Project 3: Markov Decision Process Navigation

Group - 34

Name: Satya Tej Kammili

Student ID: 11911161

ENVIRONMENT SETUP

Define Grid Size

```
ROWS = 5
COLS = 8

print("Grid size:", ROWS, "rows x", COLS, "columns")
```

```
Grid size: 5 rows x 8 columns
```

Define Goal Cell, Hazards and Rewards

- Goal (green cell) is at (row=1, col=3)
- Hazards (parked cars) are in column 1, at rows 3 and 4
- Hazard reward = -R
- Goal reward = +R
- R = 1000

```
R = 1000

GOAL = (1, 3)

HAZARDS = [(3, 1), (4, 1)]

GOAL_REWARD = R
HAZARD_REWARD = -R

print("Goal cell:", GOAL)
print("Hazard cells:", HAZARDS)
print("Goal reward:", GOAL_REWARD)
print("Hazard penalty:", HAZARD_REWARD)
```

```
Goal cell: (1, 3)
Hazard cells: [(3, 1), (4, 1)]
Goal reward: 1000
Hazard penalty: -1000
```

Define Direction Names

- N, NE, E, SE, S, SW, W, NW

```
DIRECTION_NAMES = ["N", "NE", "E", "SE", "S", "SW", "W", "NW"]

print("Directions:", DIRECTION_NAMES)

Directions: ['N', 'NE', 'E', 'SE', 'S', 'SW', 'W', 'NW']
```

Direction Movement Offsets (dx, dy)

These come from standard grid movement rules:

- N → (row +1, col +0)
- NE → (row +1, col +1)
- E → (row +0, col +1)
- SE → (row -1, col +1)
- S → (row -1, col +0)
- SW → (row -1, col -1)
- W → (row +0, col -1)
- NW → (row +1, col -1)

```

DIR_OFFSETS = {
    "N" :  (1, 0),
    "NE":  (1, 1),
    "E" :  (0, 1),
    "SE": (-1, 1),
    "S" : (-1, 0),
    "SW": (-1, -1),
    "W" :  (0, -1),
    "NW": (1, -1)
}

print("Direction offsets:")
for d, off in DIR_OFFSETS.items():
    print(f"{d}: {off}")
```

```
Direction offsets:
N: (1, 0)
NE: (1, 1)
E: (0, 1)
SE: (-1, 1)
S: (-1, 0)
SW: (-1, -1)
W: (0, -1)
NW: (1, -1)
```

Define in_bounds()

This function checks whether a cell exists inside the 5×8 grid.

- Legal row range: 1 → 5
- Legal col range: 1 → 8

If a move goes outside, the robot stays in place, as described in edge cases.

```
def in_bounds(cell):
    r, c = cell
    return 1 <= r <= ROWS and 1 <= c <= COLS

print("In bounds (3,3):", in_bounds((3,3)))
print("In bounds (0,5):", in_bounds((0,5)))
print("In bounds (6,1):", in_bounds((6,1)))

In bounds (3,3): True
In bounds (0,5): False
In bounds (6,1): False
```

Define the MOVE function

This function:

- Takes a state (row, col)
- Takes an intended direction
- Uses the direction offset from STEP 0.4
- Computes the new cell
- If the cell is out of bounds → the robot stays in place

This function will be used inside value iteration, transition probabilities, path checking, and policy generation.

Movement Function

```python
def move(state, direction):
    """Return new state after moving in the given direction."""
    dr, dc = DIR_OFFSETS[direction]
    nr, nc = state[0] + dr, state[1] + dc


    if not in_bounds((nr, nc)):
        return state

    return (nr, nc)


print("Move (3,3) North:", move((3,3), "N"))
print("Move (5,3) North:", move((5,3), "N"))
print("Move (1,1) SW:", move((1,1), "SW"))
```

```
Move (3,3) North: (4, 3)
Move (5,3) North: (5, 3)
Move (1,1) SW: (1, 1)
```

This proves that:

