COMS W4701: Artificial Intelligence, Summer 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 (12 points)

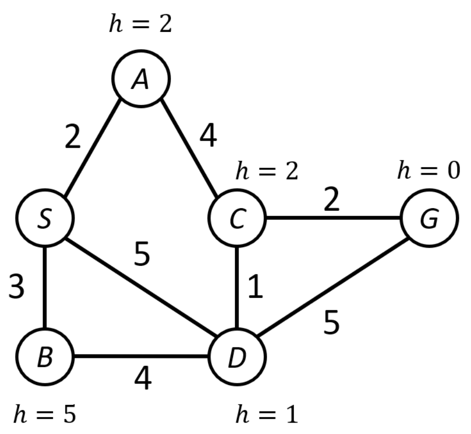
One potentially good way to use ChatGPT, in this and/or other classes, is in the manner of an interactive tutor. You can (repeatedly) ask the tool for step-by-step explanations, examples, and references to related resources. Try it out for topics that we are covering in this class (e.g., A* search) and other areas.

1. (4 pts) Give a complete PEAS description of the task environment to which the ChatGPT tutor 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 the ChatGPT tutor best be classified as a simple reflex agent, model-based reflex agent, goal-based agent, learning agent, or some combination of these? Assume that the underlying models do not change or update due to interaction with users. Briefly explain your answer.

Problem 2 (24 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 in **reverse alphabetical order** of the corresponding states, and that search algorithms expand states in **reverse alphabetical order** when ties are present.

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 and cycle pruning.



1. (6 pts) Suppose we run DFS. Assuming that it finishes, list the sequence of states that are **popped** from the frontier, 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. (6 pts) Repeat part 1 above for BFS.

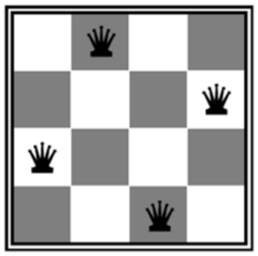
3. (4 pts) Suppose we run UCS with a reached table. List the sequence of popped states as well as the solution for each of the following scenarios: a) early goal test, b) late goal test.

4. (4 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 $h(D)$.

Problem 3: Local Search (14 points)

We will use local search to solve the 4-queens problem. To simplify representation, a state will be represented by a set of 4 coordinate tuples, one for each queen. Following NumPy indexing, the top left cell has coordinates $(0, 0)$ and the bottom right cell has coordinates $(3, 3)$. So the example state shown below is $\{(0, 1), (1, 3), (2, 0), (3, 2)\}$.



We will define neighboring states as those where exactly one queen is moved to a different column. We will use the "min conflicts" evaluation function h, which counts the number of different queen pairs that are in the same row, column, or diagonal.

1. (2 pts) Consider the initial state with all queens in the left column, or $\{(0,0),(1,0),(2,0),(3,0)\}$. What is its h value? Remember to count *all* pairs of queens in conflict. Are there any neighboring states with the same or greater h value? What kind of feature would this state represent in the state space landscape?

2. (6 pts) Starting from the above initial state, trace the hill-"descent" procedure in which we repeatedly move to a neighboring state with the *lowest* h value among all neighbors, until we can improve no further. Write out the state and h value after each iteration.

3. (2 pts) Now consider the initial state $\{(0,1),(1,3),(2,2),(3,0)\}$. What is its h value? Are there any neighboring states with the same or lower h value? What kind of feature would this state represent in the state space landscape?

4. (4 pts) Starting from the above initial state and now allowing "sideways" moves to neighbors with equal h-values, trace a hill-descent outcome (as in part 2) that results in a consistent solution in the fewest number of iterations. Explain whether it is possible to never find a solution if we make no other changes, and how a simple limit on the number of sideways moves may or may not change this outcome.

Problem 4 (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 cost of 3. These are the most common cells that the robot encounters.
- A *pond* cell has 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 cost of 5. The robot must exercise precaution to ensure it does not topple over.
- A *mountain* cell has a cost of ∞. The robot cannot pass through these 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.

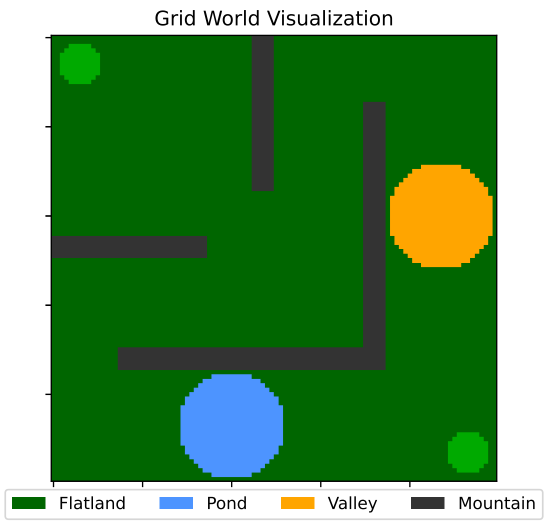


Figure 1: Example grid world. The start and goal cells are demarcated by the highlighted green regions. Terrain features are also present, including flatland, pond, valley, and mountain cells.

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

You should test your implementation before moving on to the next part. From a terminal, you can run the command `python main.py worlds [id]`, and replace `id` with an integer between 1 and 3 for the sample world that you would like to test on. You should then 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. By default, this image is static. You can use the `-a` option to show an animation instead. `-a 1` will animate the expansion process, and `-a 2` 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 and cost. A previously reached state may be re-added to the frontier if its new cost is lower than its old one. Finally, if `mode` is `PathPlanMode.BEAM_SEARCH`, the frontier should only keep the `width` cheapest nodes at the end of each 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: Local Search (14 points)

In some problem settings, we can get away without keeping track of an active frontier. The heuristic function gives us an efficient way of evaluating a gridworld state, and we can use this to locally search our way to the goal from the current state. Implement `local_search()`, which will take in the same arguments as `uninformed_search()`, with the addition of the `heuristic` parameter. This method should **only** return a `path` of nodes as the solution; it should not keep around any frontier or expanded node sets.

At each iteration, you should compute the heuristic values of all neighbors of the current node (last node in `path`). You should append the neighbor with the lowest value to the path, assuming it is lower than the current node. If not, we have hit a dead end and should return an empty list indicating no solution found.

Part 3.5: Testing Local Search

To test local search, you can set `-e` to either 3 or 4 to run it with Manhattan or Euclidean distance, respectively. Note that a solution may not be found in some worlds even if one exists and was previously found using other methods.

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). Show output figures for two cases: 1) a suboptimal solution is returned, 2) no solution is returned. Report the beam width values in each case.

4. Run local search on each world using either heuristic. Show the output figure for one in which it finds a path solution. Briefly explain why it fails to find a solution in the other worlds.

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