P2: Monte Carlo Tree Search for Ultimate Tic-Tac-Toe

Introduction

In this programming assignment you will need to implement, in Python, a bot that plays Ultimate Tic-Tac-Toe using Monte Carlo Tree Search (MCTS). Ultimate Tic-Tac-Toe is a turn-based two-player game where players have to play a grid of 9 tic-tac-toe games simultaneously in order to complete one giant row, column, or diagonal. The catch is that each on each player's turn, they can only place a cross or circle in *one* square of *one* board; and whichever square they pick, their opponent must play *on the corresponding (possibly different) board*. The bots will be implemented using a digital version of this game implemented in python, which is described in detail in the next sections. In order to evaluate your implementation, you will also need to perform and analyze two different experiments.

Example

In order to run the game interactively you need to execute the following command (assuming you are in the /src folder):

```
$ python p2_play.py human human
### actually p2_play.py PLAYER1 PLAYER2, where either can be 'human',
'mcts_vanilla', 'mcts_modified', 'random_bot', 'rollout_bot'.
```

When you run this code you should see a textual display of the board and a prompt for which board and which square to move in. Figure 1 shows an example of a game's first few moves:

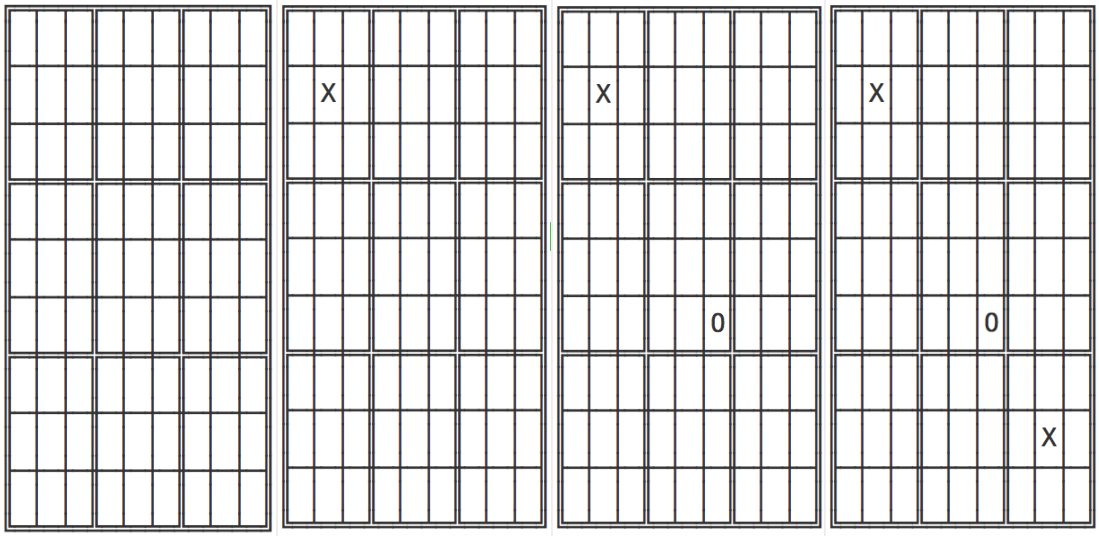

Figure 1: Game state after three moves

The game can also be executed in a simulation mode, where there is no rendering. This is useful to test how one bot plays against another. To execute this type of simulation, you can run the following command:

```
$ python p2_sim.py PLAYER1 PLAYER2
```

Base Code Overview

- **p2_sim.py**
  A multiple-game simulator useful for running repeated rounds between pairs of bots. To change which bots are active in either the graphical or tournament versions of the game, pass different command-line arguments. (python p2_sim.py botname1 botname2)
- **p2_play.py**
  An interactive version of the game. Bots are given as above (python p2_play.py botname1 botname2).
- **mcts_vanilla.py and mcts_modified.py**
  These are the files where you will implement your MCTS bot. In your bot modules, implement a function called "think" that takes a board (game setup, see p2_t3.py's Board class) and an immutable state, which represents the current state of the game. Here is the allowed interface:

  - `board.legal_actions(state)` → returns the moves available in state.
  - `board.next_state(state, action)` → returns a new state constructed by applying action in state.
  - `board.is_ended(state)` → returns True if the game has ended in state and False otherwise.
  - `board.current_player(state)` → returns the index of the current player in state.
  - `board.points_values(state)` → returns a dictionary of the score for each player (eg {1:-1,2:1} for a second-player win). Will return None if the game is not ended.
  - `board.owned_boxes(state)` → returns a dict with (Row,Column) keys; values indicate for each box whether player 1, 2, or 0 (neither) owns that box
  - `board.display(state)` → returns a string representation of the board state.
  - `board.display_action(action)` → returns a string representation of the game action.

