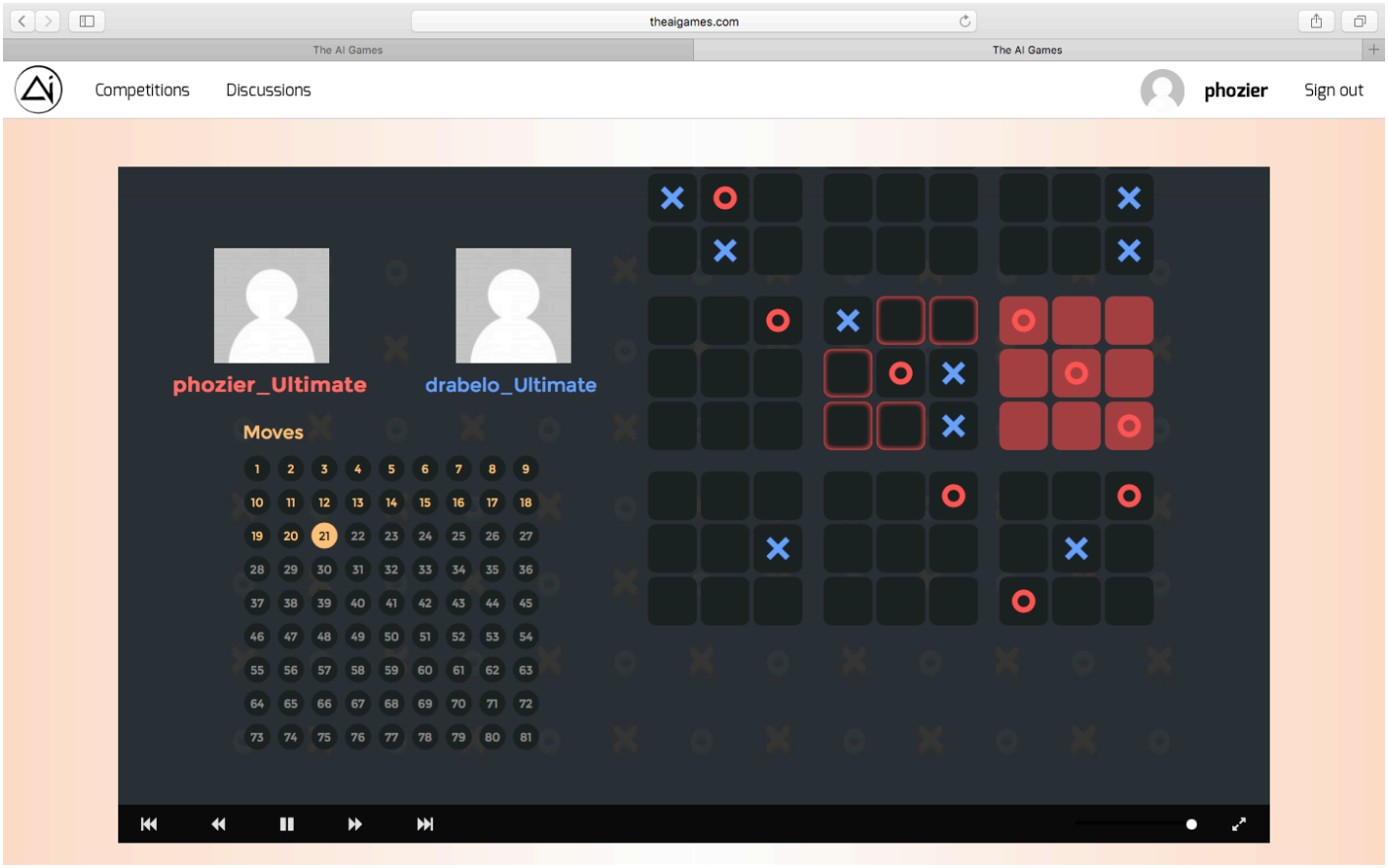

Figure 1.

