COMS W4701: Artificial Intelligence, Spring 2024

Homework 1

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 (6 points)

Read the following article on *Nature* (click "Access through your institution", type and select "Columbia University", and then log in using your UNI):

- ChatGPT has entered the classroom: how LLMs could transform education

Respond to the following questions with about three or four sentences each. There are no correct or incorrect answers, but try to think about what you've read and use any other knowledge or experiences you have had to formulate thoughtful responses.

1. Do you think that there are legitimate use cases for ChatGPT in every university class, either now or in the near future? Or are there certain subjects that should not incorporate any AI tools whatsover? Give a couple examples to support your response.
2. Suppose you used ChatGPT for homework in this class. What negative effects, if any, do you think it may have on your learning? You may refer to examples raised in the article or otherwise. Please try to reflect on your **individual** learning style and tendencies.
3. Suppose you used ChatGPT as a general learning supplement in this class. What positive effects, if any, do you think it may have on your learning? You may refer to examples raised in the article or otherwise. Please try to reflect on your **individual** learning style and tendencies.

Problem 2 (12 points)

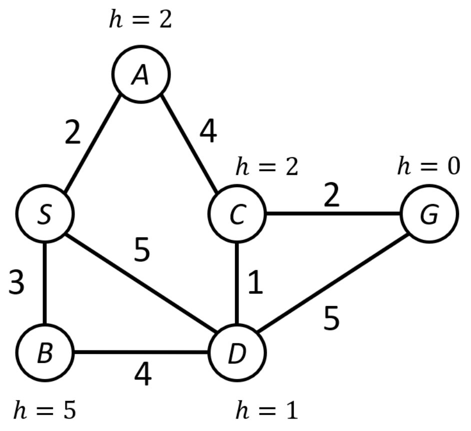
AlphaGeometry was a recently announced AI system that can solve geometry problems at the "gold-medal" level of the International Mathematical Olympiad. (You should read the linked blog post to complete this problem, but you do not have to read the scientific article on *Nature*.)

1. (4 pts) Give a complete PEAS description of the task environment to which AlphaGeometry is a solution.
2. (6 pts) 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.

3. (2 pts) Would AlphaGeometry best be classified as a simple reflex agent, model-based reflex agent, goal-based agent, learning agent, or some combination of these? Briefly explain your answer.

Problem 3 (16 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 undirected (or bidirectional). Assume that node expansion returns nodes by alphabetical order of the corresponding states, and that search algorithms expand states in alphabetical order when ties are present.



Here, the *early* goal test refers to checking that a state is the goal right *before* insertion into the frontier. The *late* goal test refers to checking it right *after* popping from the frontier. Usage of a *reached* table allows for multiple path pruning.

1. (3 pts) Suppose we run DFS. Assuming that it finishes, list the sequence of states that are **expanded** (not including the goal), as well as the state sequence **solution**, for each of the following scenarios: a) early goal test and reached table, b) late goal test and reached table, c) late goal test and no reached table. Otherwise, indicate that DFS does not finish.
2. (3 pts) Repeat part 1 above for BFS.
3. (2 pts) Suppose we run UCS with a reached table. List the sequence of expanded states as well as the solution for each of the following scenarios: a) early goal test, b) late goal test.
4. (2 pts) Repeat part 3 above for A* search.
5. (4 pts) Suppose we change the heuristic function. Find the range of nonnegative heuristic values for $h(A)$ that would cause A* (utilizing a late goal test and reached table) to return an suboptimal solution, or indicate if A* would behave optimally regardless of $h(A)$. Repeat for $h(B)$, $h(C)$, and $h(D)$.
6. (2 pts) A heuristic value has not been assigned to S. Give the upper bound on the value of $h(S)$ so that a) h is admissible (but possibly inconsistent), and b) h is consistent (and admissible).

Problem 4 (16 points)

We are trying to schedule five activities (A, B, C, D, E) into four timeslots (1, 2, 3, 4). Each activity may be assigned to any of the timeslots, and a timeslot may contain no or multiple activities, but the following constraints must be satisfied:

- $A > D$
- $C > E$
- $A \neq C$
- $C \neq D$
- $B \geq A$
- $D > E$
- $B \neq C$
- $C \neq D + 1$

1. (5 pts) We set up this problem as a constraint satisfaction problem and run arc consistency. Find the domain of A if it is the first variable to be made arc-consistent with the full domains of all other variables. Repeat for each of the other variables, indicating their domains in the scenario that each is the first variable to be made arc-consistent.
2. (5 pts) Now continue from the domains that you found by repeatedly making all variables arc-consistent with all other variables (i.e., run the AC-3 algorithm). List the resultant domains when the process finishes.
3. (2 pts) Is arc consistency by itself sufficient to yield a unique solution to the scheduling problem? Briefly explain why or why not.
4. (2 pts) Find all valid solutions to the scheduling problem.
5. (2 pts) Going back to the arc-consistent, but unsolved, CSP in part 2, make a variable assignment that does not appear in a valid solution. Trace the subsequent constraint propagation steps, and indicate which constraint is ultimately violated, leading to backtracking.

