P3: Dots and Boxes

## Introduction

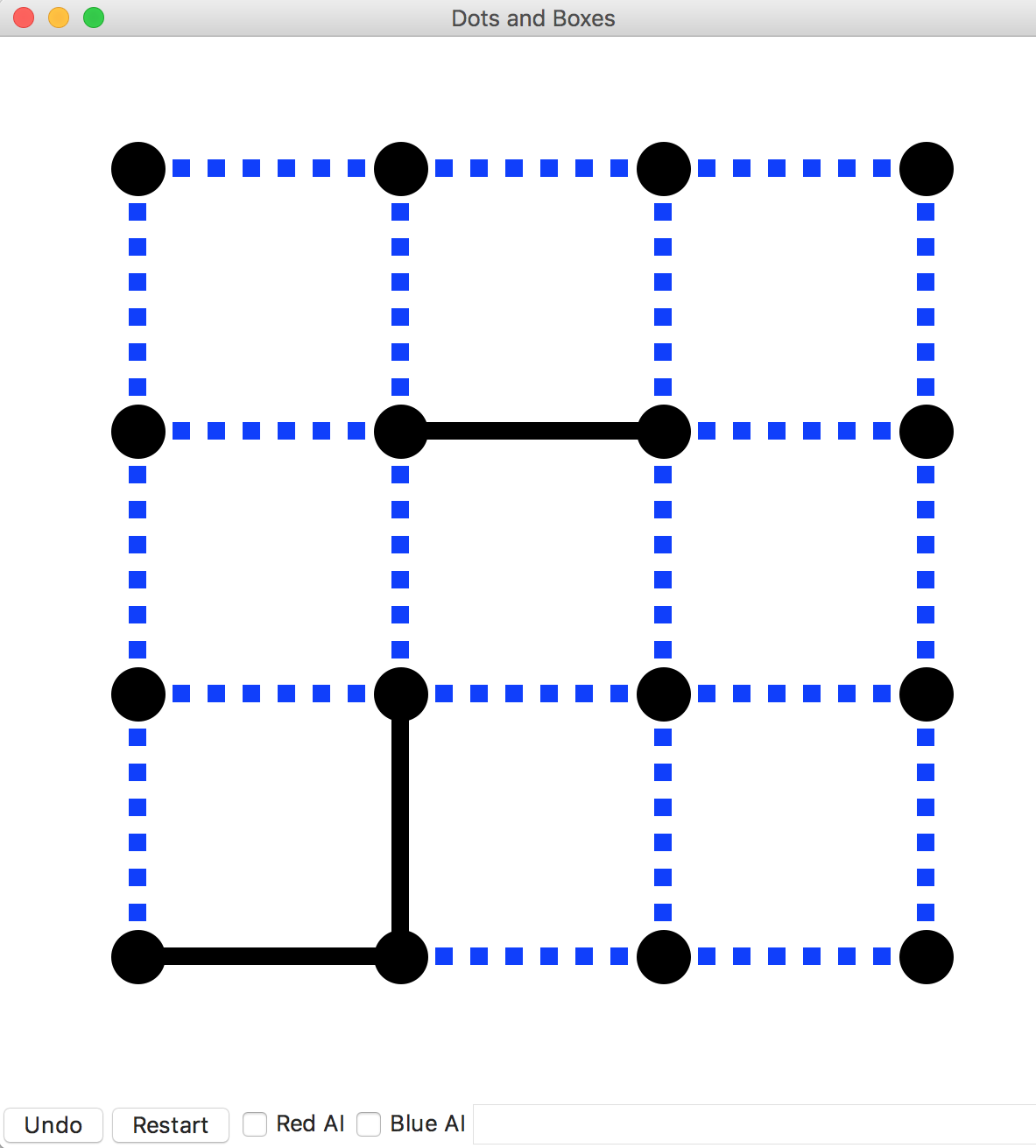
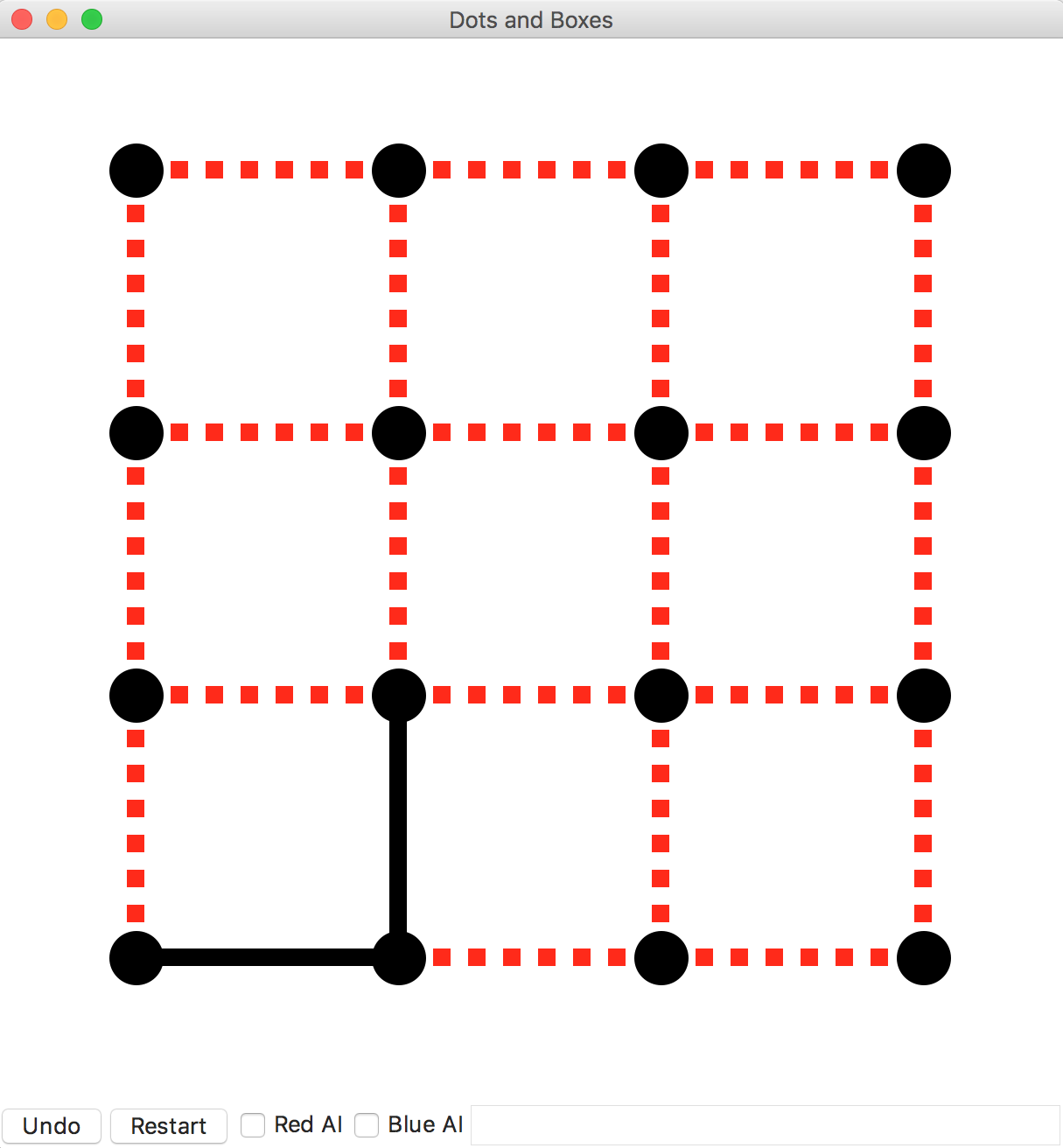
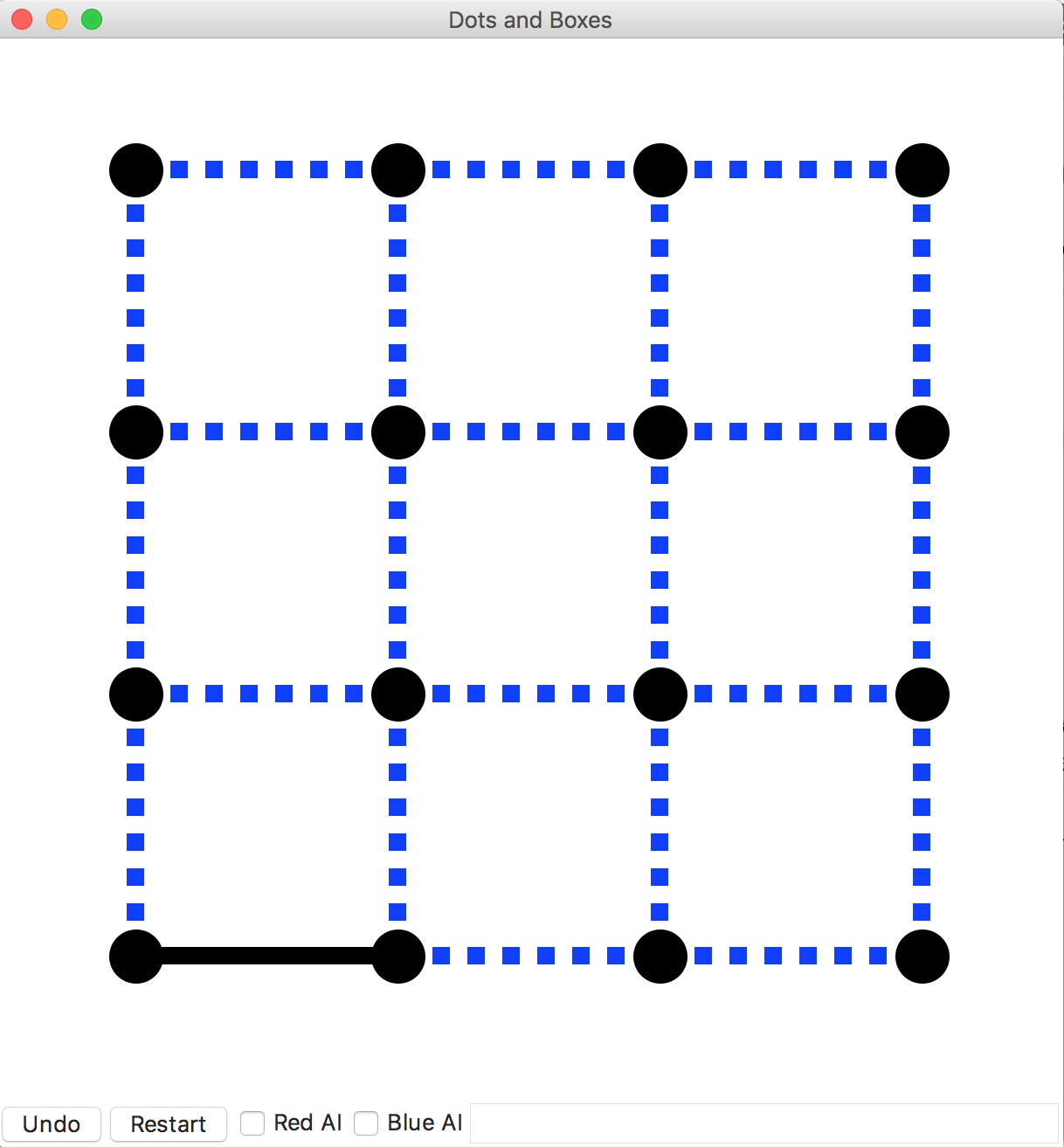
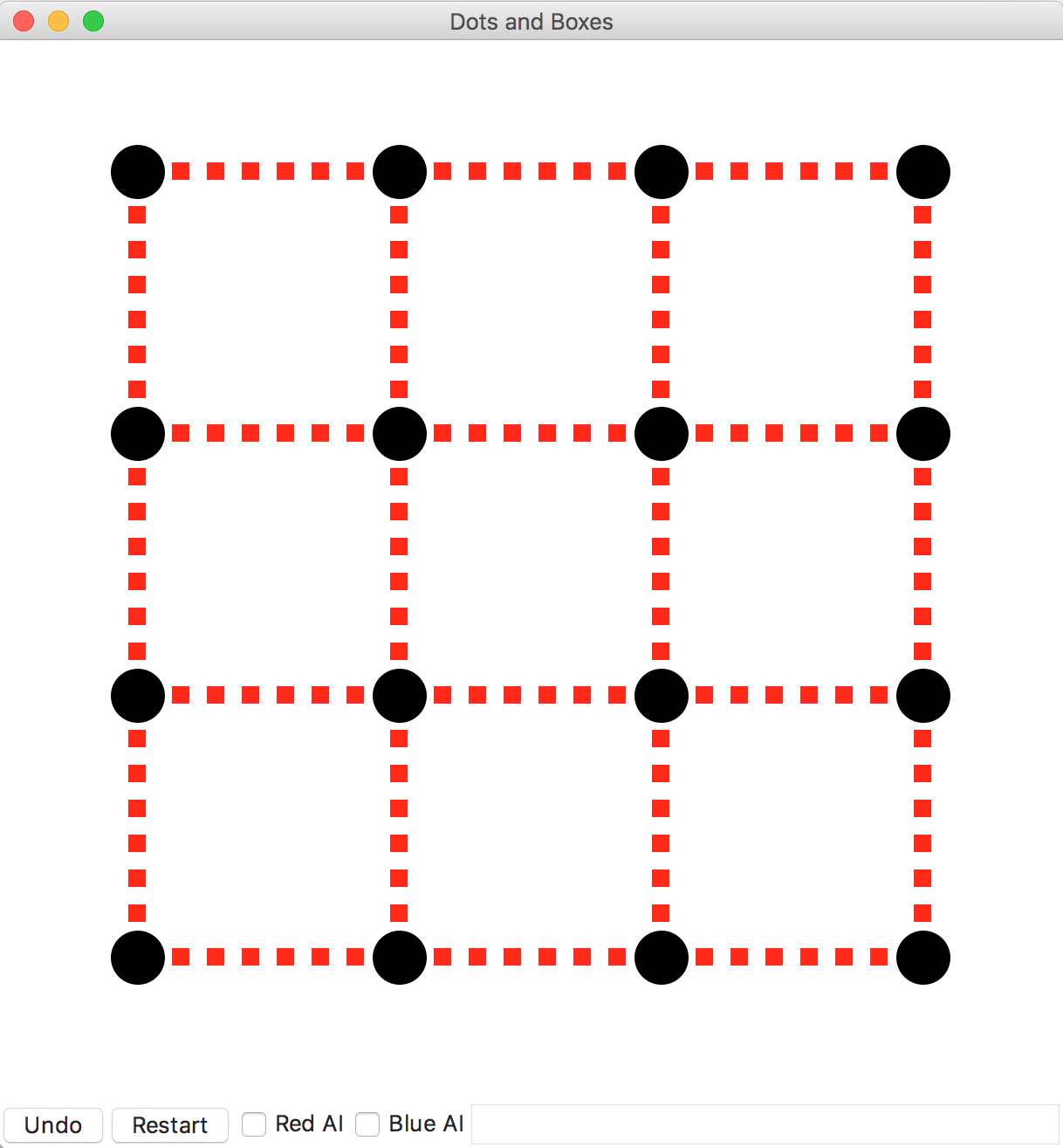
In this programming assignment you will need to implement, in Python, a bot that plays the Dots and Boxes game using Monte Carlo Tree Search (MCTS). Dots and Boxes is a turn-based two-player game where players have to earn as many points as they can by drawing lines on a grid to form boxes. Starting with an empty grid of dots, two players take turns adding a single horizontal or vertical line between two unjoined adjacent dots. The player who completes the fourth side of a 1×1 box earns one point and takes another turn. 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 analyse two different experiments.

