# COMS W4701: Artificial Intelligence, Summer 2022

Homework 1

**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: HODL (15 points)

Consider the problem of investing in cryptocurrency. We have many trader agents who must decide when and how much to invest at any given time. Their decisions are based on individual risk levels, current market conditions, and inferences about company events (e.g., IPO, earnings reports, acquisition, etc.).

(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: Monty Hall (15 points)

This is a variation of the famous Monty Hall problem. Let's say we play the following game prior to our final exam. You are given 50 mystery boxes and you have to choose one of them. One mystery box grants you a free A on the final ($U = 20$). Another one will force you to retake the entire class in the fall ($U = -100$). All other boxes give you a single bonus point on the final exam ($U = 1$).

(a) Compute the expected utility of selecting a mystery box at random.

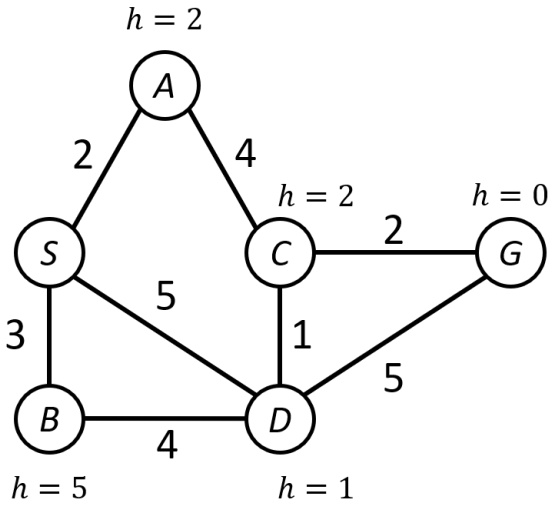
(b) Suppose you've chosen a box. Before you open it, we choose and open five other boxes granting the single exam bonus point. What is the expected utility of switching to another box after this occurs?

(c) Is it better to stick with your original choice or switch to another one? Compute the value of information of seeing the five boxes granting the single bonus point.

(d) We will now slightly change the game rules. After you select a box, we will open it for you. You can then either claim the revealed prize, or choose to open another mystery box and keep the new prize (forgoing the first one). What is the expected utility of this game procedure?

## Problem 3: Search Practice (20 points)

In the state space graph below, S is the start state and G is the goal state. Costs are shown along edges and heuristic values are shown adjacent to each node. All edges are bidirectional. Assume that search algorithms expand states in alphabetical order when ties are present.




(a) List the ordering of the states expanded as well as the solution (as a state sequence) returned by each of DFS and BFS, assuming that they use the early goal test (insertion into the frontier) and a reached table.

(b) List the ordering of the states expanded as well as the solution (as a state sequence) returned by each of DFS and BFS, assuming that they use the late goal test (removal from the frontier) and a reached table.

(c) List the ordering of the states expanded as well as the solution (as a state sequence) returned by each of UCS and A*, assuming that they use the late goal test (removal from the frontier) and a reached table.

(d) Change exactly one heuristic value so that A* returns a suboptimal solution. Give the value of the heuristic that you changed and explain why it becomes inadmissible. List the ordering of the states expanded as well as the solution (as a state sequence) returned by A*.

(e) A heuristic value has not been assigned to S. Give the upper bound on the value of $h(S)$ so that i) $h$ is admissible (but possibly inconsistent), and ii) $h$ is consistent (and admissible). Do these values change how A* runs on this graph?

## Problem 4: Word Ladders (50 points)

In this problem you will implement and compare the performance of various search algorithms on solving 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".

We have provided a Python skeleton file for you to complete this problem. Note that the function headers are type annotated to clearly indicate argument and return value types. You can optionally use static type checkers like mypy to verify your code as you work on it.

In our implementation, **states** will be equated with words (strings). Successor states are words that differ from the current state 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 be treated uniformly (e.g., 1).

To perform search, we represent a search tree **node** as a Python dictionary containing three components: state, parent, and cumulative cost. For example, the root node may be defined as {`'state':start, 'parent':None, 'cost':0`}. The frontier can be implemented as a list, while the implementation of the reached set will differ in each part below.