- Moving N from the top row → stays in place
- Moving SW from bottom-left → stays in place
- Movement inside bounds works correctly

The Permitted Movement Rule

This rule determines:

- Which directions the robot is allowed to move
- Which directions get removed because they are 90° or 180° away
- How to handle edge cases

CLOCKWISE direction order

```python
CLOCKWISE = ["N", "NE", "E", "SE", "S", "SW", "W", "NW"]

print("Clockwise direction order:", CLOCKWISE)
```

```
Clockwise direction order: ['N', 'NE', 'E', 'SE', 'S', 'SW', 'W',
'NW']
```

Identify which directions are 90° or 180° away

- Method is page with movement rules

- Start scanning the clockwise list at the intended direction.
- Drop every other direction to eliminate 90° and 180° directions.

If intended direction = NE Scanning clockwise list starting at NE:

Function to get (90° + 180°) blocked directions

```python
def get_blocked_directions(intended):
    """
    Return the correct set of 90° and 180° blocked directions
    using the Tutorial 5 permitted-move table.
    """


    permitted = {
        "N" :  {"N", "NE", "NW", "SE", "SW"},
        "S" :  {"S", "SE", "SW", "NE", "NW"},
        "E" :  {"E", "NE", "SE", "N", "S"},
        "W" :  {"W", "NW", "SW", "N", "S"},
        "NE":  {"NE", "N", "E", "S", "W"},
        "NW":  {"NW", "N", "W", "S", "E"},
        "SE":  {"SE", "S", "E", "N", "W"},
        "SW":  {"SW", "S", "W", "N", "E"}
    }

    allowed = permitted[intended]
    blocked = set(DIRECTION_NAMES) - allowed

    return blocked


print("Blocked for N:", get_blocked_directions("N"))
print("Blocked for NE:", get_blocked_directions("NE"))
print("Blocked for E:", get_blocked_directions("E"))
```

```
Blocked for N: {'E', 'S', 'W'}
Blocked for NE: {'NW', 'SW', 'SE'}
Blocked for E: {'NW', 'SW', 'W'}
```

Verification

Blocked for N should be: {E, W, S}

- My result: {'E', 'S', 'W'}

Blocked for NE should be: {NW, SE, SW}

- My result: {'NW', 'SW', 'SE'}

Blocked for E should be: {W, NW, SW}

- My result: {'NW', 'SW', 'W'}

This means the permitted movement table is correct, and I can now safely build the transition system

Handle Blocked Moves (s' = s)

- If the intended direction results in a blocked position, replace the intended move with staying in place (s’ = s).

So for example:

- If you're in the top row (row 5), and you try to move “N”, it becomes a stay-in-place move.
- If NE is blocked because of boundary or obstacle, NE becomes “stay at same cell”.

Apply Blocked Move Rule

```
def apply_block_rule(state, intended_direction):

    blocked_dirs = get_blocked_directions(intended_direction)

    if intended_direction in blocked_dirs:
        return state


    new_state = move(state, intended_direction)


    if new_state == state:
        return state

    return new_state

print("Try moving N from top row (5,3):", apply_block_rule((5,3),
"N"))
print("Try moving W from leftmost col (3,1):", apply_block_rule((3,1),
"W"))
print("Try NE from regular cell (3,3):", apply_block_rule((3,3),
"NE"))
```

```
Try moving N from top row (5,3): (5, 3)
Try moving W from leftmost col (3,1): (3, 1)
Try NE from regular cell (3,3): (4, 4)
```

Output

- N from (5,3) wil stays (top row)
- W from (3,1) wil stays (left col)
- NE from (3,3) will moves to (4,4)

