Mini Project 3: Search and Rescue

Instructions

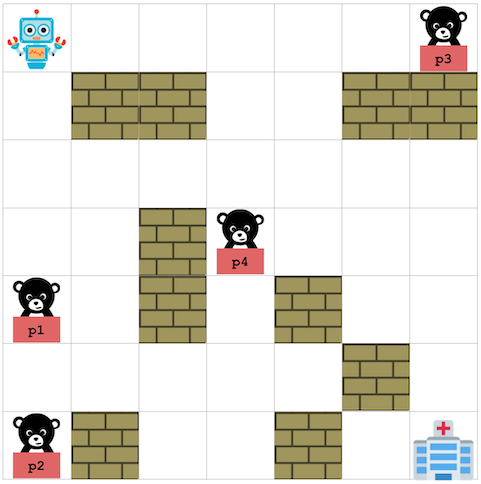
- The assignment is due by April 1, 2026 at midnight.
- There is a Google Colab with helper code located at `https://colab.research.google.com/drive/1ntPnZQB_kqz2juRDwE6ZUN-GjdhoBUa8?usp=sharing`. Open the notebook, go to "File" in the top left, and **save a copy**.
- Please submit to Gradescope (accessible via Canvas). There will be **two** assignments on Gradescope which you will need to submit to: one for your written answers, and one for code upload.
- Your written answers should be in a single PDF file, ideally produced with LaTeX. Your code should also be in a single PDF. To save the Colab to PDF, go to "File" in the top left, click "Print" and then save as PDF.
- The goal here is learning. You can work with other students, but be sure that, in the end, the solutions you submit were your work and that you understand them completely.
- You may use LLMs to help with this assignment; note, though, that we've tried these questions in several LLMs and they often produce incorrect answers, so the correctness is up to you!

Setup

We are going to continue the theme from mini-projects 1 and 2, but now we're going to reason about saving people from the fire.

Our "search and rescue" robot will be charged with navigating a sometimes dangerous grid to find and help people in need. We will first consider planning to navigate to, pick up, and drop off people at a hospital. This problem domain is deterministic; we will look at fully observed and partially observed cases.

In the search-and-rescue domain, we will first focus on a single planning problem, illustrated below:



This problem features four "people" (bears) with names p1, p2, p3, and p4. A robot, initialized in the top left corner, should navigate to each person, pick them up one by one, and deliver them to the hospital (bottom right).

- We always know the locations of the people, the robot, and the hospital.
- Some locations may have walls in them. You know about all the walls in advance, as well.
- There is also fire! And smoke! But, initially, you may not know which locations have fire and/or smoke. Nonetheless, you need to move around in the domain, make observations, do belief updates, make plans, and rescue bears.
- The way the environment works, whenever the robot enters a grid cell, it observes the true environment state of all the neighboring cells. Each environment cell contains exactly one of: wall (`W`), fire (`F`), smoke (`S`) or nothing (clear, `C`).
- The robot may safely enter any cell that is clear (`C`) or contains smoke (`S`).

Now we are going to ask you to put this all together into a planning and execution system.

Look at the code at the top of the colab notebook. `State` is a class with the following attributes:

- `state_map`: a 2D numpy array of characters `W`, `F`, `S`, `C`.
- `robot`: a (row, col) tuple representing the robot's location.
- `hospital`: a (row, col) tuple representing the hospital's location.
- `carrying`: the str name of a person being carried, or `None` if no person is being carried.
- `people`: a dict mapping str people names to (row, col) locations. If a person is being carried, they do not appear in this dict.

States have a couple of useful methods: `render` prints a representation of the state and `copy` does what you would expect. `get_safe_grid` returns a Boolean numpy array where `True` represents a safe space (not fire or wall).

Actions are strings. The following actions are defined:

- `up` / `down` / `left` / `right`: moves the robot. The robot cannot move into obstacles or off the map.
- `pickup-{person}`: if the robot is at the person, and if the robot is not already carrying someone, picks them up.
- `dropoff`: if the robot is carrying a person, they are dropped off at the robot's current location. *Allow for there being multiple dropoff locations, even though we will only dropoff at hospitals in this example.*

Please now take a moment to read the docstring for `SearchAndRescueProblem` to make sure that you understand the state and action spaces.

1 Search and rescue PDDL planner

Let's make a planner to solve a `SearchAndRescueProblem`. We will be using a Python PDDL planner called `pyperplan`, which you can get some practice with in HW5. Definitely do that first! The core function in this planner class is `get_plan`; it does the following:

1. Create PDDL domain and problem strings for search and rescue. The operators should work for any grid size, obstacles, people locations, and hospital location.
2. Invoke `run_planning` using the given `search_algo` search algorithm with the `heuristic` heuristic.
3. Convert the output of `run_planning` (pyperplan Operators) into actions that can be executed, via `execute_plan`.