## Example

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

*$ python3.5 p3\_gui.py*

When you run this code you should see a new window with the initial state of the board: only dashed lines colored in red. The color here represents the player, where red is player 1 and blue is player 2. You can interact with this window by clicking on the dashed lines. If you click on any one of them, they become full lines and the color is changed (indicating the next player has to make a move). Figure 1 shows an example of a game state after three moves:



*Figure 1: Game state after three modes*

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:

*$ python3.5 p3\_sim.py*

## Base Code Overview

* **P3\_sim.py**

A text output only 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, edit the lines that look like this:

* + import first\_bot as red\_bot
* Here’s how you’d have the vanilla mcts play the blue player
  + import mcts\_vanilla as blue bot
* **P3\_gui.py**

An interactive, graphical version of the game. You may import bots as above. Bots are disabled by default, but you can turn them on with checkboxes and then restart the game.

* **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 state, which represents the current state of the game (an instance of State from p3\_game.py). Here is the allowed interface:

* state.apply\_move(move) → updates state to apply a move, assuming it was legal.
* state.copy() → produces a copy of this state that can be mutated (with apply\_move) without changing the original.
* state.is\_terminal → returns True if the game has ended and False otherwise.
* state.score → returns a dictionary of the score (eg {‘red’:3,’blue’:1}).
* state.winner → returns the name of the winning player or ‘tie’ if the scores are tied

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.

Experiment 1 – Tree Size

You are going to pit two versions of the vanilla MCTS bot against each other. Blue will be fixed at 100 nodes/tree. Test at least four sizes of tree for Red for at least 100 games each (use p3\_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 “Lastname1-Lastname2-P3.zip” containing:

* 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.

[Submission Link](https://goo.gl/fSAoOQ)

## References

Browse Cameron Browne’s MCTS lecture slides: <http://ccg.doc.gold.ac.uk/teaching/ludic_computing/ludic16.pdf>