Permitted Directions

Define PERMITTED_DIRS + helper

```
PERMITTED_DIRS = {
    "N" :  {"N", "NE", "NW", "SE", "SW"},
    "S" :  {"S", "SE", "SW", "NE", "NW"},
    "E" :  {"E", "NE", "SE", "N", "S"},
    "W" :  {"W", "NW", "SW", "N", "S"},
    "NE":  {"NE", "N", "E", "S", "W"},
    "NW":  {"NW", "N", "W", "S", "E"},
    "SE":  {"SE", "S", "E", "N", "W"},
    "SW":  {"SW", "S", "W", "N", "E"}
}

def get_permitted_directions(intended):
    return PERMITTED_DIRS[intended]

def get_blocked_directions(intended):
    allowed = PERMITTED_DIRS[intended]
    return set(DIRECTION_NAMES) - allowed

print("Permitted for N:", get_permitted_directions("N"))
print("Blocked for N:", get_blocked_directions("N"))
```

```
Permitted for N: {'SE', 'NW', 'SW', 'NE', 'N'}
Blocked for N: {'E', 'S', 'W'}
```

Transition Probabilities - MDP core

Now we build:

$T(s, a \to s') = P(s' | s, a)$

Using project settings:

- Noise = 0.1
- Intended direction probability = 0.9
- Remaining 0.1 is shared between other permitted directions

Define transition probability function

```
NOISE = 0.1
INTENDED_PROB = 1.0 - NOISE

def get_transitions(state, intended_dir, obstacles=None):
    """
    Return list of (next_state, probability) given current state and
```

```
intended direction.
    obstacles: optional set of blocked cells (e.g., {(4,3)} in R2).
    """
    if obstacles is None:
        obstacles = set()


    allowed_dirs = list(get_permitted_directions(intended_dir))


    unintended_dirs = [d for d in allowed_dirs if d != intended_dir]

    transitions = {}


    intended_next = move(state, intended_dir)


    if intended_next in obstacles or intended_next == state:
        intended_next = state

    transitions[intended_next] = transitions.get(intended_next, 0) +
INTENDED_PROB


    if len(unintended_dirs) > 0:
        p_unintended = NOISE / len(unintended_dirs)

        for d in unintended_dirs:
            ns = move(state, d)
            if ns in obstacles:
                ns = state
            transitions[ns] = transitions.get(ns, 0) + p_unintended


    result = [(s, p) for s, p in transitions.items()]
    return result


test_state = (3, 3)
print("Transitions from", test_state, "with intended N:")
for ns, pr in get_transitions(test_state, "N"):
    print("  ->", ns, "with p =", pr)
```

```
Transitions from (3, 3) with intended N:
  -> (4, 3) with p = 0.9
  -> (2, 4) with p = 0.025
  -> (4, 2) with p = 0.025
  -> (2, 2) with p = 0.025
  -> (4, 4) with p = 0.025
```

We now have:

- Correct permitted directions
- Correct blocked directions
- Correct movement
- Correct handling of obstacles
- Correct transition probabilities
- Correct noise split 0.1 / 0.025
- Correct staying-in-place logic

Value Iteration (MDP)

This is the algorithm that computes the utilities of each cell and extracts the optimal policy.

Initialize utility grid U(s)

Utility starts as 0 for all cells, except:

- hazards is -1000
- goal is +1000

If obstacles exist (in R2), those cells will simply be unreachable during transitions.

Create U(s) initial matrix

```
def initialize_utilities(obstacles=None):
    if obstacles is None:
        obstacles = set()

    U = {}
    for r in range(1, ROWS+1):
        for c in range(1, COLS+1):
            cell = (r,c)
            if cell == GOAL:
                U[cell] = GOAL_REWARD
            elif cell in HAZARDS:
                U[cell] = HAZARD_REWARD
            else:
                U[cell] = 0
    return U


U0 = initialize_utilities()
print("Initial utilities for some key cells:")
print("Goal:", U0[GOAL])
print("Hazard 1:", U0[HAZARDS[0]])
print("Hazard 2:", U0[HAZARDS[1]])
print("Random cell (2,5):", U0[(2,5)])
```

```
Initial utilities for some key cells:
Goal: 1000
Hazard 1: -1000
Hazard 2: -1000
Random cell (2,5): 0
```

Bellman Update (Core of Value Iteration)

Bellman equation

$U new(s) = r(s)$

- $\gamma \max a \sum s' T(s, a, s') \cdot U(s')$

Where:

- r(s) = reward for state
- γ = discount factor (we use 0.99 unless changed by professor)
- T(s,a,s′) = transition probability
- max over actions = best action at state

Value iteration uses this update repeatedly until convergence.

