CS 4320 PA3: Game Playing with UCT
Final Deadline: Wednesday, Nov 26 at 11:59 PM

Groups: You are expected to work in a group of two or three for this assignment.

Turn in only one copy per group, clearly listing all team members. In your writeup you should including a brief discussion of what the contributions of each individual team member were towards the final submission. It is expected that each team member will contribute significantly towards the design, coding, testing, and documentation aspects of the project. Paired coding and similar approaches are encouraged. This is a project of significant complexity, and you should plan for it to take multiple programming sessions over a couple of weeks to get it completed.

Objective: To implement and experiment with Monte Carlo Tree Search as a method for planning/strategic reasoning, using Connect Four as a sample domain.

AI Agents for Game Playing

In this assignment you will be developing AI agents to play the game of Connect Four based on the MCTS approach we have discussed in class. The game of Connect Four is played on a grid with 7 columns and 6 rows, and the basic goal of each player is to get four of their game pieces in a horizontal, vertical, or diagonal line. Please see https://en.wikipedia.org/wiki/Connect_Four for more details on the rules and gameplay. In the first part you will develop and test algorithms for making move selections given a specific board state. In the second part you will test these algorithms against each other in actual game play.

Part I: Algorithms for Selecting Moves

In the first part you will implement three algorithms that build on each other for selecting moves for given board configurations. Your code must read a game board from a file given in a specific format and run an algorithm with given parameters. The expected output of your algorithms is described below. The board is specified in a standard text file; you do not need to check for the validity of the board (your code will only be tested on valid game states). An example of a game file is shown below. The first line specifies an algorithm to run. The second line specifies the player who will make the next move (R or Y). The next six lines represent the current configuration of the game board. We will use the colors Red and Yellow for the two players; their pieces are represented by the characters 'R' and 'Y' respectively. The character 'O' represents an open space. Moves are made by specifying a valid column from 1-7 to add a new piece to (columns that are already full are illegal moves). We will consider the Red player the "Min" player and represent a win for Red as a -1. The Yellow player is the "Max" player and a win for yellow is represented by a 1. A draw has a value of 0.

```
UR
R
OOOOOOO
OOOOOOO
OOYOOOY
OOROOOY
OYRYOYR
YRRYORR
```

Your code should run on the command line and take in three parameters. The first specifies the name of the input file to read. The second parameter specifies "Verbose" "Brief" or "None" which controls what your algorithm will print for output. The last parameter is specific to the algorithm (typically this will be the number of simulations to run).

python test1.txt Verbose 0
python test4.txt Brief 500

Algorithm 1: Uniform Random (UR)
This is a trivial algorithm used for basic testing and benchmarking. It selects a legal move for the specified player using the uniform random strategy (i.e., each legal move is selected with the same probability. The last parameter value should always be 0 for this algorithm. You should print only the move that is selected.

FINAL Move selected: 4

Algorithm 2: Pure Monte Carlo Game Search (PMCGS)
This algorithm is the simplest form of game tree search based on randomized rollouts. It is essentially the UCT algorithm without a sophisticated tree search policy. Please refer to https://en.wikipedia.org/wiki/Monte_Carlo_tree_search for detailed descriptions and examples for both PMCGS and UCT (of course you can also use the course slides, textbook, and other references as needed). The main steps in this algorithm are the same as in UCT, but every move both within the tree search and the rollout is made at random. Output the value for each of the immediate next moves (with Null for illegal moves) and the move selected at the end. Only if the "Verbose" mode is selected you should also print out additional information during each simulation trace, as shown below. For each node in the search tree output the current values of wi and ni, and the move selected. When you reach a leaf in the current tree and add a new node print "NODE ADDED". For the rollout print only the moves selected, and when you reach a terminal node print the value as "TERMINAL NODE VALUE: X" where X is -1, 0, or 1. Then print the updated values. The last parameter is the number of simulations to run for this algorithm.

wi: 187
ni: 456
Move selected: 2

```
wi: 67
ni: 178
Move selected: 6

wi: 18
ni: 62
Move selected: 7
NODE ADDED

Move selected: 3
Move selected: 1
Move selected: 5
TERMINAL NODE VALUE: -1

Updated values:
wi: -1
ni: 1

Updated values:
wi: 17
ni: 63

Updated values:
wi: 66
ni: 179

Updated values:
wi: 186
ni: 457

Column 1: 0.32
Column 2: 0.12
Column 3: -0.54
Column 4: 0.24
Column 5: Null
Column 6: 0.27
Column 7: -0.31
FINAL Move selected: 4
```

Algorithm 3: Upper Confidence bound for Trees (UCT)