Problem 5 (50 points)

You will be implementing a path planning solution for a robot in an environment with a set of terrain features. The environment is discretized as a grid world surrounded by four impassable walls. Within the walls, the robot can move to one of up to eight adjacent cells from a given cell. You will implement a variety of search algorithms and analyze their capabilities and outputs.

NumPy Occupancy Grid: The world is represented by a 2D NumPy array indicating the status of each cell. Each cell takes on a value from the set $\{0, 1, 2, 3\}$ indicating the terrain feature at that cell. Each terrain feature indicates the cost incurred when passing through a given cell:

- A *flatland* cell has a value of 0 and a cost of 3. These are the most common cells that the robot encounters.
- A *pond* cell has a value of 1 and a cost of 2. Since robots cannot move through water, a sailor offers to take the robot through lake cells.
- A *valley* cell has a value of 2 and a cost of 5. The robot must exercise precaution to ensure it does not topple over.
- A *mountain* cell has a value of 3 and a cost of ∞. The robot cannot pass through these cells.

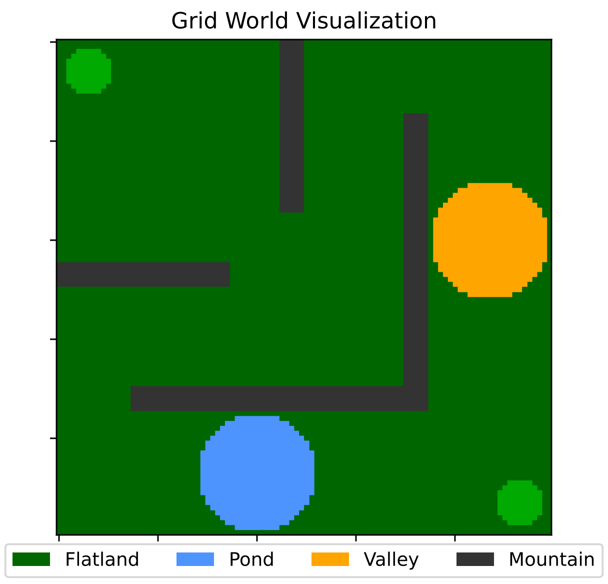


Figure 1: Example grid world. The start and goal cells are demarcated by the highlighted green regions. Terrain features are also present, including flatland, pond, valley, and mountain cells.

Custom Worlds: We have provided four example grid worlds with different arrangements of terrain, all saved as NumPy `.npy` files. You can load an environment using `numpy.load()` and visualize it using `visualize_grid_world()` in the `utils.py` file. An example is shown in Figure 1.

All coding implementations in the following parts will be completed in the `path_finding.py` file, and you will just need to submit this file to Gradescope.

Part 1: Uninformed Search (14 points)

We will start by implementing the two uninformed search algorithms of depth-first search and breadth-first search. These do not consider the true costs of the cells in the gridworld. Flatland, pond, and valley cells are all treated the same. The only exception to this rule is that mountain cells, with a cost of ∞, are impassable; movement into a mountain cell is not allowed.

Implement the `uninformed_search()` method. The inputs are a `grid`, the `start` and `goal` states (both tuples of grid coordinates), and a `PathPlanMode` variable called `mode`. `mode` may have value `PathPlanMode.DFS` or `PathPlanMode.BFS`, indicating which algorithm to run. For node expansion, you should use the `expand` method in `utils.py`. You should also not add cells to the frontier that have been previously encountered.

Data structures: You will need to define and update the following variables during the search process. Please adhere to all specs, as some of these will be returned when the method completes.

- `frontier`: This stores the frontier states as the search progresses. For uninformed search, it is sufficient to implement it as a list, and you can remove from either the back or the front of the list depending on the search method. Add successor states to the frontier *in the same order as they appear when returned from* `expand()`.

- `frontier_sizes`: This is a list of integers storing the size of the frontier at the beginning of each search iteration (before popping and expanding a node).
- `expanded`: This is a list of all expanded states. You can simply append each state to the list after popping it from the frontier.
- `reached`: This can be implemented as a dictionary. Keys are the states that have been reached (again, these are just tuples of grid coordinates), and corresponding values are their parent states. There is no need to track actions or cost information.

If the goal has been found, you will need to reconstruct the `path` solution. This should be a **list** of coordinate tuples, with `start` and `goal` as the first and last elements, respectively, and a sequence of valid adjacent cells between them. If the goal was not found, then `path` would just be an empty list. The completed method should return `path`, `expanded`, and `frontier_sizes` (in that order).

Part 1.5: Testing Uninformed Search

Before moving on to the next part, it is a good idea to test that your implementation is working correctly (or at least running without errors). From a terminal, you can run the command `python main.py worlds [id]`, and replace `id` with an integer between 1 and 4 for the sample world that you would like to test on. After doing so, you should see a summary of the search results of DFS and BFS in the terminal, as well as a figure pop up with the corresponding visualization.

One option that can be toggled with the command above is animation. By default, a static image pops up showing expanded cells in brown and the path in red. You can use the `-a` option to show an animation instead. A 1 argument value will animate the expansion process, and a 2 argument value will animate the path.

