COMS W4701: Artificial Intelligence, Summer 2022

Homework 2

Instructions: Compile all solutions to the written problems on this assignment in a single PDF file. **Show your work by writing down relevant equations or expressions, and/or by explaining any logic that you are using to bypass known equations**. Coding solutions may be directly implemented in the provided Python file(s). **Do not modify the filename or any code outside of the indicated sections.** When ready, follow the submission instructions to submit all files to Gradescope. Please be mindful of the deadline and our late policy, as well as our course policies on academic honesty.

Problem 1: Tic-Tac-Twist (22 points)

Two players are playing a modified 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 one player gets three marks in a row or the board has no empty spaces. When the game ends, a score is computed as the sum of the values of the X spaces **minus** the sum of the values of the O spaces. In addition, if X has three in a row, 3 points are added to the score; if O 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

(a) The right grid shows the current board configuration, whose current value is -3. It is O's turn to move. Draw out the entire game tree with the root corresponding to the current board. Use game tree convention to draw the MAX and MIN nodes. Also sketch out the tic-tac-toe board configuration for each terminal node (e.g., draw them right below each node).

(b) Compute the minimax values of each node. What is the best move for O to make, and what is the expected score of the game assuming both players play optimally?

(c) Suppose we are performing alpha-beta search. In what order would the successors of the root node have to be processed in order to maximize the number of nodes that can be pruned? Identify the node(s) in the game tree that can be pruned, and specify the α or β inequality that allows pruning to take place.

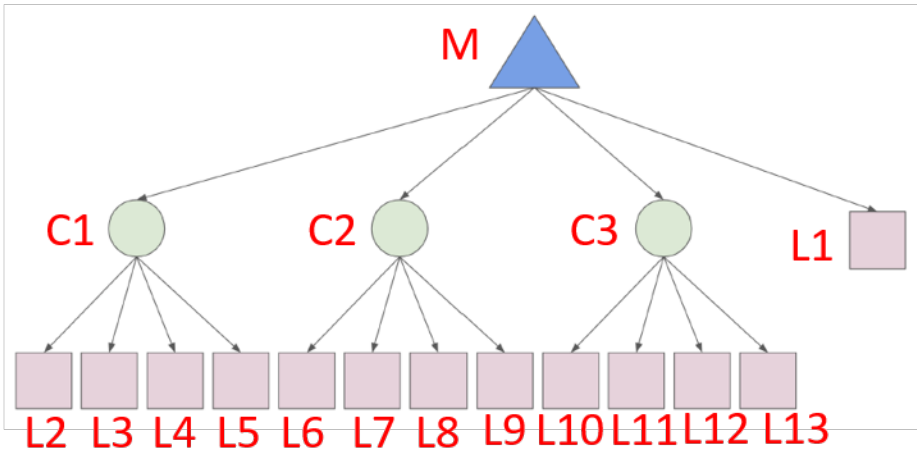
(d) Suppose that instead of playing to maximize the score, player X chooses a random valid move with uniform probability. Explain how the game tree will change (you do not have to redraw it), and compute the new utility and best move for O starting from the current state.

Problem 2: Yahtzee (18 points)

We will consider a simplified version of the game of *Yahtzee*. We first roll three 4-sided dice (with results 1, 2, 3, 4 occurring with equal probability), and then we can either reroll one of the dice or keep the original result. Let S be the sum of the final dice results. We get a score equal to the max over S and one of the following, if the situation applies:

- 10 points for two-of-a-kind
- 15 points for three-of-a-kind
- 7 points for a series (1-2-3 or 2-3-4)

The expectimax tree representing the player's decision is shown below:



Suppose we initially rolled a 3, 4, and 4. From left to right, the second layer of the tree represents actions of rerolling the 3, rerolling one 4, rerolling the other 4, and keeping the original dice results. The third layer represents the outcomes of each of the three die rerolls from 1 to 4 (going left to right for each).

(a) Fill in the values of the nodes of the tree (you can simply write out the values for each node variable, e.g. $M = 1$). Start with the utilities of the leaf nodes, followed by the chance nodes and finally the value at the root. What decision maximizes our expected score?

(b) Suppose we change the rules slightly. We can still either reroll a die or keep the initial results, but choosing the "reroll" action results in a **random** die being rerolled (we do not get to choose which). Describe precisely how the structure of the expectimax tree would change and say what the new optimal action is. You do not need to draw the new tree or compute new node values.

