CS 4320/5314
Programming Assignment 2: Game Playing Agents
DUE: Sun, Nov 12 at 11:59 PM

Groups: You are encouraged to work in a group of two for this assignment; due to differences in requirements it is preferable to have homogeneous groups of either graduate or undergraduate students, but this is not required. If you need to work in a group of three please get prior approval (this should be rare).

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 some core algorithms for game tree search, including minmax and Monte Carlo Tree Search.

AI Agents for Game Playing

In this assignment you will be developing AI agents to play the game of Connect Four based on some core algorithms 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 implement four algorithms for selecting moves for given board configurations. Your code must read a game board from a file given in a specific format and run a specified algorithm with a specified parameter setting. 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 two lines specify an algorithm to run and a parameter for the algorithm. The third 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
0
R
OOOOOOO
OOOOOOO
OOYOOOY
OOROOOY
OYRYOYR
YRRYORR
```

Your code should run on the command line and take in two parameters. The first specifies the name of the input file to read. The second parameter specifies "Verbose" "Brief" or "None" which control what your algorithm will print for output.

```
python test1.txt Verbose
python test4.txt Brief
```

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 parameter value should always be 0 for this algorithm. You should print only the move that is selected.

```
FINAL Move selected: 4
```

Algorithm 2: Depth-Limited MinMax (DLMM)
This algorithm uses depth-first minmax search out to a specified maximum depth to select a move. You should test each node to see whether it is a terminal state (i.e., a win for one player or a draw); if so return that value immediately. If you reach the depth limit without reaching a terminal state, apply a heuristic evaluation function to the game state and return that value. You may choose to implement any reasonable heuristic; you will give a brief description in your report. The heuristic should take in any valid (non-terminal) board state as input and return a predicted value for that state between -1 and 1. For example, you may want to count the number of lines of 2 or 3 pieces that can be extended for each player. If there are ties, break them randomly. Make sure you keep track of which player (maximizing or minimizing) is making the choice at each node. Output the value for each of the immediate next moves from the root (with Null for illegal moves) and the final move selected, as shown below.

```
Column 1: Null
Column 2: 0
Column 3: 0.67
Column 4: 0.12
Column 5: -0.23
Column 6: 0
Column 7: 0.55
FINAL Move selected: 3
```

Algorithm 3: 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.

```
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 4: 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).

```
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

<u>Implementation Notes</u>

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 six 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 six against each other for 100 games, recording the number of wins for each algorithm. There should be 36 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) DLMM (5)
3) PMCGS (500)
4) PMCGS (10000)
5) UCT (500)
6) UCT (10000)

Part III: Enhancements

Graduate student group must implement at least one of the following three enhancements. Each additional one is worth up to 10 points of extra credit. For groups with *only* undergraduates, all of these are worth up to 10 points of extra credit.

1) Modify your code so you can play your AI algorithms against a human player. Have your team members play at least 5 games each against algorithms 2, 4, and 6 and record your results; comment briefly on your experience in the report.

2) Develop and test at least one significant improvement to the basic UCT algorithm (e.g., adding heuristics to guide the initial search). Do experiments to demonstrate that it improves significantly over the baseline version for the same number of simulations.

3) Develop and test at least one enhancement for the baseline depth-limited MinMax search (e.g., an improved/more sophisticated evaluation function). Do experiments to demonstrate that it improves significantly over the baseline version of the algorithm.

What to Turn In

You may implement your program in either Java or Python (or ask me if you prefer another language) and turn in both the source code and an executable file. Your code must run on the command line as specified for Part I. You may write the code to run the tournaments and anything necessary for Part III as a special case in your code (it does not need to run on the command line).

In addition to the code, turn in a short report documenting:
1) Group member contributions
2) What heuristic you implemented for DLMM (briefly)
3) Your results from testing the algorithms in part II
4) Anything necessary for Part III