Homework 2: Search and Probability Due Date: February 3rd, 11:59PM

Programming: Search and Rescue (25 points)

In this assignment, you will program a robot agent to search for and retrieve victims in a building that has been hit by some natural disaster. The assignment has you implement various action and heuristic functions to solve increasingly difficult search and rescue missions. Although there is more than one way to approach the problems, we've included comments in the code that will help guide you to where to make changes. It may help to search for all occurrences of "STUDENT CODE." Finally, you can refer to HW1 or the comments in the code for the descriptions of the functions in `map.py`.

We have designed and tested the software to run under Python3 – it may run on other versions of Python, as well, but you are best off using python3 for this and all future assignments in this course.

Introduction

There are several files that are used to implement the search algorithms:

- `SAR.py`: defines the functions for the Search And Rescue (SAR) domain - **this is the only file you will need to modify.**
- `map.py`: map representation for the search and rescue domain
- `params.py`: defines constants and other parameters used in the code
- `search.py`: framework for uninformed and heuristic search - the `SAR_State` class in SAR.py derives from the `State` class in this file.
- `simulate.py`: simulates your search plan to determine failure or success

The environment is an NxM grid, and the agent can move in the four compass directions (N, S, E, W). Map files describe the environment graphically (see, for instance, `maps/medium_v2.map`). `'#'` represents a wall - if the agent tries to move into a wall, it stays where it is. `'E'` represents the entry to the building - there is always just one entry and the agent always starts there (and, when it retrieves victims, they must be moved to there). `'V'` represents a victim. Numbers from 1-9 represent how much damage is in the grid cell (blank cells have value 1). When the agent is moving, this is the cost of traversing the square. **However**, when the agent is carrying a victim, the cost of traversing the cell is the square of the obstacle cost - due of the danger of hurting the victim when the ground is not level.

For all three problems in this assignment, you will make use of the `SAR_State` representation. The state has five attributes:

- `map` - the map associated with the problem you are trying to solve. Do not alter this variable;
- `locn` - the current location of the agent. It is an instance of the class `map.Locn` and has X (row) and Y (column) values;
- `status` - the agent's current status. It can be one of `SEARCHING, VISITED, CARRYING, RETRIEVED`;
- `victims` - a list of `map.Locn` objects that indicate the initial locations of the victims. While, for generality, it is a list, for problems 1 and 2 you may assume that the list is only one element long;
- `goalType` - the type of mission (`VISITED_GOAL` or `RETRIEVE_GOAL`). Problem 1 deals with visit missions, problem 2 deals with retrieve missions, problem 3 deals with both. Do not alter this attribute.

Important Note: Do not augment the state representation. It is sufficient, as presented, to solve all the problems.

Testing

We have provided two files to help you debug and test your code, prior to submission:

- `autograder.py`: this will allow you to unit-test the changes you make to SAR.py. You invoke it using `python autograder.py <options>`. The options are:

  - `-p or --problem`: which problem to test (1, 2, 3)
  - `-s or --step`: which step to test (1 - transitions; 2 - heuristics). Testing transitions determines whether you have implemented the action and cost functions correctly. Testing heuristics determines whether you have implemented the heuristic `costToGoal` function correctly.

  If you do not include one of the above options, it will test all the parts. For instance `"python autograder.py -p 1"` will test the transition and heuristic aspects for problem 1, while `"python autograder.py -s 2"` will test your heuristic implementations for problems 1, 2, and 3.

- `test.py`: this will allow you to test how your implementation solves the search problems. You invoke it using `python test.py <options>`. The options are:
  - `--map`: which map file to use. The choices are in the `maps` subdirectory;
  - `--search`: which search algorithm to use. The choices are DFS, BFS, UCS, GREEDY, A* (defined in `search.py`);
  - `--goal`: which mission to perform. The choices are VISIT and RETRIEVE (defined in `params.py`)
  - `--no-graphics`: by default, a graphics simulation of the agent performing the mission is shown. This option turns off the graphics;
  - `--debugging`: prints out information about the status of the search algorithm (how nodes are added/removed from the queues). May be useful in tracing down bugs in your code.

  If any of the first three options are omitted, it will test all the available choices.

Important Note: We have designed `autograder.py` so that if your code passes all the autograder tests, then the `test.py` tests should also pass (as should the GradeScope autograder). Since this is the first year we've included `autograder.py`, let us know if you do not find that to be the case, and we will try to fix it.

Another Important Note: For each problem, your implementation should build on the previous part. Your final implementation should be able to visit a single victim, retrieve a single victim, and work when there are multiple victims on the map (although, in that case, you need to visit or retrieve just one of the victims, not all of them).

Problem 1 (8 points)

In this problem, the agent's mission is to visit a single victim, that is, it must go to the cell (`map.Locn` object) where the victim is and then stop.