For reference, `get_plan` takes ∼1–3 seconds to run with our implementation if using `gbf` search and `hff` heuristic. Be sure your implementation runs in less than 10 seconds (or the rest of this project will be painful.)

Notes:

- In this problem, you will need to construct somewhat complicated strings. We *strongly* encourage you to read about Python-3 f-strings which make this process much easier than the alternatives.
- You may find `state.render()` useful for debugging.

- We also highly recommend printing out the domain and problem after they have been created, and copying them into `http://editor.planning.domains` to check whether it's possible to find a plan. This editor can be helpful for syntax checking.
- The image above with the robot and the bears is a faithful depiction of the initial state. For example, the initial locations of the people are: `p1`: (4, 0), `p2`: (6, 0), `p3`: (0, 6), `p4`: (3, 3).
- One part of this problem that may be initially counterintuitive is the way that we'll represent locations in PDDL. In the problem, a location is a tuple of integers. PDDL does not support such representations—everything needs to be just an object with a string name. So to represent a location like (3, 5), we will make a string `l3-5` (where the first character there is a lowercase L), and we'll create an object with that name, of type `location`. We will also need a way to encode the fact that the robot can only move between adjacent locations in the grid. In Python, we can compare the numeric values of locations like (3, 5) and (3, 6) to see if they are neighbors. But in PDDL, all we have are the objects with string names, and we need to encode everything in terms of predicates. So, we will create a predicate `(conn ?v0 - location ?v1 - location ?v2 - direction)`, which says that location `?v0` is connected to location `?v1` in direction `?v2`. For example, `(conn l3-5 l3-6 right)` might appear in the initial state. We can then use these `conn` predicates in the preconditions of a `move` operator to encode the fact that the robot can only move between adjacent locations.
- We do not recommend modelling the hospital explicitly with special objects / types / predicates. Instead, the goal should be to deliver all people to the hospital, that is, `l6-6`. In words, the goal should be "person1 is at l6-6 and person2 is at l6-6 and person3 is at l6-6 and person4 is at l6-6."

1. (10 points) What solution do you get for this problem with only bears 1 and 2 in the world?

2 Inference

Now, we have to worry about fire. The agent has to avoid moving into any cells with fire (smoke is okay) but doesn't know where the fire is, in advance. We'll use a variation of the problem formulation that is similar to the one we used in MP1. We have implemented the code for you in `infer_unknown_values`. It has the same input/output spec (takes in a grid of character values and returns a grid, with some of the "U" characters changed to "S", "F", or "C" if they can be inferred from the available information.

It turns out that this problem was too big to be solved efficiently by `python-constraint` but Googles `or-tools` package works great!

3 Inference, planning, and execution.

Now it's time to bring the planning and the inference together. Study the procedure `agent_loop`, which takes as input:

- `initial_state`: a true state of the environment, which includes a state map, as well as the current locations of the hospital, the robot and the people and whether the robot is carrying someone. This state is used to build a "simulator" for our agent to interact with; the agent doesn't have direct access to this information, though, except through its observations.
- `initial_belief`: an initial belief state, which is like an environment state, except that the state map may include characters `U` indicating that the contents of a location is unknown.
- `policy`: a procedure that takes in a belief state and returns the next action to take; the action can be one of the original actions or `*Success*` (which means the goal has been achieved and all the people are at the hospital) or `*Failure*` (which means that the agent is certain that the goal is impossible to achieve.)
- `max_steps`: just a total number of steps to run the simulation to avoid infinite loops.

2. (5 points) What is the agent able to observe about the world at each step?

3. (5 points) What happens when the agent gets a new observation from the world? How does it update its belief?

   Now, we'll construct some policies for the agent.

3.1 Safe but not so smart

The agent is scared and decides to sacrifice the patients and save itself by running directly to the hospital. It at least takes observations into account as it moves. Make an agent that:

- Updates the belief state based on every observation using propositional inference (implement the `belief.update` method that gets called in `agent_loop`).
- Executes a policy that, given the currently updated belief, checks to see whether it can move safely into a square that is closer (in Manhattan distance) to the hospital. If so, it returns the action that would move it closer. If it reaches the hospital, it returns `*Success*`. If it cannot safely make a move (due to walls or fire) closer to the hospital, it waves its arms in anguish and returns `*Failure*`.
- We will say that the agent can move safely into a square if it is known to be clear or known to have smoke.

