COMS W4701: Artificial Intelligence, Spring 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: Local Search (16 points)

We will use local search to solve the 4-queens problem. To simplify representation, a state will be represented by a set of 4 coordinate tuples, one for each queen. Following NumPy indexing, the top left cell has coordinates $(0, 0)$ and the bottom right cell has coordinates $(3, 3)$. So the example state shown below is $\{(0, 1), (1, 3), (2, 0), (3, 2)\}$.

We will define neighboring states as those where exactly one queen is moved to a different column. We will use the "min conflicts" evaluation function h, which counts the number of different queen pairs that are in the same row, column, or diagonal.

1. (3 pts) Consider the initial state with all queens in the left column, or $\{(0, 0), (1, 0), (2, 0), (3, 0)\}$. What is its h value? Remember to count *all* pairs of queens in conflict. Are there any neighboring states with the same or greater h value? What kind of feature would this state represent in the state space landscape?

2. (6 pts) Starting from the above initial state, trace the hill-"descent" procedure in which we repeatedly move to a neighboring state with the *lowest* h value among all neighbors, until we can improve no further. Write out the state and h value after each iteration.

3. (3 pts) Now consider the initial state $\{(0, 1), (1, 3), (2, 2), (3, 0)\}$. What is its h value? Are there any neighboring states with the same or lower h value? What kind of feature would this state represent in the state space landscape?

4. (4 pts) Starting from the above initial state and now allowing sideways moves, trace a hill-descent outcome (as in part 2) that results in a consistent solution in the fewest number

of iterations. Explain whether it is possible to never find a solution if we make no other changes, and how a simple limit on the number of sideways moves may or may not change this outcome.

Problem 2: 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 the 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 solving the game through 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 3: Simulated Annealing (24 points)

Sudoku is a logic-based number placement puzzle. The basic rules are as follows: Given a $n \times n$ grid (where n is a perfect square) and a set of pre-filled clue cells, fill in the remaining cells such that each row, column, and $\sqrt{n} \times \sqrt{n}$ "major" subgrid contains an instance of each number from 1 to n (inclusive). Classic sudoku has a 9×9 grid.

You will implement a sudoku solver using simulated annealing. A state is a 2D NumPy array with n of each of the numbers from 1 to n. We will only consider states in which every major subgrid is already consistent (a reasonable assumption, as this is trivial to assign). A transition occurs by swapping two non-clue numbers within a major subgrid. Finally, we can measure the number of "errors" for a given state by counting the number of missing values in each row and column.

Programming Task (12 points)

Implement the `simulated_annealing()` function in `sudoku.py`. You should use the provided `successors()` and `num_errors()` functions. In addition to the initial `board` and `clues` indices, `simulated_annealing()` will also take in several arguments relevant to the temperature schedule. Your temperature parameter `T` should be defined as

```
T = startT * (decay**iter)
```

where `iter` is the iteration number (starting from 0).

The search procedure should end if any one of the following conditions occurs: `T < tol`, the current state has no successors, or the current state is a solution (number of errors = 0). When any of these occurs, return the following two values: the current board state, and a list of integers containing the number of errors of each state encountered (including initial state) during search.

Analysis (12 points)

Your main goal will be to solve 9×9 sudoku puzzles with $c = 40$ clues. You can run `python sudoku.py` with the `-n` and `-c` options set to these values. Without setting any other parameters, the default values of `startT` and `decay` are 100 and 0.5, respectively. You will likely see the solver doing very poorly on most instances with these parameters.

1. Use the `-d` option to experiment with the `decay` value. Specifically, find a value such that when you set the option `-b 30` (batch of 30 runs), at least half of the instances have a final error of 0. Explain why the new value is able to improve the performance of simulated annealing. Also include two plots, one showing the error history of an individual search that successfully finds a solution, and one showing a final error histogram of a batch of 30 searches.

2. Keeping the `decay` value that you found above, experiment with the `startT` parameter using the `-s` option. Show three individual search result plots with this value set to 1, 10, and 100. Explain how the different values of `startT` affect the progression of the search.

3. Repeat the above three experiments on 30-batch runs. Show the resultant histograms of each, and again explain how the different values of `startT` affect (or do not affect) the overall performance of the searches.

Problem 4: Monte Carlo Tree Search (36 points)

Othello, also known as Reversi, is a game in which two players take turns placing colored disks on a square grid. 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 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. So 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 options to be `randy_ai.py`, which will allow you to play against Randy.
- Specify both options to be `randy_ai.py`, which will show a game between Randy and itself.

Programming Task

Your task is to implement Monte Carlo tree search (MCTS) to play Othello, which should easily outperform Randy and hopefully most people in this class. Specifically, you will need to complete 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's children list contains all possible successors of the board state, then repeatedly move to the one with the highest UCT value. You can obtain these successors using the imported `get_possible_moves()` and `play_move()` functions. If the current node has at least one successor not in the tree or is a terminal state, then it stops and returns the current node.
- `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 `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`. One way to do so is to simply execute a random move at each node until reaching a terminal state. It then computes and returns the utility of the final state (you can use `compute_utility()`).
- `backprop()` backpropagates the computed utility from `node` back up to the root. First, we increment the current node's `N` value. Next, the node's `value` update depends on the player's utility. For the light player (2) we can use `utility` directly, since their parent (1) wants to *maximize* these values. For the dark player (1), we need to use the *negative* of `utility` for the opposite reason. The new (average) value of each node can then be computed as follows:

```
node.value = (node.value * (node.N - 1) + player_utility) / node.N
```

Testing (no submission)

Since all four 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, 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). The MCTS agent should handily beat other non-MCTS agents (unless you are a Othello champion) on most boards. 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.

Coding Submissions

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