Step 1: You need to implement the `tryToMove` function. The function already has code to determine how the agent would move for different actions (`NORTH, SOUTH, EAST, WEST`). The variable `newLocn` is the location of the agent after attempting the action. Note that this includes cases where the action is not legal (i.e., hiting a wall). If the agent moves somewhere new, return a tuple consisting of a new `SAR_State` indicating the new location and the cost of the move (using function `moveActionCost`). If not, return `(None, 0)`.

Make sure to update the agent's `status` if the move ends up at the same location as the victim. Note how the function `isGoal` is implemented - it returns true for the `VISIT_GOAL` once the agent's status is `VISITED`; this is what tells the search algorithm to stop. As noted above, for this problem you can assume that there is only one element in the victims list.

Step 2: In this step, you need to create a heuristic for the cost of visiting a victim from the agent's current location.

Since the agent moves on a grid, manhattan distance is admissible and informed. You need to implement the `manhattan` function in `SAR.py`, which, given two `Locn` objects returns the manhattan distance between them. Then, use the `manhattan` function to compute the `costToGoal` for the `VISIT` goal.

After testing steps 1 and 2 using `python autograder.py -p 1`, your agent should pass `test.py` with `--map maps/medium.map` and `--goal VISIT`.

Problem 2 (12 points)

In this mission, the agent is to retrieve a single victim. The agent must go to the square where the victim is, execute the `PICKUP` action, go to the entry location, and execute the `PUTDOWN` action.

Step 1: You will need to:

- Implement the `tryToPick` function. If the `PICKUP` action is legal in the current state, the fuction should return the new `SAR_State` and `PICKUP_COST` (defined in `params.py`). If not, it should return `(None, 0)`. Make sure to update the agent's status in the new state.
- Implement the `tryToPut` function. It should return the new `SAR_State` and cost if the `PUTDOWN` action is legal in the current state and `(None, 0)` otherwise. Make sure to update the agent's status in the new state.
- Extend the `moveActionCost` function to account for carrying a victim (in that case, it is the square of the obstacle cost)

Test this step using `python autograder.py -p 2 -s 1`. Also you'll find that doing regression tests[1] for Problem 1, Step 1, is going to be helpful to make sure you haven't introduced any incompatible changes to the code.

Step 2: You will need to extend the `costToGoal` function to create an admissible, informed heuristic for the `RETRIEVE_GOAL` mission. Hint: Think about what are the estimated costs for each of the different statuses that the agent can have (SEARCHING, VISITED, CARRYING, RETRIEVED).

Once you have successfully tested using `autograder.py` for both steps, your code should successfully pass `test.py` for the `RETRIEVE` goal and all the maps with just one victim.

Problem 3 (5 points)

In this problem, we extend the implementation to handle maps with multiple victims. To simplify things, though, your agent needs to visit or retrieve only one of the victims. However, your agent should choose the victim for which it can achieve the mission for the lowest cost.

Step 1: In problems 1 and 2, you could assume that there was only one element on the `victims` list. If you didn't make that assumption, you probably already solved this part of the problem, but if you did, now is the time to extend your implementations.

- Extend `tryToMove` to indicate that the status changes to `VISITED` if *any* of the victims have been visited.
- Extend the `tryToPick` function to determine if the action is legal in cases where there are multiple possible victims to retrieve.

Step 2: Here, you need to extend the `costToGoal` function for both the visit and retrieve goals.

- For the `VISIT_GOAL`, the heuristic should be admissible and relatively informed for the goal of visiting the closest victim
- For the `RETRIEVE_GOAL`, the heuristic should be admissible and relatively informed for the goal of retrieving the victim that costs the least to find **and** retrieve.

[1] Regression testing is a type of software testing that is used to confirm that new additions to a codebase has not affected the correctness or functionality of existing features. You want to make sure that what you implemented in this step doesn't create issues for the features you implemented for Problem 1.

As before, test using both `autograder.py` and `test.py`. For `test.py`, use maps that have multiple victims (with `v2` or `v4` in their names). Don't forget to do regression testing for problems 1 and 2, to make sure your solution is backwards-compatible.

Submission and Grading

You will only need to edit and submit `SAR.py` to Gradescope. You have an unlimited number of submissions, and your grade on the programming portion will be indicated by the Gradescope autograder output. You will **not** be graded on code style; however, we may manually grade portions of your code (e.g. questions that are more open-ended or have multiple implementations).

For all problems, points are awarded for (1) how close the cost of the path your solution finds is to the actual (simulated) cost; (2) how close the number of nodes searched is to the refsol for UCS and A* searches; and (3) and how close your heuristic is to the actual cost for a select subset of states. Running `test.py` will indicate you how well your solution performs with respect to the first two criteria and running `"autograder.py -s 2"` will provide indications about criterion 3. Note that the autograder on Gradescope may test more than what is provided locally.

Note that inadmissible heuristics will receive a penalty of 0.25 points unless they outperform the refsol (which is admissible).