4. (10 points) What is the first plan constructed by your agent? Does it eventually save itself?

5. (5 points) In the specification for this behavior, we said it was okay to move into any square that was known to be clear or known to have smoke. That was actually too restrictive. What would have been a better condition? Would we need to modify our inference method to handle it? If so, how?

3.2 Safe and smart

We will continue to run the belief update method that you implemented above for the rest of these cases, so we only need to worry about the policy. This time the agent is going to be safe and smart, but maybe a bit too conservative.

- The first time the policy is called to generate an action, it should formulate, in PDDL, a planning problem to find a complete plan for moving people to the hospital that only traverses squares that are *known* to be clear or *known to be smoke-filled*, and returns the first step.
- On subsequent calls to the policy, it should just return the next step of the plan, if it remains safe. If that step is no longer safe, it should replan. If it can't find a safe plan, it should return `*Failure*`. At least it tried.

6. (10 points) What initial plan does your agent make? Does it eventually get the patient to the hospital?

3.3 Reckless

Now we are going to try to be more aggressive but still not step into the fire.

- The first time the policy is called to generate an action, it should formulate in PDDL a planning problem to find a very optimistic plan for moving people to the hospital that only traverses squares that are **not known to have** fire or walls, and returns the first step.
- On subsequent calls to the policy, if the next step in the plan is safe to execute given the updated belief, it should return that action. Otherwise, it should make a new plan!

7. (5 points) What initial plan does your agent make? Does the agent eventually get the patient to the hospital?

3.4 Safe and smart if possible, else reckless

Construct a new policy that is a useful combination of "safe and smart" and "reckless" strategies, which combines the best aspects of each.

8. (5 points) What initial plan does your agent make? Does the agent eventually get the patient to the hospital?

3.5 Looks before it leaps

A potentially more effective way to approach this problem is to plan in belief space. That sounds fancy, but actually can be relatively simple to do:

- In your PDDL formulation, instead of just having a fluent `(is-safe ?loc)`, we'll have two fluents: `(is-safe ?loc)` and `(is-unknown ?loc)`.
- The precondition for moving into a square should still be `(is-safe ?loc)`, which means it is known to be clear.
- You can add an operator that explicitly "looks" at a neighboring square; we'll assume that it's optimistic and so if you look at a square that was previously unknown, it is now known to be clear.
- Note that you can define a subclass of `SearchAndRescuePlanner` and redefine the `update_pddl_domain` method to add to the previous PDDL domain definition. You'll also need to change the `parse_plan` method to handle the new action and the `get_init_strs` method to handle the new facts.
- Make a replanning policy that plans in this belief-space formulation and executes its plan as long as it's safe, and replans otherwise.

9. (10 points) What initial plan does your agent make? Does the agent eventually get the patient to the hospital?

4 Questions

4.1 Formulation

10. (5 points) Pyperplan uses domain-independent heuristics. Suggest a domain-dependent heuristic that you would expect to work better, in the basic version of the problem where there is no fire. Explain whether or not it is admissible.

11. (5 points) In the policy loop, we do one observation before choosing our first action. Give an example scenario where omitting this step and using the reckless or two-phase planner would make an error.

12. (10 points) In the last three policies, we replan only when executing the next step would be unsafe. That might not be the best replanning strategy. Describe another strategy, give a concrete example of where it would do something differently than the current strategy, and say what the general trade-offs would be between that one and the current one.

4.2 Tests

Using the `test_policy` function defined in the Colab, run your policies in the following three scenarios. The initial beliefs and states are also defined in the Colab. In defining your policy, use an algorithm and heuristic that can find **optimal** paths. Hint: *lmcut* is an admissible heuristic.

- `belief_map = P1_B0`, `true_map = P1_G0`
- `belief_map = P1_B1`, `true_map = P1_G0`
- `belief_map = P2_B1`, `true_map = P2_G0`

4.2.1 Analysis

13. (10 points) In your answers below, don't count "look" as a step.

    1. How many steps does each policy take to solve each problem?
    2. Which method takes the fewest steps summed over all three problems?
    3. Which one would you choose if planning is very expensive compared to execution?
    4. Which one would you choose if execution is very expensive compared to planning? (In this case, what additional modifications might you make to your method?)

4.3 One at a time

14. (5 points) In the tests above, we had only one person to save. What happens to the running time if you add more people to save?

    When there are multiple people to save, an alternative strategy would be to:

    - arbitrarily pick a person to save
    - plan to pick them up and take them to the hospital
    - repeat, until all are at the hospital

    First discuss the pros and cons of this strategy in the completely observable case. Then discuss whether the considerations change when there is partial observability.