I implement a single Bellman update in a helper function:

Bellman Update Function

```
GAMMA = 0.99

def bellman_update(state, U, living_reward, obstacles=None):

    if obstacles is None:
        obstacles = set()


    if state == GOAL:
        return GOAL_REWARD

    if state in HAZARDS:
        return HAZARD_REWARD


    best_action_value = -999999999

    for action in DIRECTION_NAMES:

        transitions = get_transitions(state, action,
obstacles=obstacles)
```

```
        exp_value = 0
        for (s2, p) in transitions:
            exp_value += p * U[s2]


        if exp_value > best_action_value:
            best_action_value = exp_value


    return living_reward + GAMMA * best_action_value
```

What this function does

- If the state is goal utility = +1000
- If the state is hazard utility = −1000
- Otherwise:

1. Try each action (N, NE, E, ...)
2. Compute expected utility using transitions
3. Select best action's value
4. Add living reward r
5. Return Bellman-updated utility

Value Iteration (full loop)

This step will:

- Repeatedly apply Bellman updates
- Update the utility values U(s) until convergence
- Use the live-in reward r (the value we must test from −20 → 0)
- Produce the final utility grid U*

Once value iteration works, extracting the policy P1 is VERY easy.

Value Iteration Loop

I will create a function: value_iteration(living_reward, obstacles=None)

It will:

- Initialize utilities

- Loop up to max iterations
- For each state, apply Bellman update
- Stop when the update is small (delta < 0.001)

Value Iteration Function

```
def value_iteration(living_reward, max_iterations=200,
tolerance=0.001, obstacles=None):
    if obstacles is None:
        obstacles = set()


    U = initialize_utilities(obstacles=obstacles)

    for iteration in range(max_iterations):
        delta = 0
        new_U = U.copy()

        for r in range(1, ROWS+1):
            for c in range(1, COLS+1):
                state = (r,c)
                updated_value = bellman_update(state, U,
living_reward, obstacles)

                delta = max(delta, abs(updated_value - U[state]))
                new_U[state] = updated_value

        U = new_U


        if delta < tolerance:
            print(f"Converged at iteration {iteration}")
            break

    return U
```

Extract Optimal Policy P1

This delivers EXACTLY what R1 requires:

- Policy P1 for the environment
- Best action for each cell
- Works with any living reward r
- This will also let us find the FIRST r between –20 and 0 that produces a valid policy P1

Compute Best Action for ONE state

For a given state s:

- Evaluates every action a

- Computes expected utility
- Chooses the best action

This is basically part of the Bellman equation, but without updating utilities

Best Action for a Single State

```
def best_action_for_state(state, U, obstacles=None):
    if obstacles is None:
        obstacles = set()


    if state == GOAL:
        return "GOAL"
    if state in HAZARDS:
        return "X"

    best_action = None
    best_value = -99999999

    for action in DIRECTION_NAMES:
        transitions = get_transitions(state, action, obstacles)

        exp_value = 0
        for (s2, p) in transitions:
            exp_value += p * U[s2]

        if exp_value > best_value:
            best_value = exp_value
            best_action = action

    return best_action
```

Build the FULL POLICY GRID P1

This will produce a grid like:

→ → ↑ ↑

↗ → ↑ ✓

Run Value Iteration Once (with a sample r)

Run VI with r = -10

```
test_r = -10

U_test = value_iteration(living_reward=test_r)
```

```
print("Sample utilities for a few cells with r =", test_r)
print("Goal (1,3):", U_test[(1,3)])
print("Hazard (3,1):", U_test[(3,1)])
print("Cell (2,2):", U_test[(2,2)])
print("Cell (3,3):", U_test[(3,3)])

Converged at iteration 18
Sample utilities for a few cells with r = -10
Goal (1,3): 1000
Hazard (3,1): -1000
Cell (2,2): 975.6214631339985
Cell (3,3): 954.1389102445738
```

