COMS W4701: Artificial Intelligence, Spring 2022

Homework 1

Instructions: Compile all solutions to the written problems on this assignment in a single PDF file (typed, or handwritten *legibly* if absolutely necessary). Coding solutions may be directly implemented in the provided Python file(s). When ready, follow the submission instructions to submit all files to Gradescope. Please be mindful of the deadline, as late submissions are not accepted, as well as our course policies on academic honesty.

Problem 1: Wordle (15 points)

Let's describe the game of Wordle as a task environment. This is a single-player game in which the objective is to guess a hidden word in six tries or fewer. For each incorrect guess that the player (agent) makes, the computer (environment) responds with hints regarding whether a letter is actually present in the hidden word and if it is in the right place, or if a letter does not appear in the hidden word at all. We only consider the "hard" version of the game, in which each subsequent guess **must** be consistent with all hints so far.

(a) Give a state space description of this problem. What information should individual states contain? What are the valid actions that an agent can take in each state?

(b) Classify this task environment according to the six properties discussed in class, and include a one- or two-sentence justification for each. For some of these properties, your reasoning may determine the correctness of your choice.

Problem 2: Utilities and Preferences (15 points)

Consider two lotteries $L_1 = [p, A; 1-p, C]$ and $L_2 = [q, B; 1-q, L_1]$, where $A > B > C > 0$.

(a) Compute $U(L_2)$. For what value of p is $U(L_2)$ a fixed value independent of q?

(b) Suppose that $L_1 \succ L_2$. Is it guaranteed that $p > q$? Explain why or why not. (You may use a counterexample to prove your assertion if your answer is no.)

(c) L_2 can be represented as a lottery with each of A, B, and C as three separate outcomes. Solve for p and q in terms of A, B, and C such that each of the three outcomes has the same expected utility when weighted by their likelihoods.

Problem 3: Monty Hall (20 points)

This is a variation of the famous Monty Hall problem. Suppose you are a contestant on a game show and you have to open one of 50 closed doors. Behind one door is a job offer from a FAANG company ($U = 100$). Behind another door is your school tuition bill ($U = -10$). Behind all other doors is a goat ($U = 1$).

(a) Compute the expected utility of opening a door at random.

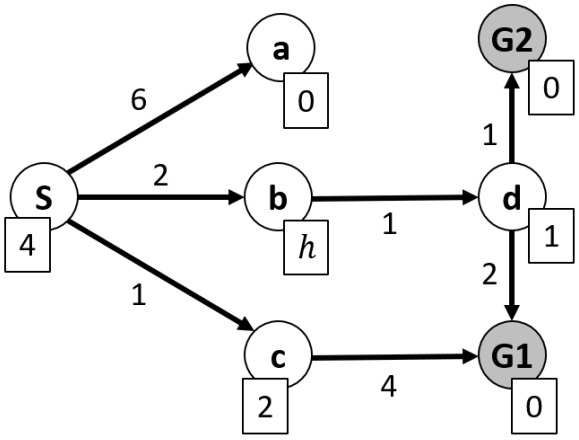
(b) Suppose you've chosen a door. Before you open it, the show host offers to open 5 doors with a goat behind them. What is the expected utility of switching to another door choice?

(c) Is it better to stick with your original door or switch to another one? Compute the value of information of seeing 5 doors that are hiding goats.

(d) The 5 doors have been revealed and there are now 45 unopened doors left. The show host offers to open your initially chosen door, after which you can either claim that door's prize or open another unopened door. What is the expected utility of this offer?

Problem 4: Search Practice (15 points)

In the state space graph below, S is the start state and G1 and G2 are both possible goal states. Costs are shown along edges and heuristic values are shown in the rectangles next to each node. Assume that search algorithms expand states in alphabetical order when ties are present (G1 comes before G2).



(a) List the ordering of the states expanded as well as the solution (as a state sequence) returned by each of DFS, BFS, and UCS. Assume that DFS and BFS use the early goal test. Assume that all algorithms use a reached table.

(b) For what range of values is h an admissible heuristic? For what range of values of h does A* return a suboptimal solution? For what range of values of h does A* expand fewer nodes than UCS?