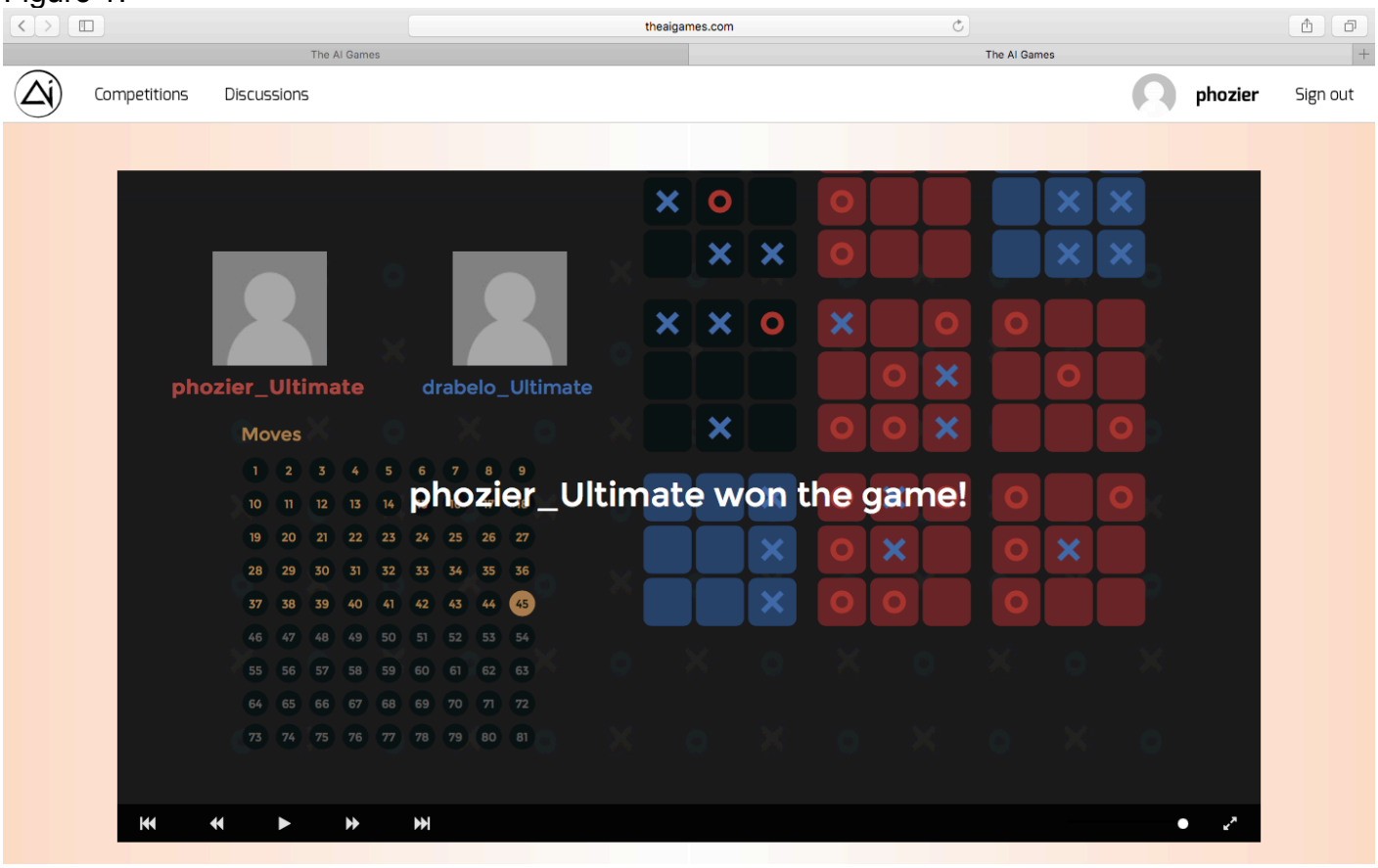

Figure 2.