COMS W4701: Artificial Intelligence, Summer 2024

Homework 2

Instructions: Compile all written solutions for this assignment in a single, typed PDF file. Coding solutions may be directly implemented in the provided Python file(s). **Do not modify any filenames or code outside of the indicated sections.** Submit all files on Gradescope in the appropriate assignment bins, and make sure to **tag all pages for written problems**. Please be mindful of the deadline and late policy, as well as our policies on citations and academic honesty.

Problem 1: Game Tree Search (24 points)

Two agents are playing a game using the following scoreboard. Player 1 (P1) first eliminates a column from the board. Player 2 (P2) next keeps a row from the current board. Finally, P1 selects one of the two values remaining as the game score. P1's objective is to minimize the score, while P2's objective is to maximize it.

6	4	7
5	1	2
3	8	9

1. (8 pts) Draw the full game tree, clearly indicating MAX nodes, MIN nodes, and terminal nodes and their values. Please order the nodes in each ply corresponding to rows going from top to bottom or columns going left to right. Finally, label all MAX and MIN nodes with their minimax values. What is the optimal sequence of actions taken by each player?

2. (8 pts) We perform alpha-beta search on the game tree following a depth-first, left to right order. Identify all nodes that are pruned during the search procedure (you can refer to each as "node i in ply j", both starting from 1). For each pruned node, give the α and β values of their *parent* node, as well as the value comparison that resulted in the pruning.

3. (4 pts) Now suppose that P2 plays randomly rather than optimally. Regardless of what P1 does, P2 keeps the first row with probability 25%, second row with probability 50%, and third row with probability 25%. Compute the new expected score of the game (show your calculations). Briefly explain why P1 would never change their initial strategy regardless of P2's row selection probabilities.

4. (4 pts) Now suppose that rather than using a full search, we use an evaluation function to estimate the game score immediately after P1 eliminates a column. We consider two such functions: a) Minimum of remaining matrix values, and b) Average of remaining matrix values. Compute the expected game score and best action for P1 for each function.

Problem 2: MDPs and Dynamic Programming (24 points)

A mobile robot is moving around on a rechargeable battery. There are three battery level states: *high*, *low*, and *off*. In the first two states, the robot may move *fast* or *slow*, while in *off*, the robot may only *recharge*. Transitions are stochastic; moving *fast* is guaranteed to lower the robot's battery level, while moving *slow* may sometimes do so. A transition and reward function are defined for this robot as follows.

s	a	s'	$T(s,a,s')$	$R(s,a,s')$
high	*fast*	*low*	1.0	+3
high	*slow*	*high*	0.5	+2
high	*slow*	*low*	0.5	+2
low	*fast*	*off*	1.0	+2
low	*slow*	*low*	0.75	+2
low	*slow*	*off*	0.25	+1
off	*recharge*	*high*	1.0	−2

1. (6 pts) Consider the policy π in which the robot goes *fast* in both the *high* and *low* states and *recharges* in the *off* state. Write down the system of linear equations describing the value function V^π in terms of γ, and then solve for the values using $\gamma = 0.5$. (You don't need to do the last step by hand, but please state if you are using any programs to help with it.)

2. (6 pts) Suppose the values that you obtained above are the time-limited values V_i in iteration i of value iteration, which is being used here to find the optimal policy π^*. Show the computations done in the next iteration, and find the next set of time-limited values V_{i+1}.

3. (6 pts) We find that the optimal values are $V^*(high) = 4.42, V^*(low) = 2.84, V^*(off) = 0.21$ at convergence. Show the computations done by policy extraction to find π^*.

4. (6 pts) Now suppose $\gamma = 0$; find the optimal policy for this scenario. (You should be able to do so without resorting to dynamic programming.) Briefly explain how changing γ to 0 also changes any optimal actions from those you found in part 3 above with $\gamma = 0.5$.

Problem 3: Othello (52 points)

Othello, also known as Reversi, is a game in which two players take turns placing colored disks on a square grid. Take a look at some of the example board states on the linked Wikipedia page, and suppose it is the dark player's move. A valid move is one in which a new dark disk lies on the same line (horizontal, vertical, or diagonal) as another dark disk. In addition, there must be at least one opponent light disk and no empty spaces between the two dark disks. Once the disk is placed, all light disks that lie on *any* line between the new and an existing dark disk are "captured," or flipped over to dark. The light player plays similarly, and the objective of both players is to maximize the number of disks on the board corresponding to their color when the game ends. This occurs when either player has no legal moves left, even if empty spaces still exist.