Part 2: A* Search & Beam Search (14 points)

Now you will implement basic A* search, as well as a small variation of it as beam search. The `a_star()` method takes in the same inputs as `uninformed_search`, in addition to two new ones. `mode` will be either `PathPlanMode.A_STAR` or `PathPlanMode.BEAM_SEARCH`. `heuristic` will be either `Heuristic.MANHATTAN` or `Heuristic.EUCLIDEAN`. `width` will only be used for beam search.

The frontier should now be implemented as a `PriorityQueue` object. To ensure proper sorting of the frontier, each element can be a tuple of the form `(priority, state)`, where `priority` is computed as the sum of the cell's backward cost g and heuristic h. You can use the `cost` method in `utils.py` to compute a cell's cost, and you will need to compute the heuristic value, using either Manhattan or Euclidean distance from the cell to the goal.

The values of the `reached` dictionary must now store a state's parent as well as cost. A reached state may be re-added to the frontier if its new cost is lower than the previously encountered one. Finally, if `mode` is `PathPlanMode.BEAM_SEARCH`, the frontier should only keep the `width` most promising nodes at the end of each search iteration.

As in `uninformed_search`, your method should return `path`, `expanded`, and `frontier_sizes`. If the goal was not successfully found, `path` should be an empty list.

Part 2.5: Testing A* Search

To test A*, you can again run `python main.py worlds [id]`, and add to this one or two more arguments. With the `-e` option, an argument of 1 will set Manhattan distance, and 2 will set Euclidean distance. This will run both A* and beam search, the latter with a default width of 100. You can also set the `-b` option followed by an integer specifying a different width for beam search.

Part 3: IDA* Search (14 points)

A* can result in large frontiers, which can be especially apparent as the size of the gridworld grows. Beam search is one way of controlling frontier size, but it can potentially prune away optimal solutions. Iterative deepening is another variant that may end up taking more time to run, but can save on space while still guaranteeing an optimal solution.

For the last task, we provide `ida_star()`, which calls the helper method `__dfs_ida_star()`, which you will complete. This will incorporate elements of both DFS and A*. The frontier can be implemented as a regular list as in DFS, and nodes should be expanded in LIFO order. However, a successor node should only be added to the frontier (and `reached`) if **both** of the following hold:

- It is not in `reached`, or its cumulative cost g is less than its previous cost in reached. `reached` will thus be implemented as in A*, storing both a state's parent and its best cost so far.
- The sum of its cumulative cost and heuristic, $g + h$, is less than or equal to `bound`. Otherwise, it is skipped, and `next_bound` is potentially updated as described below.

If the goal is found at some point, then this method concludes and the same quantities are returned as in the previous implementations. A fourth quantity `next_bound` should also be returned as well. This contains the *minimum* of all $g + h$ values of the skipped nodes. (You will need to store and update this value during the search process.) `ida_star()` will take this and set this to be the next `bound` value when starting the search again.

The final `expanded` and `frontier_sizes` lists returned by `ida_star()` may have different sizes. `expanded` only contains the expanded states in the **last** iteration of `ida_star()`, so a state appears at most once in this list even if it is expanded multiple times. `frontier_sizes` contains the size of the frontier over **all** iterations of `ida_star()`.

Part 3.5: Testing IDA* Search

To test IDA*, you can set `-e` to either 3 or 4 to run IDA* with Manhattan or Euclidean distance, respectively. Note that IDA* can take a much longer time than regular A* or beam search, so it should be sufficient to just run it on world 4, which is smaller than the other worlds, specifically using the Manhattan heuristic.

Part 4: Experimentation & Analysis (8 points)

Once you have finished your implementations, perform the following tasks and provide responses to the questions in the same PDF document as your solutions to the previous problems.

1. Run uninformed search on each of the four worlds. Show the DFS and BFS output figures for one of the worlds (your choice) in your document. What do you notice about the *empirical* time and space complexities of DFS and BFS as indicated by the number of expanded

nodes and maximum frontier size? Explain any discrepancies as compared to the *theoretical* complexities of the two algorithms.

2. Run A* and beam search (using the default width) on each of the four worlds. You should verify that both heuristics produce similar results. Compare the resultant space complexities of the two algorithms, as well as the optimality of the solutions returned. Show the output figures of the two algorithms for a world in which the solutions for A* and beam search are not identical, and explain why that may occur.

3. Tuning the beam width in beam search can be a delicate process. Let's focus on world 3. Try some values of width smaller than 100 and observe what changes, if any, occur to the beam search results (using either heuristic). Around what width value do we start seeing a suboptimal solution? Around what width value do we fail to find a solution? Report the two values that you find, and show the output figures for beam search in each case (the latter should show no path).

4. Run IDA* on world 4 using the Manhattan heuristic. Show the output figure for IDA*. What do you notice about the optimality of the solution, as well as time and space complexities? In what situations would IDA* be a good use case (compared to regular A*), and when might it not be?

For code submission, you will just need to upload the `path_finding.py` file to Gradescope.