### 4.1: Depth-Limited Depth-First Search (15 points)

We will start by implementing depth-limited DFS. First, write the `expand` function, which takes in a node and returns a list of nodes, one per successor state to the state in the given node. You should use the provided `successors` function here.

The `depth_limited_dfs` function takes in a start state, a goal state, and a depth limit. Nodes at `depth` will be considered to be leaves. In this procedure, you can implement the reached set as a list of *states* (as opposed to the frontier, which is a list of *nodes*). Since this is DFS, you should perform the **early goal test** for time efficiency by checking for the goal upon a node's insertion into, rather than removal from, the frontier.

You should also continually update two local variables: `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 the beginning of each search iteration, updated before a node is popped. Once the goal node has been found, the procedure returns that node, along with `nodes_expanded` and `frontier_size`. Alternatively, if the frontier becomes empty, you should return `None` along with the latter two values.

If you would like to test your solution before moving on, you can run the code in the main function and also add your own (see 4.4). Investigate the return values of `depth_limited_dfs`. You can also pass in the goal node into the provided `sequence` function to retrieve the sequence of words from start to goal. Think about how changing the depth limit affects the solution.

### 4.2: Iterative Deepening (5 points)

Since we have a DFS implementation that can account for depth limits, a natural extension would be an iterative deepening wrapper around it. Recall that this algorithm repeatedly calls `depth_limited_dfs` with a larger `depth` parameter each time, starting from 0. To prevent this from potentially searching forever, `iterative_deepening` will stop and return no solution if one is not found by `max_depth`.

As with `depth_limited_dfs`, `iterative_deepening` should also update and return quantities indicating `nodes_expanded` and `frontier_size`. The former should be the total number of expanded nodes over all search iterations. The latter should be a single flat list containing the frontier sizes over all iterations.

### 4.3: A* Search (10 points)

Your iterative deepening implementation, like all uninformed search approaches, uses no information about the goal word. But we should use this knowledge to our advantage. It makes sense to "favor" successor words that look more like the goal. We can do this using A* search, and a suitable heuristic would be the Hamming distance between a current word and the goal, or the number of indices where the corresponding letters are different.

A* search will mostly follow the implementation of depth-first search with a few key changes. 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 f-cost and the last element is the node itself. For example, `(1, node1)` would be ordered before `(2, node2)`. 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.

Another difference is that the `reached` structure is now a *dictionary* with each key as a state and the value as the cheapest node reaching that state. Even if a state already exists in `reached`, a cheaper path to it may be discovered later, and so a cost comparison should be done when determining whether a child node should be added to the frontier. Finally, remember to conduct the goal test only when **popping** a node from the frontier—the "early" version should no longer be used.

Implement the `astar_search` function following the specifications above. The return values are the same as those of your other implementations.

### 4.4: Analysis (20 points)

You should now be able to run each search algorithm implementation and solve different 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. Consider the puzzles below:

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

(a) Let's investigate A* first as the results will help us better understand the other two algorithms. Run A* on each of the puzzles above and report the solution length as well as number of nodes expanded. Also generate three line plots (e.g., using `matplotlib`), one per puzzle, showing the size of the frontier per iteration.

(b) Now let's look at iterative deepening search. From what you saw with A*, what is the range of maximum allowable depth values that would yield solutions for the first two puzzles but no solution for the third? Explain whether these solutions would be identical to those of A*.

(c) Pick two different integer values in the range you found above and perform two runs of IDS on the three puzzles with these max depths (we recommend you choose smaller values for faster runs). Report the solution lengths and number of nodes expanded for each, and compare these values with those of A* for each puzzle.

(d) Do the same experiment using the max depth values you used above for depth-limited DFS. Report the solution lengths and number of nodes expanded. Compare these values with those of IDS for each puzzle. Why might the results be different even though the maximum depths are all identical?

## Submission

You should have one PDF document containing your solutions for problems 1-3, as well as the information and plots for 4.4. You should also have the completed code file implementing the search algorithms for the word ladder problem in a `.py` file. 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.**