(c) Consider another change of rules. Instead of choosing one die to reroll, the options are to choose exactly **two** dice to reroll (we can indicate which two dice we want) or to keep the initial results. Describe precisely how the structure of the expectimax tree would change. You do not need to draw the new tree or compute new node values.

Problem 3: Sudoku (30 points)

Sudoku is a logic-based number placement puzzle. The basic rules are as follows: Given a $n^2 \times n^2$ grid and a set of pre-filled cells (*clues*), fill in the remaining cells such that each row, column, and $n \times n$ (non-overlapping) subgrid contains an instance of each number from 1 to n^2 (inclusive). Classic sudoku has $n = 3$ and a 9×9 grid. Like n-queens, we can define sudoku as a *constraint satisfaction problem*, which we can then solve using exhaustive or local search methods.

We are providing a simple Python scaffold for solving sudoku using hill-climbing search. A state is a filled-in grid (2D NumPy array) with n^2 of each of the numbers from 1 to n^2. A transition occurs by swapping two numbers that are not in the *clue* indices. Finally, we can measure the number of "conflicts" for a given state by counting the number of errors in each row, column, and subgrid.

The code that we provide generates a sudoku puzzle, which you can test out yourself. Given the order (n) of the desired puzzle and the number of clues (which should be fewer than n^4, the total number of cells), `generate` returns a dictionary (`problem`) specifying the indices and values of clue cells. The `display` function prints out the board with the clues filled in and zeroes elsewhere. `initialize` returns a state that satisfies the clues (but is not necessarily a solution), and `successors` returns a list of valid successor states given a current state. The latter two functions will be used by your hill-climbing procedure.

3.1: Computing Errors (5 points)

Write the `num_errors` function that will count and return the number of errors in a given state. The simplest way to do this would be to iterate over each of the n^2 rows, n^2 columns, and n^2 non-overlapping $n \times n$ subgrids and count the number of missing values in each. Note that this will inevitably double- or triple-count individual errors.

3.2: Basic Hill-Climbing (10 points)

Implement the basic hill-climbing procedure in `hill_climb`, ignoring the optional arguments for now. You should only initialize the given problem once, and no sideways moves are permitted. Either move to a "better" successor state with the lowest number of errors (choose a random one if there are multiple best states), or return the current state if no successor state has fewer errors than the current one. Also create a list tracking the number of errors in each iteration and return that along with the solution.

3.3: Preliminary Tests (5 points)

You should find that your basic procedure performs decently on order-2 puzzles, although it is certainly prone to getting stuck at local minima. Generate a few different puzzles with $n = 2$ and $c = 5$ clues (the minimum number of clues for a unique solution, though not sufficient for all puzzles). For two puzzles, one that `hill_climb` can solve and one that `hill_climb` cannot solve successfully, show the problem using `display`, the returned solution state, and a plot showing the number of errors over each iteration.

Run `hill_climb` in batch on 100 random order-2 puzzles with 5 clues each. Report the average success rate (proportion of final states with 0 error) as well as the average error over all final states (counting both successes and failures).

3.4: Sideways Moves (5 points)

The basic hill climbing procedure can be improved in a couple ways. First, we should allow for sideways moves. If the best neighbors all have the same number of conflicts as the current state, we can move to a random one. You should also keep a counter and increment it when a sideways move is taken consecutively. If `max_sideways` moves are made consecutively, quit and return the current state.

Generate a puzzle with $n = 2$ and $c = 5$ in which at least one (preferably more) sideways move is taken and show the corresponding error per iteration plot. The sideways move(s) should clearly be seen as a horizontal segment on this plot. Then do a batch solve of 100 random puzzles allowing up to 10 sideways moves each, and report the success rate and average error.

3.5: Random Restarts (5 points)

The last modification of hill climbing that you will implement is including random restarts. When either all current neighbors are worse than the current state or when `max_sideways` moves are made, we should initialize a new sudoku board and assign it to be the current state. A counter on the number of restarts should be created and incremented each time this happens. The return condition would be when we have hit `max_restarts` or when a solution is found.