Problem 5: Word Ladders (35 points)

In this problem you will implement and compare the performance of the various implementations of best-first search on word ladder puzzles. Given two English words, the goal is to transform the first word into the second word by changing one letter at a time. The catch is that each new word in the process must also be an English (dictionary) word. For example, given a start word "fat" and a goal word "cop", a solution would be the word sequence "fat", "cat", "cot", "cop".

In our implementation, possible words will correspond to different states. Successor states are thus words that differ from the given word by one letter. Using this idea, we provide a `successors` function that returns a list of "actions" and successor states given a state. The action is simply the index of the changed letter. This function uses the `pyenchant` library to perform dictionary checking. Finally, the "cost" of each action can simply be treated uniformly (e.g., 1).

To perform search, we represent a search tree node using a Python dictionary containing three components: state, parent, and cumulative cost (keeping track of action is not useful for us here). For example, the root node may be defined as {`'state':start, 'parent':None, 'cost':0`}. The frontier can be implemented as a list and the reached set as a dictionary of states and costs.

5.1: Best-First Search (15 points)

Use the provided `successors` function to implement best-first search. The arguments are the start state, goal state, and priority function. Your function should implement the frontier and reached set as described above. You are recommended, but not required, to write a separate helper function for node expansion. The search should conclude once the goal state is found.

To simulate priority queue behavior of the frontier, the heapq module, and in particular the `heappush` and `heappop` functions, can be used to efficiently treat regular Python lists using a priority function. In order to "sort" the nodes in the frontier, you can first place each node within a data structure like a tuple, so that the first element captures the priority value and the last element is the node itself. For example, `(1, node1)` would be ordered before `(2, node2)`. Note that if the first elements of two tuples are equal, they are then compared according to their second elements, followed by their third and so on. For this problem, you should break ties (and thus include an additional element between the f-cost and node) by alphabetical order of the states.

In addition to returning the node containing the goal, your function should also update and return two additional quantities: `nodes_expanded` and `frontier_size`. The first is an integer that is incremented every time that a node is expanded. The second is a list that contains the size of the frontier at each iteration of the search, updated before a node is popped.

5.2: Priority Functions (5 points)

Given a working implementation of best-first search, it remains to write the different priority functions that lead to the various search algorithm behaviors. A priority function takes in a node and the goal state as arguments, although the latter does not necessarily have to be used (particularly by uninformed search algorithms). Implement the priority functions for depth-first search, breadth-first search, and uniform-cost search. You should be able to do each with just a single line of code; they should all look quite similar (perhaps even identical).

A priority function for A* search should use a suitable heuristic. The Hamming distance between the current state and goal state can act as a heuristic; this is simply the number of letters that are different between the two words. Implement the priority function for A* search using the Hamming distance as a heuristic.

5.3: Analysis (15 points)

You should now be able to run your program and solve word ladder puzzle instances. We provide a `sequence` function that takes the goal node output from best-first search and returns the entire sequence of words from start to goal. Compare the performance of all algorithms on the first puzzle, and all algorithms except for DFS on the second and third puzzle:

- Start: "fat"; goal: "cop"
- Start: "cold"; goal: "warm"
- Start: "small"; goal: "large"

For each puzzle, report the length of the solution and number of nodes expanded for each algorithm. Also for each puzzle, show a figure (e.g., using `matplotlib`) with three line plots showing the size of the frontier of each algorithm per iteration (you may exclude DFS for the first puzzle). It would be simplest to write this code directly in or call it from the `main` function of your code file. Include the numbers and plots on your pdf writeup. Please also briefly address the following questions:

- Which algorithms are optimal? Why do we generally want to avoid DFS for this problem?
- Which algorithms have roughly the same performance regardless of puzzle?
- Explain the trend of the frontier size. What does it mean for a search to finish when the frontier size is still increasing, near its maximum, or decreasing?

Submission

You should have one PDF document containing your solutions for problems 1-4, as well as the information and plots for 5.3. You should also have the completed code file implementing best-first search for the word ladder problem in a `.py` file; make sure that all provided function headers are unchanged. Submit the document and code file to the respective assignment bins on Gradescope. **For full credit, you must tag your pages for each given problem on the former.**