Homework 3: MDP and RL Due Date: February 17th 11:59PM

Programming: Search under Uncertainty (25 points)

In this assignment, you will program a robot agent to search for victims in a building that has been hit by some natural disaster. Unlike the last assignment, actions will not result in deterministic changes in state. Now, when an action is taken, the robot has a chance to slip in another direction. Therefore, a simple plan is no longer sufficient to solve the problem. You will need to develop a *policy* for which actions to take at each state. You will do this by defining a Markov Decision Problem (MDP) and then solve the MDP to produce the policy, which can then be used for controlling the agent. 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."

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_MDP.py`: Creates a representation of the MDP problem for the search and rescue domain, and creates and applies a policy based off of it - **you will need to modify this file**;
- `MDP.py`: Solves an MDP given a list of states and actions, and dictionaries representing transition and cost functions - **you will need to modify this file**;
- `SAR.py`: defines the functions for the Search And Rescue (SAR) domain. Use your implementation from HW2;
- `map.py`: map representation for the search and rescue domain;
- `params.py`: defines constants and other parameters used in the code;
- `simulate.py`: simulates your policy 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 the first two problems in this assignment, you will make use of the `SAR_State` class from HW2. You should copy your version of `SAR.py` and overwrite the version in the download. If you have not completed that assignment, let us know and we can help provide you with the necessary code.

Recall that the `SAR_State` class has five attributes:

- `map` - the map associated with the problem you are trying to solve;
- `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;
- `goalType` - the type of mission (`VISITED_GOAL` or `RETRIEVE_GOAL`).

To solve the MDP, the agent must create a list of states and actions, as well as dictionaries representing the transition and reward functions. Solutions to problems 2, 3, and 4 involve implementing functions in the

`SAR_MDP` class in the file `SAR_MDP.py`. Solution to the problem 3 involves implementing functions in the `MDP` class in the file `MDP.py`.

The `SAR_MDP` class inherits from the `MDP` class in file `MDP.py`. The `MDP` class has 6 attributes:

- `states` - the list of states that make up the MDP;
- `actions` - the list of actions that make up the MDP;
- `transitions` - a dictionary of state-action transitions. The keys are (state, action) tuples and the values of the dictionary is a list of (state, probability) tuples;
- `rewards` - a dictionary of state rewards; The keys are states and the values are the immediate reward for being in that state;
- `values` - a dictionary of state values; The keys are states and the values are long-term rewards for that state ($V(s)$);
- `discount` - the discount factor γ used in the Bellman update;

Testing

We have provided `autograder.py` to help you debug and test your code, prior to submission. You invoke it using `python autograder.py <options>`. The options are:

- `-p or --problem`: which problem to test (1, 2, 3, 4);
- `-s or --step`: which step to test, if a problem has multiple steps;
- `-m or --map`: which map file to use. The choices are in the `maps` subdirectory;
- `-g or --goal`: which mission to perform. The choices are VISIT and RETRIEVE (defined in `params.py`);
- `-t or --test`: if included, overrides the `-p` and `-s` options and tests the whole MDP implementation, using the simulator and (optionally) the graphics;
- `--no-graphics`: by default, a graphics simulation of the agent performing the mission is shown. This option turns off the graphics. Useful only with the `-t` option;
- `--debugging`: prints out information about the status of the MDP (states, rewards, transitions generated). May be useful in tracing down bugs in your code.

If neither the `-p` or `-t` options are provided, if will default to testing all the problems (`-p`).

Important Note: We have designed `autograder.py` so that if your code passes all the autograder tests (including the `-t` option), then it should also pass the GradeScope autograder. Since this year involves significant changes to `autograder.py`, let us know if you do find that not to be the case, and we will try to fix it.

Problem 1 (10 Points)

For this problem you will implement several components of the `SAR_MDP` class.

Step 1: Implement the function `create_sink_state`, which takes a `map` object and the goal type (either the `VISIT_GOAL` or `RETRIEVE_GOAL`). The sink state represents where the agent transitions to after the goal has been achieved. This is to make the MDP able to have an infinite horizon (i.e., the agent can keep taking actions forever). The sink state should be of type `SAR_STATE` whose location (`Locn` object) is at (-1,-1) and whose status is the same as that of the goal state (i.e., either `VISITED` or `RETRIEVED`, depending on the goal type that is passed as a parameter).

Test using `autograder.py -p 1 -s 1`. Test using both goal types, using the `-g` option.

Step 2: Implement the function `generate_states`, which returns a **list** of states for the MDP. Each element of the list is an instance of type `SAR_State`, found in file `SAR.py` (see the Introduction section for a reminder of the elements needed to specify a `SAR_State`).

You should generate all feasible states that the agent can find itself in while performing a visit or retrieve mission. A feasible state is one that could occur as a result of legal moves starting from the initial state. The creation of some non-feasible states is permitted so long as the autograder doesn't timeout, but all feasible states must be created. An example of a non-feasible state is having the rescuer at a victim location, but with the status listed as SEARCHING.

Don't forget to add the sink state that was created in Step 1 to the list.

Hint: With proper coding, no copying is necessary for this assignment. Using copy may slow down your code significantly, and we recommend that you avoid doing so.

Test using `autograder.py -p 1 -s 2`. You can try using different goals, using the `-g` option, and different maps (from the `maps` directory), using the `-m` option.

Step 3: Implement the function `generate_rewards`, which returns a **dictionary** whose keys are states (of type `SAR_State`) and whose items are the respective **rewards** of said states. The SINK state should have a reward of 0. Goal states should have a reward of 100. All other states should have a reward equal to the ***negative*** obstruction cost of their respective map cell (negative because in HW2 we were minimizing cost but in the assignment we are maximizing reward).