Run the batch of 100 ($n = 2$, $c = 5$) puzzles with `max_sideways` and `max_restarts` both equal to 10 and report your findings. Lastly, experiment with ($n = 3$, $c = 40$) puzzles; these are standard-sized sudoku problems, but on the easier side given the number of clues. Run hill climbing on a few of these and show the result of one puzzle that was successfully solved, including the given problem, returned solution, and a plot of errors per iteration. This may take a few tries, and you may tweak the `max_sideways` and `max_restarts` parameters. Write down the parameter values that you used.

Problem 4: Connect k (30 points)

The game of Connect Four bears some similarities to tic-tac-toe. Two players take turns placing marks or pieces on a grid, trying to be the first to achieve four consecutive pieces. The difference, aside from the size of the board (typically six rows, seven columns) and that we want four rather than three in a row, is that the grid is *vertical* and affected by gravity. In short, a player's actions only consist of the *column* in which to place their piece, and the piece must then subsequently be placed in the bottom-most free cell in that column.

We are providing a simple Python scaffold that will be developed into a Connect Four-playing agent using alpha-beta depth-limited search. A game state is represented as a 2D NumPy array. We will use `'X'` and `'O'` to represent each player just as in tic-tac-toe, and the character `'.'` to represent an empty cell. To make our implementation more general, we will allow grids of any size ($m \times n$) and make this a "Connect k" game, rather than just Connect Four. `'X'` will always go first and be the maximizing player, and `'O'` will be the minimizing player. `'X'` wins have positive utility, `'O'` wins have negative utility, and draws have zero utility.

We provide several utility functions; the relevant ones for you are `terminal` and `eval`. `terminal` takes in a current game state and returns either a utility value if the state is terminal or `None` if the game is not terminal (this thus serves as both a utility function and terminal test). `eval` takes in the same inputs as `terminal` and returns an evaluation of a non-terminal state based on both players' *potential* for winning (how this is computed is not important for this assignment, but we are happy to talk more about it if you are interested). You will use both functions when implementing alpha-beta search.

Finally, given the game parameters, `game_loop` runs a game from start to finish, using your implemented alpha-beta search procedure for both players. While this nominally means that both players will be playing with identical search capabilities, the optional `X_params` and `O_params` will allow you to tweak the strength of each player in the form of search depth.

4.1: Successor Function (5 points)

Before writing alpha-beta search, we first need to write the `successors` function. Given both a state and the player making a move (either `'X'` or `'O'`), return a list of possible successor states. Be sure to follow the rules of Connect k as described above. Note that this function combines the `actions` and `result` functions if following the textbook's pseudocode.

4.2: Depth-Limited Alpha-Beta Search (20 points)

We provide the wrapper function for depth-limited alpha-beta search, which is called repeatedly by `game_loop`. You will write the two recursive functions, mostly following the textbook pseudocode but with some additional functionalities. The arguments to each are the state, alpha and beta values, k value, current node depth, and maximum allowed search depth. Each function should perform the following tasks:

- Perform the terminal test; if terminal, return the state's score and `None` move.
- If current node depth is equal to maximum allowed depth, this state is treated as a leaf. Run the evaluation function on the state and return the value and `None` move.
- Otherwise we expand the node's successors and search them using the alternative recursive function. Before doing so, you must generate and sort the successors such that the "best" nodes occur first for each player. You can do this by calling `eval` on each successor and then sorting them from largest to smallest for MAX and smallest to largest for MIN.
- The recursive calls should be similar to the pseudocode. Be sure to increment the current node depth by 1 when you make a new recursive call.

4.3: Testing Connect k (5 points)

Once you are ready to test your implementations, have your agent play itself in Connect 3 and then Connect 4, both on a 4×4 board. Run both **without** depth limiting; you can simply omit the two optional arguments to `game_loop`. Show the final state for each.

To play larger games, depth limiting will be necessary. Have your agent play the full Connect 4 game on a 6×7 board. Show the final game states for maximum search depths of 5 and 6 (for both players). You should find that one ends in a draw and the other ends in a win for one player. Then try to change the depth values, making them different if necessary, so that the *other* player wins the game. Show the final state for this last run.

Submission

You should have one PDF document containing your solutions for problems 1-2, as well as the information and plots in the relevant parts of 3-4. You should also have the completed code file implementing problems 3 and 4; make sure that all provided function headers are unchanged. Submit the document and code files to the respective assignment bins on Gradescope. **For full credit, you must tag your pages for each given problem on the former.**