You can run our implementation of the game using `python othello_gui.py`. (You may have to install `tkinter` first.) The `-b` option specifies the board size (default 4). The `-p1` and `-p2` options specify if each player will be controlled by an AI file. We provide `randy_ai.py` (Randy), which is essentially a random-move agent. Players that are not specified are controlled by a human. At this point, you can try out the following scenarios:

- Leave out the `-p1` and `-p2` options, which will allow you to play against yourself (or a friend). You can make a move by simply clicking on the cell in which you are placing a disk.
- Specify one of the players to be `randy_ai.py`, which will allow you to play against Randy.
- Specify both players to be `randy_ai.py`, which will show a game between Randy and itself.

3.1: Alpha-Beta Minimax Search (12 points)

Your first task is to implement alpha-beta minimax search in `minimax_ai.py`. In this file, `run_ai()` will call `minimax()`, which will then call one of `max_value()` or `min_value()`. You need to implement the latter two methods, which will recursively call each other until a terminal state is reached.

You can use the helper function `get_possible_moves()`; if this returns an empty list, then the given state is terminal, and the utility can be computed using `compute_utility()`. Otherwise, you should iterate through the given moves, find the successor state using `play_move()`, and perform recursion. Remember to switch the player each time you move down the game tree.

Once you are finished with this agent, you can test it against itself on a 4 by 4 board. Unfortunately, this is the extent to which we can use this agent, as the game trees of larger boards are prohibitively large for a complete search.

3.2: Monte Carlo Tree Search (32 points)

To get around this issue, we will next implement Monte Carlo tree search, specifically the four helper functions for MCTS in `mcts_ai.py`. You will see that these are repeatedly invoked in sequence by the `mcts()` function, which is used to determine the agent's move on its turn.

We provide a `Node` class, which will be used to define the nodes of the search tree. Each Node contains the corresponding board state (NumPy array), player (1 for dark, 2 for light), parent node, list of children nodes, node value, and N (number of rollouts). There is also a `get_child()` method, which returns the child node given a board state or `None` if the child node does not exist. Here are some hints for implementing the four helper functions:

- `select()` takes in the root node and α value for UCT calculation. If the current node has at least one successor not contained in its children list (or has no successors), then return the current node. Otherwise, repeatedly move to the successor with the highest UCT value (the exploitation term can be computed as $\frac{value}{N}$, similar to win rate).
- `expand()` attempts to expand the tree. It finds a successor of the given state that is currently not in `node.children`, creates a new leaf node, and adds it to `node.children`. It then returns the leaf node. If `node` has no successors, then it simply returns `node` back.
- `simulate()` runs a rollout starting from the given `node` using a random rollout policy. It then computes and returns the utility of the final state.
- `backprop()` backpropagates the computed utility from `node` back up to the root. First, we increment the current node's `N` value. The node's `value` update depends on the player. For the light player (2) we *increment* its value by `utility`, since their parent wants to *maximize* these values. Conversely, for the dark player (1) we *decrement* its value by `utility`.

Unit Testing

Since all four MCTS methods need to be correctly implemented before you can run the agent, it would be best to unit test each one separately. We provide a `mcts_tests.py` file, which manually constructs a small MCTS tree and tests whether each of the four individual methods yields the correct result. Please note that **passing** a test does **not** guarantee that you have the correct implementation, but **failing** a test almost certainly indicates that you have an incorrect implementation. You are encouraged to modify this file and create other test cases while debugging.

Once you are finished, you can have the MCTS agent play against yourself, Randy, minimax, or another MCTS agent. You may notice that `rollouts` and `alpha` are set to default values of 100 and 5, respectively, in `mcts()`; these should be sufficient for the most part (though you are welcome to adjust them). You will also see the MCTS agent slowing down on boards larger than 10×10; you can optionally try decreasing `rollouts` so that it makes each move more quickly, although you should avoid making it too small as it may impact overall performance.

3.3: Analysis (8 points)

1. Test two minimax agents against each other on a 4 by 4 board. What is the result? Explain whether the same result will occur every time you start a new game.
2. Test a minimax agent against Randy on a 4 by 4 board a few times, and also make sure to test both scenarios in which minimax is player 1 vs being player 2. Does minimax always win, or does it only do so if it is a specific player? Give a brief explanation for your observations.
3. Repeat the above experiments and analysis, but now test minimax against a MCTS agent instead of Randy.
4. Move up to a 6 by 6 board, and test MCTS against Randy a few times. How does MCTS generally perform if it is player 1? Player 2?

Coding Submissions

You can submit just the completed `minimax_ai.py` and `mcts_ai.py` files together under the HW2 Coding bin on Gradescope.