COMS W4701: Artificial Intelligence, Spring 2025

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 **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.

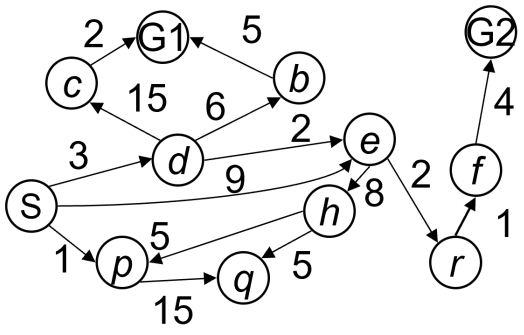
Problem 1 (10 points)

One possibly useful way to use ChatGPT, in this and/or other classes, is by treating it as an interactive tutor. You can interact with the chatbot to ask 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. (2 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 (16 points)

In the state space graph below, the node S corresponds to the initial state, and the nodes labeled G1 and G2 correspond to possible goal states. Action costs are shown along the edges.



1. (4 pts) By inspection, list **all** state sequences starting at S and ending in a goal state, along with their total costs.

2. (6 pts) Suppose we run depth-first search on this graph, and we allow nodes to be visited more than once. Draw out the resultant search tree that is traversed until the first goal node is found as a child of an expanded node (early goal test). If multiple nodes appear in the same tier of the tree, DFS pops the node that occurs **latest** in alphabetical order. There is no need to include action costs in your tree. Indicate the nodes that are still in the **frontier** when search concludes, and also give the state sequence **solution** that is returned.

3. (6 pts) Repeat the above for breadth-first search. This time, assume that we use multiple-path pruning so that no node is visited more than once. A node that is encountered again after its first appearance in the tree is simply skipped.

Problem 3 (12 points)

We will continue examining the state space graph from Problem 2.

1. (4 pts) Suppose we run uniform-cost search with a reached table. Remember that a node may be inserted into the frontier more than once if a cheaper path to it is found later. List the **sequence of expanded nodes** as well as the **solution** for each of the following scenarios: a) early goal test, b) late goal test.

2. (3 pts) Suppose a heuristic function is assigned such that $h(n)$ is the length of the shortest path between n and the closest goal. Thus, each goal node has $h(n) = 0$, their parents have $h(n) = 1$, and so on. A node that has no path to a goal has $h(n) = \infty$. Briefly explain why h is guaranteed to be admissible and consistent on this graph. Propose a simple change to one or more of the action costs that would make h inadmissible.

3. (5 pts) Consider running A* search using the heuristic function defined above. List a) the sequence of nodes that are expanded and their g, h and f values at time of expansion, b) the solution sequence, and c) the nodes and their g, h, and f values that are still in the frontier when the search concludes.

Problem 4: Local Search (12 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 row. 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 top row, or $\{(0,0),(0,1),(0,2),(0,3)\}$. 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. (4 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 $\{(0,3),(1,0),(2,2),(3,1)\}$. 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) What is the result if we were to run local search from the above initial state, while only allowing transitions to neighboring states with a lower h value? Suppose we additionally allow transitions to neighbors with *equal* h value. Trace a hill-descent outcome (as in part 2) that results in a consistent solution in the fewest number of iterations.

Problem 5 (50 points)

You will be implementing path planning solutions 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.

NumPy Occupancy Grid: The world is represented by a 2D NumPy array, with each value representing a cell. Each cell has a type corresponding to one of the following terrain features, along with an associated cost incurred by the robot when passing through that cell:

- A *prairie* cell has a cost of 1. These are the most common cells that the robot encounters.
- A *desert* cell has a cost of 0.5. The robot moves more easily through these cells as there are less flora and fauna to contend with.
- A *pond* cell has a cost of 2. The robot can swim, but it must deploy aquatic gear and move more slowly.
- 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.

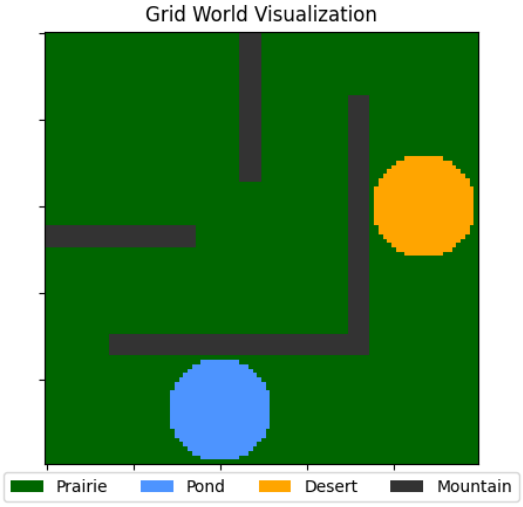


Figure 1: Example grid world with terrain features: prairie, desert, pond, and mountain cells.

Part 1: Uninformed Search (14 points)

Recall that the two uninformed search algorithms of depth-first search and breadth-first search do not consider the true costs of the cells in the gridworld. Prairie, desert, and pond 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 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.

The goal test should be conducted when a node is **popped** from the frontier (**late goal test**). Then 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` must 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: Heuristic Computation (4 points)

You will next implement informed search methods, but these will require a heuristic computation. The `compute_heuristic()` method takes a query `node`, the `goal`, as well as the `heuristic` type to use (either `MANHATTAN` or `EUCLIDEAN` distance). It should compute and return an **admissible** heuristic value of the specified type. Implement these computations with an appropriate *scaling factor* so that admissibility is satisfied. At the same time, the scaling factor should be large enough so that the heuristic value is **equal** to the true Manhattan or Euclidean distance between two adjacent cells of at least one terrain type.

Part 3: 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 can use `compute_heuristic()` from above.

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 3.5: 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 50. You can also set the `-b` option followed by an integer specifying a different width for beam search.

Part 4: Local Search (10 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 4.5: 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 5: 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 three 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. Show the output figures for a world in which the two algorithms return different solutions. Compare the resultant space complexities of the two algorithms, as well as the optimality of the solutions returned in each of the worlds. Also comment on how the choice of heuristic impacts complexity and optimality.

3. Let's take another look at the world for which A* and beam search returned different solutions. Tweak the beam width until you reach a value for which the solutions match (in both path length and cost). Show the output figures, and comment on why this new value is "better" than the default one for finding the solution of A*. How does the complexity of search change in response?

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.