Name: Chengjing Li, Catherine Nguyen, Gayathri Talasila, Hengchuan Zhang

Project 3 Report

1. Project Description:

The Project requires to implement Generalized Tic Tac Toe using Minimax Alpha-Beta pruning algorithm.

2. Tic Tac Toe and Minimax:

We have designed a Board class that has players (to represent X or O), boardSize, target, user (to represent which player we are), other (to represent which player the opponent is), and most importantly, a board that is represented as a 2-D numpy array for faster operations.

```
class Board:
   def __init__(self, size=10, target=5,
user=0):
       self.size = size
       self.totalMoves = size**2
       self.target = target
       self.board = np.zeros((size, size),
                             dtype=np.int8)
       self.players = ['O', 'X']
       self.user = user
       self.other = 0 if user == 1 else 1
       self.win = False
       self.winner = -1
```

Our Minimax algorithm utilizes Alpha Beta Pruning for clearing out redundant searches. Many operations such as getting available spots, adding and removing spots on board is made simpler by using the Board class operations.

3. Heuristic Function:

The heuristic function that we decided to use calculates the potential of wins at each location on the board. At each point on the board, the row, column, and diagonals starting at that point, and has a length of target, is calculated (we refer to this as each *line*). At each line, we only calculate the heuristic if the line is as long as the target, and if there is not both players on it.

An example board:

X	O	
	O	
X		

If we are analyzing the point [1 1], its lines are [O, O, -], [X, O, -], [-, O, X] and [-, O, -]. But heuristic is only calculated for [O, O, -] and [-, O, -] since they are the only lines that have the potential to win.

Heuristic is calculated by counting the number of existing moves and the free spaces. If all spaces in the line are filled with the same players, then it is a win, and returns a large value.

The algorithm for calculating scores per line:

```
def perLine(line, target):
    if 1 in line:
        turn = 1
    else:
        turn = -1

    nonZeros = np.count_nonzero(line)

    if nonZeros == target:
        return 10000 * turn

    if nonZeros == target - 1:
        return 500 * turn
    if nonZeros == target - 2:
        return 200 * turn

    return (nonZeros * 5 + target
                - nonZeros) * turn
```

Otherwise there is a tradeoff between the number of spaces available and the amount of pieces already in that line. We put more emphasis on the pieces that already exist on that line, compared to the number of spaces, though they also contribute to the total score.

4. Optimization:

Though the basic minimax works with a 3x3 board, we needed a lot of optimization in order to play bigger boards. This includes changing the board from a 2-D list to a 2-D numpy array for faster operations. Aside from alpha beta pruning, we also have the maxDepth be set dynamically while playing the game in order to get the best minimax results without compromising the game.

We previously checked the board for wins every iteration, but realized that if we kept a local board while the game was running, we could simply check only the *lines* of that move every time a move was made. This reduces the checkWin operation significantly from O(n*n) every minimax iteration to only O(n).

```
def add(self, move, player):

    if player == self.user:
        self.board[move[0]][move[1]] = 1
        turn = 1
    else:
        self.board[move[0]][move[1]] = -1
        turn = -1

    self.win = self.checkWin(move, turn)
    if self.win:
        self.winner = turn
```

We also utilize a similar - but smaller - heuristic function than the one described above. This is to sort the available moves before selection for minimax, for the most optimal alpha beta pruning. This way, the best move is checked first, leading to more efficient pruning.