COMS W4701: Artificial Intelligence, Spring 2025

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: Simulated Annealing (12 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.

We provide a script `sudoku.py` that implements a sudoku solver using simulated annealing. The initial puzzle is a 2D NumPy array, where 0s indicate blanks and other values indicate clues. We initialize a starting state by replacing the 0s so that every major subgrid is consistent (a relatively easy task to perform). 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.

You will be analyzing the performance of simulated annealing on 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 failing to find a solution on most instances with these parameters.

1. (4 pts) Use the `-d` option to experiment with the `decay` rate of the temperature. 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 generate 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. It may be easiest to add some code to `sudoku.py` to do this.

2. (4 pts) Keeping the `decay` value that you found above, experiment with the `startT` (starting temperature) 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. (4 pts) 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 2: CSPs (14 points)

Now let's think about sudoku as a constraint satisfaction problem. We have a partially filled in 4×4 sudoku grid shown below. We name each of the empty cells X_i, where i starts from 1 and increases going across each row and then down the grid. So the two empty cells in the first row are X_1 and X_2, the second row has X_3 through X_6, and so on until the last variable X_{10}. Each variable currently has the domain $\{1, 2, 3, 4\}$.

	1	3	
3	4	2	1

1. (5 pts) Suppose we run **one round** of arc consistency, processing each variable once, starting from X_1. We do not re-add any constraints back into the queue after a variable's domain is pruned. Give the resultant domains for each of $X_1, ..., X_{10}$.

2. (5 pts) Now consider continuing arc consistency from the above point until it terminates. What are the resultant domains for each of $X_1, ..., X_{10}$?

3. (4 pts) Suppose we run backtracking search from this point onward, and it starts by making the assignment $X_1 = 2$. What happens if we run arc consistency again? Alternatively, suppose we do **not** run arc consistency and make a second assignment $X_6 = 4$. What happens in the next iteration of backtracking search?

Problem 3: Game Tree Search (24 points)

Two players are playing a misère tic-tac-toe game in which grid spaces have point values, shown on the left grid below. The players take turns marking a grid space with their own designation (X or O) until either a) one player gets three marks in a row or b) the board has no empty spaces left. When the game ends, a score is computed as the sum of the values of the O spaces **minus** the sum of the values of the X spaces. In addition, if O has three in a row, 3 points are added to the score; if X has three in a row, 3 points are subtracted from the score. X seeks to **maximize** the total score and O seeks to **minimize** it.

2	5	2
5	1	5
2	5	2

X		
X		O
O	O	X

1. (6 pts) The right grid shows the current board configuration. It is X's turn to move. Draw out the entire game tree with the root corresponding to the current board; be sure to draw the MAX and MIN nodes according to game tree convention. Label each node in the tree top-bottom, left-right using letters starting from A.

2. (6 pts) Sketch out the tic-tac-toe board configuration corresponding to each of the terminal nodes. Find the minimax value of all nodes (both terminal and nonterminal). What is the best move for X to make, and what is the expected score of the game assuming both players play optimally?

3. (3 pts) Suppose we perform alpha-beta search. Give an ordering of the tree nodes that would be processed such that **no** nodes are actually pruned. This ordering should include all nodes, starting from the root all the way to the last leaf node.

4. (3 pts) Given an ordering of the tree nodes that would be processed such that as many nodes are pruned as possible. Which nodes end up with incorrect minimax values as a result of pruning?

5. (3 pts) Suppose that player O plays stochastically. The probability of placing an O in cell j is $p_j = \frac{v_j}{\sum_i v_i}$, where $\sum_i v_i$ is the sum of all currently free cell values. For example, if there are two available cells with values 5 and 2, their probabilities are $\frac{5}{7}$ and $\frac{2}{7}$, respectively. Explain how the game tree will change (you do not have to redraw it), and compute the new value and best move for X starting from the current state.

6. (3 pts) Suppose that player X is considering playing either in the center or the upper right cells, and that player O will still play randomly following X in the latter case. Solve for the probabilities for player O making either of the two remaining moves so that X becomes **indifferent** about the two options on the first turn.

Problem 4: Othello (50 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 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.

4.1: Alpha-Beta Minimax Search (14 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.

4.2: Monte Carlo Tree Search (28 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.

4.3: 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?

Submission

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