The final algorithm builds on PMCGS and uses most of the same structure. The only difference is in how nodes are selected within the existing search tree; instead of selecting randomly the nodes are selected using the Upper Confidence Bounds (UCB) algorithm. For any node that is NOT a leaf (i.e., all possible children are already in the tree), calculate the UCB value for all children, and pick the one with the highest (or lowest) value depending on which player is choosing a move. The output should be mostly the same as for PMCGS, but adds the UCB values computed for the children before specifying the move that is selected for the tree search part of the simulation when in "Verbose" mode. Note that the final values printed for the children of the root and the final move selection are NOT based on the UCB equation, but the direct estimate of the node value (i.e., just wi/ni). The last parameter is the number of simulations to run for this algorithm.

```
wi: 143
ni: 495
V1: 0.11
V2: -0.34
V3: 0.42
V4: 0.31
V5: -0.24
V6: 0.49
V7: 0.37
Move selected: 6

wi: 45
ni: 131
V1: 0.21
V2: 0.39
V3: -0.32
V4: 0.12
V5: -0.41
V6: 0.23
V7: -0.28
Move selected: 5

wi: 24
ni: 52
V1: -0.17
V2: -0.55
V3:  0.37
V4: 0.24
V5: -0.33
V6: -0.29
V7: 0.42
Move selected: 7
NODE ADDED

Move selected: 4
Move selected: 2
Move selected: 6
TERMINAL NODE VALUE: 0

Updated values:
wi: 0
ni: 1

Updated values:
wi: 24
ni: 53

Updated values:
wi: 45
ni: 132

Updated values:
wi: 143
ni: 496

Column 1: 0.32
Column 2: 0.12
Column 3: -0.54
Column 4: 0.24
Column 5: Null
Column 6: 0.27
Column 7: -0.31
FINAL Move selected: 1
```

Implementation Notes

You will need to implement a method that checks for whether a game state is a terminal state (win for either player or a draw). You are welcome to use/adapt code that you find elsewhere for doing this, as long as you cite (in the code) where you got the code from and check that it works correctly. You will also need to keep track of game nodes in the search tree; you may use any tree library you like to support this functionality. ***You should not need to store the full game board/state in search tree nodes, only the wi and ni values.*** You can keep track of the game state during the search process by modifying the current board state based on the moves that are made.

Part II: Algorithm Tournaments and Evaluation

The second part builds on what you have done to develop, test, and debug algorithms in part I. You should not need to write much additional code for this part. You will set up you program so that it can play full games of Connect Four using combinations of the algorithms you developed in part I, and you will use this capability to conduct some experiments to test the strength of the algorithms. Since you already have methods to select moves, all you need to do is set it up to start from the initial (empty) state and alternate calls to select moves for the two players, update the board state for each move selected, and run the test to see if you have reached a goal state. To maximize the speed of the simulations the "none" setting for the output so the algorithms do not print out anything during game play.

You will run an experiment to test five variations of your algorithms against each other, as listed below. The numbers after the algorithm acronym are the parameter settings. Run a round-robin tournament where you run each combination of the five against each other for 100 games, recording the number of wins for each algorithm. There should be 25 combinations in total. Present a table in your final report of the results, showing the winning percentage for the algorithm specified on the row vs the algorithm specified on the column.

1) UR
2) PMCGS (500)
3) PMCGS (10000)
4) UCT (500)
5) UCT (10000)

What to Turn In

You may implement your program in either Java or Python (or ask me if you prefer another language). You must turn in both the source code and an executable file. Your code must run on the command line as specified for Part I.

In addition to the code, turn in a short report documenting:
1) Group member contributions
2) Your results from testing the algorithms in part II

<u>Additional Notes/Tips</u>

Some students have reported performance challenges when scaling up to larger numbers of simulations (e.g., 10000). If you are not able to run these results in a reasonable time, as a last resort you can scale down the number of simulations (say to 1000) and note this in your report. However, you should first do some basic performance debugging of your code. The most likely culprit is inefficient memory use. You do NOT need to create and store new copies of the game board for every node in the tree, or even as you are running the simulations. You can use the same object to track the state, and create simple do/undo functions for executing a move. The only things you need to keep in the game tree are the variables needed for tracking wins and simulations. A second issue could be inefficiency in checking for wins/terminal conditions. Note that any winning line must include the last move, which significantly limits the number of checks you need to do (you do not need to examine the full board each time). Also make sure that you are not printing out any information when running large simulations; that will also dramatically slow down your code.

Debugging code for these types of algorithms can be challenging because there are many subtle things that can lead to problems. Think carefully about how you test. Do basic sanity checks (e.g, are all of the values being computed in the range of -1 to 1? Does the size of the tree grow at the right rate? Are the numbers in the nodes updated correctly after a single simulation?). Create some test boards with moves that are obviously good/bad for a player and make sure you get numbers/results that make sense. For fixed board states are you getting consistent values for the different moves? Do the different algorithms agree?

In general, you should see improvement for the MCTS algorithms against UR as you increase the number of simulations, but it may require a fairly large number to really see clear results (for debugging you may try even larger numbers of simulations to get a better sense of the effects). The exact performance depends on implementation details and where you are in the game; typically you will see stronger results/differences in the midgame sooner (due to fewer remaining possible game paths). Things like changing the exploration constant can also have an effect. Make sure that you are picking final moves to actually play based only on the expected performance of those nodes (exploration is not a factor at this point).