Mini Project 4: Planning Under Uncertainty

Instructions

- The assignment is due by April 15, 2026 at midnight.
- There is a Google Colab with helper code located at `https://colab.research.google.com/drive/1MdNKQij9xMr9SsB2uIr1QV6lfYyOGVDN`. Open the notebook, go to "File" in the top left, and **save a copy**.
- Please submit to Gradescope (accessible via Canvas). There will be **three** assignments on Gradescope which you will need to submit to: one for your written answers, one for code upload, and one for video submissions.
- 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.
- Some questions ask you to submit animations as **.mp4** videos (not .mov). To generate a video from the Colab, use:

  ```
  from matplotlib.animation import FFMpegWriter
  anim.save("filename.mp4", writer=FFMpegWriter(fps=5))
  ```

  Then download the file from Colab (click the folder icon in the left panel, find the file, right-click, and select "Download"). Upload the `.mp4` files to the **video submission** assignment on Gradescope. Please use the exact filenames specified in each question.
- 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!

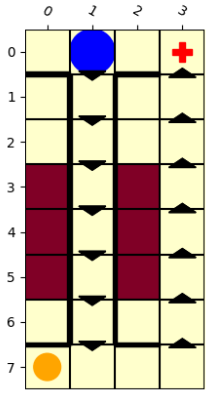
Setup

We are going to continue the theme from earlier mini-projects, but now we consider a *stochastic* version of the pickup and rescue problem in the form of an MDP. You will develop approximate online planning solutions using heuristic search and Monte-Carlo tree search.

The pickup and rescue problem

We consider a gridworld domain consisting of a robot, a patient, and a target hospital.

The robot's goal is to pick up the patient and deliver them to the hospital. The blue circle represents the robot, the orange circle represents the patient, and the red cross represents the hospital. The environment has **one-way roadblocks**, indicated by a flat triangle between two cells (the robot can only cross in one direction), and **two-way roadblocks**, indicated by thick black boundaries between two cells (the robot cannot cross in either direction).



The evolution of fire

Our grid now has fire. The fire starts at some initial locations and evolves through time. The fire grid at time t is independent of the robot and patient, and depends only on the fire grid at the previous time step. Fire completely ignores walls, roadblocks, and the hospital.

Given the fire grid at time t, the probabilities of fire at any two different cells at time $t+1$ are independent:

$$P(\mathbf{F}^{t+1} \mid \mathbf{F}^t) = \prod_{(i,j)\in\texttt{grid}} P\left(\mathbf{F}^{t+1}_{(i,j)} \mid F^t_{(i',j')\in\texttt{neighbors}((i,j))}\right),$$

where $\texttt{neighbors}((i,j)) = \{(i',j') \mid |i-i'| \le 1 \wedge |j-j'| \le 1\}$ is the 3×3 patch of cells centered at (i,j), including (i,j) itself.

The probability of fire in cell (i,j) at time $t+1$ is the weighted probability of its neighboring cells being on fire at time t, for a given fixed weight matrix $W \in \mathbb{R}^{3\times 3}$:

$$P\left(\mathbf{F}^{t+1}_{(i,j)} \mid F^t_{(i',j')\in\texttt{neighbors}((i,j))}\right) = \sum_{i',j'} W[i'-i+1,\ j'-j+1]\cdot F[i',j'],$$

where the weight matrix W is normalized so that its entries sum to one.

You might recognize that given the fire grid at time t (a matrix of 0–1 values), the probability of fire at time $t+1$ is the 2D convolution of the fire grid with the weights W.

Here is another way to understand the fire process:

- We start with some initial fire grid.
- At each time step t, for each cell, we randomly select a neighbor (including the current cell) with probability proportional to the weight matrix; the current cell at time $t+1$ gets the selected neighbor's fire value at time t. Neighbors outside the grid do not have fire.

Our approach

We adopt an **online-planning** approach, where at every step our agent:

1. plans according to the current state,
2. executes an action, and
3. observes a new state of the fire grid and replans (restarts from step 1).

Please take a moment to look at the code at the top of the Colab notebook. The `PickupProblem` class represents the deterministic pickup-and-rescue problem (without fire). The `FireProcess` class represents the stochastic fire process. The `FireMDP` class combines these into the full MDP.

1 Determinized planning

In our first approach, at each planning step we turn the MDP into a **deterministic min-cost path problem**:

- The state space no longer includes the fire grid; it only contains the state of the pickup-and-rescue problem (robot location, patient location, and whether the robot is carrying the patient).
- For a transition (s_t, a, s_{t+1}) at time t, we charge a cost of

$$c - \log(1 - P(\texttt{on\_fire}_{t+1}(s_{t+1}))),$$

  where c is a small constant cost for each step, and $P(\texttt{on\_fire}_t(s))$ is the marginal probability of the robot's destination cell being on fire at time t.
- We find the least-cost path to reach the patient and deliver them to the hospital—this path becomes our open-loop plan.

Once we have a min-cost path problem, we use A* search with a heuristic that ignores fire: the sum of the Manhattan distance from the robot to the patient plus the Manhattan distance from the patient to the hospital, scaled by the step cost c. When the robot is carrying the patient, the robot-to-patient distance is zero.

