Checkpoint 3: Final Specification

CS51: Final Project

Kristina Hu
Hana Kim
Alexander Solis
Stephen Xi

Detailed Description

In recent years, algorithmic game theory has become a popular topic of scholarly interest. In games such as Tic Tac Toe, where both players compete to place their markers over a limited number of spots to achieve the same end goal (three in a row), game theory provides us with the tools necessary to analyze equilibria and formalize strategies in gameplay.

Our project, an unbeatable game of Tic Tac Toe, utilizes the Minimax algorithm to come up with an AI which will never lose against a human player (always win or tie). The user will interact with the computer through a graphical GUI, alternating moves with the computer until the game is over. Time permitting, we would like to implement additional features that include an optimization of the Minimax algorithm using Alpha-Beta pruning, and increased interaction with the user (e.g. hint provider, scorekeeper, etc.)

Signatures/Interfaces

To help us begin thinking about signatures/interfaces, we used http://www.leaseweblabs.com/2013/12/python-tictactoe-tk-minimax-ai/ as inspiration for getting started. We referenced this source to see what data and class structure they used for the Board, as well as to see one example of an implementation for the Minimax algorithm. Other sources we looked at include https://gist.github.com/SudhagarS/3942029, http://cwoebker.com/posts/tic-tac-toe, and http://neverstopbuilding.com/minimax.

After reviewing these different sources and speaking with our TF, we decided that we wanted to have a board class, keeping track of the positions of the computer (computer) and the human user (opponent). However, we plan to write methods in a class separate from the board that will check the conditions of a potential move selection. We did include a separate class but this only contained Minimax and the function allowing the player and AI to make a move. These separate basic functions would take information passed from the board class and return boolean values analyzing a given potential move. The benefits of this organization include easier modularization to equally divide work between members, and a more efficient algorithm execution since we will not need to constantly change the board class itself to analyze every new possible configuration with each move made.

Currently, we have a partial implementation for the Board class, helper methods and minimax algorithm, and our next steps are to further flesh out the Board class and its methods as well as begin working on the GUI (user interface) via command-line. We implemented this code before our meeting with Ian, and plan to follow his suggestions to make the code more efficient and better modularized by making use of an additional class separate from the board that contains our board analysis functions.

Some examples of Board class methods:

- def __init__(self): initializes the board with attributes size (3x3), positions (a 2D array of all possible spots on the board), empty (character to represent an empty spot), player ('X'), and opponent ('O')
- def board_to_string(self): returns string representation of Board that can be printed out to the command line for debugging purposes
- def bestMove(board, player): takes in a board and player and uses Minimax algorithm to determine the best next move for player. Minimax was implemented in the helper functions class.

Some examples of Analyze class methods: We didn't include an Analyze class — the below functions were moved to the board class.

- def isWinningMove(board, i, j): checks to see if the proposed move is a winning move and ends the game if it is
- def isTaken(board, i, j): checks to see if a position is already taken or not
- def isWonBy(board, player): takes in a board and a player returns the winning configuration, if game has been won by the player, otherwise returns None
- def isTied(board): takes in a board and returns True if board has been completely filled and no one has won, False otherwise

Some examples of helper functions:

- def make_move(board, x, y, player): takes in a board, a pair of x-y coordinates and a player marker ('X' or 'O'), and returns the updated board with that spot filled.
- def open_positions(board): takes in a board and returns all currently empty spaces Moved to board class.
- def get_positions_of(board, player): takes in a board and player and returns all spaces occupied by that player Moved to board class.
- def enemy_of(player): returns 'X' if player is 'O' and vice versa. Moved to board class.

Modules/Actual Code

Please see the attached file tictactoe.py, which contains detailed comments that elaborate on the thoughts mentioned in the previous section.

Timeline

Week 1 (ends 4/24)

- Have implemented full version of tic-tac-toe
  - utilizing Mini-Max algorithm
- By 4/19
  - Have Board class completely implemented
  - Able to see board positions taken by different players
  - Able to chose your next position
- By 4/22
  - Finish coding helper methods and minimax algorithm
- By 4/24
  - optimize code by following Ian's suggestions to make code more efficient
  - determine modularization for implementation of more games for next week

Week 2 (ends May 1) As mentioned before, none of these functionalities were implemented

- implement Alpha-Beta pruning optimization
- implement GUI for user interface
- Implement Connect 4

Progress Report

To date, we have set up a Github repository that everyone in the group has access to. We will be programming in the Virtual Machine to standardize any packages we download, not only for ourselves but also our TF. We have also begun implementing the Board class, Analyze class, and helper functions, which have been thoroughly commented in the file tictactoe.py.

Version Control

Our shared GitHub repository is located at https://github.com/hansss/tictactoe.