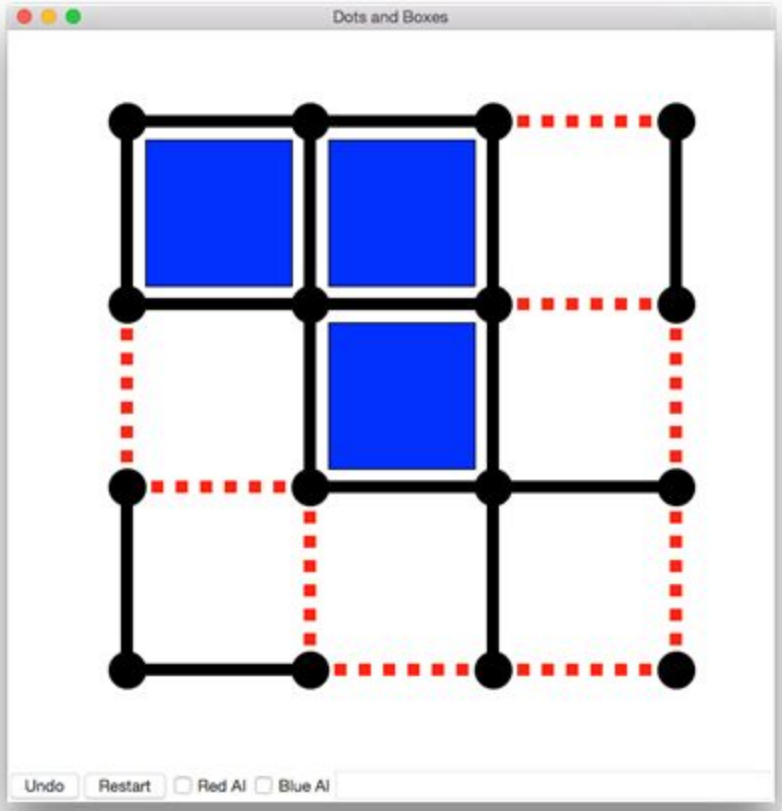
P2: Dots and Boxes



Objectives

- Understand how fixed programs can play arbitrary games by exploring a game tree.
- Understand how to explore a game tree incrementally.
- Understand the UCT algorithm.
- Familiarize yourself with running graphical Python programs.

Programming Requirements

· Implement mcts_vanilla.py that uses MCTS with a full, random rollout.

· Use your existing implementation from mcts_vanilla.py to implement mcts_modified.py, then improve the “modified” version by creating your own heuristic rollout strategy.

· Optional: You may also adjust other aspects of the tree search, by implementing the variations discussed in class (roulette selection, partial expansion, etc).

Helpful Resources

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

Base Code

python p2_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

p2_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.

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

Your bots:

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 p2_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 → (no parentheses!) returns True if the game has ended and False otherwise

· state.score → (no parentheses) returns a dictionary of the score (eg {'red':3,'blue':1})

· state.winner → (no parentheses) 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 nodes

· 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
    - actions that have been tried can be retrieved with node.child_nodes.keys()
- 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)

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

Evaluation

Preface: 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 over 100 games each (use p2_sim.py). Plot the number of wins for each tree size. Describe your result (on the submission form). Include 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. 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 (roughly equal weight)

- Miscellaneous homework companion questions on submission form
- Completion of MCTS implementation
- Analysis of Experiment 1
- Analysis of Experiment 2

Submission Link