Complete the implementation of `DeterminizedFireMDP` in the Colab. In particular:

- Complete `fire_dist_at_time` to compute the marginal probability of each cell being on fire at time t, given the true fire state at time 0.
- Using `fire_dist_at_time`, complete `step_cost`.
- Complete the heuristic function `h` based on the description above.

Then complete `FireMDPDeterminizedAStarAgent` by implementing the `determinized_problem` method. You may build on your A* search implementation from HW1.

Hints:

- Before you code, try to derive the marginal probabilities of each grid cell being on fire at time t as an expression of the marginal probabilities at time $t-1$. More concretely, start small: consider a grid consisting of only two cells, X and Y, and assume W is uniform. Write $P(X_t = 1 \mid X_0, Y_0)$ as an expression of $P(X_{t-1} = 1 \mid X_0, Y_0)$ and $P(Y_{t-1} = 1 \mid X_0, Y_0)$.
- We formulate the A* state as (`robot_loc`, `patient_loc`, `time`) and use time as an index into a precomputed sequence of marginal fire probability grids. See `DeterminizedFireMDPState` and `DeterminizedFireMDP` in the Colab for details.

1.1 Formulation and experiments

1. i. Why aren't we approaching this problem with dynamic programming? What is the size of the state space?

   ii. Derive an expression for the marginal probability of a cell being on fire at time t as a function of the marginals at time $t-1$. What mathematical operation does this correspond to? What happens to the marginal fire probabilities as $t \to \infty$ for any initial fire configuration, and why?

   iii. The step cost for a transition at time t is $c - \log(1 - P(\texttt{on\_fire}_{t+1}(s_{t+1})))$. Show that minimizing the total path cost is related to maximizing the probability that the robot survives (does not step on fire) along the entire path. Why do we include the constant c?

   iv. Run your determinized agent in the MDP given by `get_problem("just_wait")` several times. Generate an animation of a **successful run** (the robot successfully rescues the patient) in which the fire does **not completely die out** when the robot moves down from the top row. Submit this animation as `just_wait.mp4`. If you cannot produce this animation after roughly 10 attempts, your implementation may be buggy. Describe and explain the agent's "waiting" behavior, connecting it to how the marginal fire probabilities evolve over time.

1.2 Lessons

2. Run the determinized agent in the MDP given by `get_problem("the_choice")`. In this environment, the agent faces a choice from its initial location:

   - **Go down**: take a shortcut through a one-way passage, but risk getting close to fire with no room to maneuver.
   - **Go right**: enter a larger room with more fire, but with space to move around and avoid it.

   Generate an animation of a successful run and submit it as `the_choice_determinized.mp4`.

   i. What choice does your determinized agent make? The determinized approximation replaces a stochastic fire process with marginal fire probabilities. What information about the fire process is lost in this approximation, and how does that explain the agent's choice?

   ii. The determinized planner produces an open-loop plan but replans after each step. Give a concrete scenario in which this approach can be arbitrarily bad compared to a closed-loop policy. Characterize the types of environments where you would expect the determinized approach to perform well versus poorly.

   iii. If planning time were very expensive compared to execution, would you still replan at every step? Describe a replanning strategy that reduces the number of replanning calls while still maintaining good performance. What are the trade-offs?

   iv. An alternative strategy would have been to use the most-likely-outcome determinization. Describe a concrete situation in which its behavior would be different from the method we've used here.

2 Monte-Carlo Tree Search

Now that we have seen a failure mode of our determinized planning agent, let's try to do better with closed-loop planning with MCTS!

With MCTS, we have more of a chance of hedging bets, so we might be inclined to go in directions where there are more options in case we get caught, even if the expected open-loop cost is higher.

We have provided you with an MCTS implementation, `run_mcts_search`, that works on both MDPs and POMDPs. Please take a look at the documentation of `run_mcts_search` to understand how to use it on MDPs, then implement an MCTS agent for MDPs.

3. Complete the implementation of `MCTSAgent`. Hint: You can pass in the `self.planning_horizon` to `run_mcts_search`, to handle both infinite-horizon problems (by receding-horizon planning) and finite-horizon problems.

   Similarly to in the previous problems, visualize the behavior of the MCTS planning agent in `the_choice` MDP and generate an animation of a successful run and submit it as `the_choice_mcts.mp4`.

   i. Compare the behavior of your `MCTSAgent` to the `FireMDPDeterminizedAStarAgent` on `the_choice`. Does the MCTS agent avoid the narrow one-way passage more consistently? If you do not see a difference in their behavior, try running both algorithms many times. Explain why simulating specific outcomes (Monte Carlo rollouts) captures the "risk" of being trapped better than the "average fire probability" used in the determinized A* approach.

   ii. The performance of MCTS is highly dependent on the number of rollouts. If you significantly limit the computation time (e.g., only 10 rollouts per step), MCTS often performs worse than A*. Why does A* handle low-computation budgets more gracefully than MCTS in this specific gridworld?

   iii. If the robot is in a "safe" area but the random rollout policy makes it walk into fire, how does this affect the value estimate of that state?