Test using `autograder.py -p 1 -s 3`, using different goals and maps to test more fully. As problem 2 relies on your solution to problem 1, we suggest that you test your solutions thoroughly.

Problem 2 (10 Points)

In this part, you will implement the `generate_transitions` function in `SAR_MDP.py`. The function should return a `transitions` dictionary whose keys are state-action tuples and whose values are lists of state-probability tuples. For example, if `state<i>` is of type `SAR_State`, then

```
transitions[(state1, NORTH)] = [(state2, 0.2), (state3, 0.8)]
```

means that taking the action `NORTH` from `state1` will result in moving to `state2` with a 20% chance and `state3` with an 80% chance.

The rules for movement are as follows. To start, *cost* refers to the obstruction cost at the agent's location (function `moveActionCost` in class `SAR_State`). The probability of the agent **staying in place** is $0.01 * \sqrt{cost}$, the probability that the agent **slips to the left** is $0.05 * \sqrt{cost}$, and the same for **slipping to the right**. The probability of **moving forward** is **1 - (probability of slipping left) - (probability of slipping right) - (probability of staying in place)**.

If the agent is *unable* to move in some direction because it would hit a wall, the probability of slipping in that direction gets **added to the probability of staying in place**. This also applies when an agent attempts to move into a wall, the probability of moving forward should be added to the probability of staying in place.

There is also a fifth action, `DONE`, which should be used only for two occasions - staying in the sink state and moving to the sink state from a goal state. The `DONE` action is deterministic,.

You should not include infeasible state-action pairs in your dictionary, such as taking the `DONE` action in a non-goal state.

Hint: Using some of the functions from HW2 can significanly reduce the amount of code you have to write. In particular, it might be helpful to use the functions `canMove`, `tryToMove`, and `moveActionCost`.

Test using `autograder.py -p 2`. Try using different goals, using the `-g` option, and different maps (from the `maps` directory), using the `-m` option.

Problem 3 (3 points)

Now that the MDP is specified, you need code to solve the MDP. We have provided most of that - an implementation of both value and policy iteration - in file `MDP.py`. There are two functions that you need to implement, though.

Step 1: Implement the `expected_value` function in the file `MDP.py`. This function should compute the expected value of performing an action at a given state, given a set of transitions and of state values. Recall (see also the lecture slides) that the formula for expected value is:

$$\sum p(s'|a,s)V(s')$$

where $p(s'|a,s)$ is the transition probability to state s' when taking action a in state s, and $V(s')$ is the long-term value of state s'.

The parameters to `expected_value` are a `state`, an `action` to be performed in that state, `transitions`, which is a dictionary mapping `(state, action)` tuples to a list of `(state, probability)` tuples, and `values`, which is a dictionary mapping states to their current values `V(s)`. The function should return the expected value of performing the given action at the current state.

Test using `autograder.py -p 3 -s 1`. Since this is not a SAR-specific implementation, none of the other options are available.

Step 2: Implement the `bellman_update` function in the file `MDP.py`. This function should perform a Bellman update for performing the best action from a given state, given a set of immediate rewards, transitions, and state values. Recall (see also the lecture slides) that the formula for the Bellman update is:

$$r(s) + \gamma \max_a E[V(s)]$$

where $r(s)$ is the immediate reward of state s, γ is the discount factor, and $E[V(s)]$ is the expected value of taking action a in state s (which you implemented in step 1).

The parameters to `bellman_update` are a `state`, a list of available actions (note that not all actions are feasible from all states), `transitions`, which is a dictionary mapping `(state, action)` tuples to a list of `(state, probability)` tuples, `rewards`, which is a dictionary mapping `states` to immediate (local) reward, `cur_values`, which is a dictionary mapping states to their current values `V(s)`, and a `discount` factor. The function should return the Bellman update.

Test using `autograder.py -p 3 -s 2`. Since this is not a SAR-specific implementation, none of the other options are available.

Problem 4 (2 points)

Finally, you must write the `update_state` and `choose_action` functions in the `SAR_Executor` class in file `SAR_MDP.py`. The `SAR_Executor` class is used to maintain the current state of the agent and choose the best action based on the current state and the policy.

The `SAR_Executor` class has two attributes:

- **state** - the current state of the agent (of type `sAR_State`)
- **policy** - a dictionary that maps from `states` to `actions`

Step 1: The function `update_state` should update `self.state` to reflect the new state of the agent. The input parameters are the `action` that was performed and the new location (of type `Locn`) that resulted. The function should be very similar to the `act` function in `SAR_State.py`, although you need to account correctly for transitions to the sink state (`Locn(-1, -1)`. Feel free to use and/or adapt as much of the sub-functions that it calls as you want. The function has no return value.

Test using `autograder.py -p 4 -s 1`. Try using different goals, using the `-g` option, and different maps (from the `maps` directory), using the `-m` option.

Step 2: The function `choose_action` should return the action that the policy dictates the agent should make, given the agents current state (`self.state`) and policy (`self.policy`).

Test using `autograder.py -p 4 -s 2`. This is a relatively simple function to implement, so none of the other options have any effect.

Final Testing

You may want to test your complete implementation before submitting to Gradescope. You can do so using `autograder.py -t`, along with other options (such as `-m`, `-g`, and `--no-graphics`).

When testing the complete system, you can also specify the discount factor to use in solving the MDP, using the `--discount` option (the default uses a discount factor of 1). Try different discount factors to see how the policy changes. The effect is particularly noticeable for the medium map with multiple victims (`maps/medium_v2.map` and `maps/medium_v3.map`). Based on what you've learned in class about the effect that the discount factor has in preferring short-term reward over long-term reward, you should be able to explain to yourself why the different discount factors produce the various policies you observe, based on the understanding .

Submission

You will need to edit and submit `SAR_MDP.py` and `MDP.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).