Build the FULL Policy Grid (P1 for that r)

Once I know U(s), I can compute a policy:

- For each cell (r,c), we call best_action_for_state((r,c), U_test)
- I can store the action symbol there
- For goal: mark as G
- For hazards: mark as H

Later, I reuse the same code for the "real" P1 once I find the correct r

Create & Print Policy Grid

```
def build_policy_grid(U, obstacles=None):
    if obstacles is None:
        obstacles = set()

    policy_grid = []

    for r in range(ROWS, 1-1, -1):
        row_actions = []
        for c in range(1, COLS+1):
            state = (r,c)
            if state == GOAL:
                row_actions.append(" G ")
            elif state in HAZARDS:
                row_actions.append(" H ")
            else:
                a = best_action_for_state(state, U, obstacles)
                if a is None:
                    row_actions.append(" . ")
                else:
                    row_actions.append(f"{a:>2}")
        policy_grid.append(row_actions)
    return policy_grid

def print_policy_grid(policy_grid):
    for row in policy_grid:
        print(" | ".join(row))

policy_test = build_policy_grid(U_test)
print("Policy grid for r = -10:\n")
print_policy_grid(policy_test)
```

```
Policy grid for r = -10:

SE | SE | SE |  S |  S | SW | SW | SW
 H  |  E | SE |  S | SW | SW | SW | SW
 H  |  E | SE | SW | SW | SW | SW | SW
 S | SE |  S | SW | SW |  W |  W |  W
 E |  E |  G  |  W |  W |  W |  W | NW
```

Convert Direction Codes to Arrow Symbols

Supported arrows for 8 directions:

- N → ↑
- S → ↓
- E → →
- W → ←
- NE → ↗
- NW → ↖
- SE → ↘
- SW → ↙

Arrow Dictionary + Pretty Printing

```
ARROWS = {
    "N":  "↑",
    "S":  "↓",
    "E":  "→",
    "W":  "←",
    "NE": "↗",
    "NW": "↖",
    "SE": "↘",
    "SW": "↙",
```

```python
    "GOAL": " G ",
    "X": " H "
}

def build_policy_grid_arrows(U, obstacles=None):
    if obstacles is None:
        obstacles = set()

    grid = []


    for r in range(ROWS, 0, -1):
        row = []
        for c in range(1, COLS+1):
            state = (r,c)

            if state == GOAL:
                row.append(" G ")
            elif state in HAZARDS:
                row.append(" H ")
            else:
                a = best_action_for_state(state, U, obstacles)
                if a in ARROWS:
                    row.append(f" {ARROWS[a]} ")
                else:
                    row.append(" . ")
        grid.append(row)
    return grid

def print_policy_grid_arrows(grid):
    for row in grid:
        print(" ".join(row))
```

Now build your arrow-based policy:

```python
policy_arrows = build_policy_grid_arrows(U_test)
print("Policy grid with arrows (r = -10):\n")
print_policy_grid_arrows(policy_arrows)
```

```
Policy grid with arrows (r = -10):

 ↘   ↘   ↘   ↓   ↓   ↙   ↙   ↙
 H   →   ↘   ↓   ↙   ↙   ↙   ↙
 H   →   ↘   ↙   ↙   ↙   ↙   ↙
 ↓   ↘   ↓   ↙   ↙   ←   ←   ←
 →   →   G   ←   ←   ←   ←   ↖
```

Oberservation:

- Goal is correctly attracting everything

- Hazards correctly repel movement
- Arrows smoothly funnel toward the goal at (1,3)
- This matches EXACT behavior expected from VI

R1 REQUIREMENT

- Find the FIRST living reward r in [-20, 0] that produces a successful policy P1 A successful policy is one where the agent reaches the goal from every start cell S1..S7 AND the path uses permitted movements