  Think should call the appropriate functions for the stages of MCTS (which you will also implement):
  - `traverse_nodes` → a.k.a. 'selection'; navigates the tree node
  - `expand_leaf` → adding a new MCTSNode to the tree
  - `rollout` → simulating the remainder of the game

- `backpropagate` → update all nodes along the path visited

Your game tree should be constructed of MCTSNodes (see mcts_node.py). The class acts as a straightforward object containing the appropriate information:

- `parent` → the parent of the node
- `parent_action` → the action taken to transition from the parent to the current node
- `child_nodes` → a dictionary mapping an action to a child node
- `untried_actions` → a list of actions that have not been tried or
- `node.child_nodes.keys()` → a list of actions that have been tried
- `wins` → the number of wins for all games from the node onward
- `visits` → the number of times the node was visited during simulated playouts
- `tree_to_string(horizon)` → a function which will return a string representation of the game tree to a specified horizon.
  Ex: `print(node.tree_to_string(horizon=3))`

Supplied example bots:

- `random_bot.py` → selects a random action every turn
- `rollout_bot.py` → for every possible move from the immediate state, this bot samples x games (played randomly) and returns the move with the highest expected turnout

NOTES: Adversarial planning – the bot will be simulating *both* players' turns. This requires you to alter the UCT function (during the tree traversal/selection phase) on the opponent's turn. Remember: the opponent's win rate (Xj) = (1 – bot's win rate).

Requirements

- Implement mcts_vanilla.py that uses MCTS with a full, random rollout.
- Using your existing implementation from `mcts_vanilla.py` as base code, implement `mcts_modified.py` with the addition of your own heuristic rollout strategy as an improvement over vanilla MCTS. Optional: You may also adjust other aspects of the tree search, by implementing the variations discussed in class (roulette selection, partial expansion, etc).
- Perform the two experiments described below.

Evaluation

Below are two experiments we have outlined for the assignment. If it appears that your test cases require too much time, decrease the size of your game trees first, then drop the number of test games. Note this in your analyses. Also, describe how lowering these quantities affects the confidence of your conclusions.

Bonus points if you parallelize your simulations or the different parameterizations for your experiment (if you do this, show the difference in wallclock time between the serial and parallel versions).

Experiment 1 – Tree Size
You are going to pit two versions of the vanilla MCTS bot against each other. Player 1 will be fixed at 100 nodes/tree. Test at least four sizes of tree for Player 2 for at least 100 games each (use p2_sim.py). Plot the number of wins for each tree size. Describe your result (on the submission form). Include an image of your plot with your submission.

Experiment 2 – Heuristic Improvement
Next, have your modified version of MCTS play against the vanilla version with both having equal tree sizes (suggested size: 1000). Submission analysis: Does the modified version win more games? Does this change if you increase or decrease the size of the trees?

Extra Credit (Optional):
Experiment 3 – Time as a Constraint
Rather than fix the tree size, use time as the limiting factor. Set a time budget of 1 second. Alter your code to continue growing the tree as long as one full second has not passed. Test this constraint on both implementations. Does your modified version have a larger or smaller tree than the vanilla version? Why do you think this is the case? Does this comparison change at various time limits?

Helper timing code:
```
from timeit import default_timer as time
start = time()                 # Should return 0.0 and start the clock.
time_elapsed = time() - start  # This is in seconds.
```

Grading Criteria

- Completion of MCTS implementation
- Analysis of Experiment 1
- Analysis of Experiment 2

Submission Instructions

Submit a zip file named in the form of "Lastname-Firstname.zip" containing:

- A README naming the teammates and explaining the modifications done for mcts_modified.py.
- `mcts_vanilla.py` file implementing the MCTS functions.
- The `mcts_modified.py` file implementing the MCTS functions with the addition of your own heuristic rollout strategy.
- `experiment1.txt` with the analysis of the results of the Experiment 1.

- `experiment2.txt` with the analysis of the results of the Experiment 2.

References

Browse Cameron Browne's MCTS lecture slides:
http://ccg.doc.gold.ac.uk/ccg_old/teaching/ludic_computing/ludic16.pdf
See the survey paper from TCIAIG as well: http://diego-perez.net/papers/MCTSSurvey.pdf