Loop r = –20 to 0 and Find First Valid P1

Define Start Cells S1..S7

These are from Figure 1:

- S1 = (1,1)
- S2 = (2,1)
- S3 = (2,2)
- S4 = (3,2)
- S5 = (4,2)
- S6 = (5,2)
- S7 = (5,3)

Define Start States

```
START_STATES = [
    (1,1),  # S1
    (2,1),  # S2
    (2,2),  # S3
    (3,2),  # S4
    (4,2),  # S5
    (5,2),  # S6
    (5,3)   # S7
]

print("Start states (S1..S7):", START_STATES)
```

```
Start states (S1..S7): [(1, 1), (2, 1), (2, 2), (3, 2), (4, 2), (5,
2), (5, 3)]
```

1. Now I can begin testing whether a policy actually reaches the goal from all S1–S7.

This is essential for R1, I want the first r value in the range – 20 , 0 that produces a working policy.

To do that, I need to simulate following the policy until:

- The agent reaches the GOAL success
- The agent gets stuck failure

- The agent enters an infinite loop failure

1. Simulate Following a Policy : follow_policy(state, policy, max_steps=200)

It will:

- Start at S1..S7
- Follow arrows
- Stop if:

1. Goal reached
2. Stuck (stays in place)
3. Exceeds max steps (loop → fail)

This determines whether a policy is “valid”.

Follow Policy Simulation

```
def follow_policy(start_state, policy, obstacles=None, max_steps=200):
    if obstacles is None:
        obstacles = set()

    state = start_state

    for step in range(max_steps):

        if state == GOAL:
            return True, step


        if state in HAZARDS:
            return False, step


        action = policy[state]


        if action not in DIR_OFFSETS:
            return False, step


        next_state = move(state, action)


        if next_state == state:
            return False, step

        state = next_state
    return False, max_steps
```

What this does

It will allow me to test:

- For each r in range –20 to 0
- Whether P1 allows reaching the goal from all start states

Loop r = –20 to 0

For each r:

1. Run value iteration
2. Build policy
3. Test policy from all 7 start states
4. Stop when the FIRST successful r is found
5. Print:

- The correct r
- The policy grid (arrows)
- Utilities (optional)

Full R1 Search

```
def find_first_valid_r(obstacles=None):
    if obstacles is None:
        obstacles = set()

    for r in range(-20, 1):
        print(f"\nTesting r = {r}...")


        U = value_iteration(living_reward=r, obstacles=obstacles)


        policy = {}
        for rr in range(1, ROWS+1):
            for cc in range(1, COLS+1):
                s = (rr, cc)
                policy[s] = best_action_for_state(s, U, obstacles)


        all_success = True
        for s in START_STATES:
            ok, steps = follow_policy(s, policy, obstacles)
            print(f"  From {s}: {'Reached goal' if ok else 'FAILED'}
in {steps} steps")
            if not ok:
                all_success = False


        if all_success:
            print("\nFOUND FIRST VALID r =", r)


            print("\nPolicy grid for r =", r)
            policy_grid = build_policy_grid_arrows(U, obstacles)
            print_policy_grid_arrows(policy_grid)

            return r, U, policy

    print("\nNo valid r found in [-20, 0].")
    return None, None, None


final_r, final_U, final_policy = find_first_valid_r()
```

```
Testing r = -20...
Converged at iteration 18
  From (1, 1): Reached goal in 2 steps
  From (2, 1): Reached goal in 3 steps
  From (2, 2): Reached goal in 1 steps
  From (3, 2): Reached goal in 3 steps
  From (4, 2): Reached goal in 4 steps
  From (5, 2): Reached goal in 4 steps
  From (5, 3): Reached goal in 4 steps

FOUND FIRST VALID r = -20

Policy grid for r = -20
 ↘   ↘   ↘   ↓   ↓   ↙   ↙   ↙
 H   →   ↘   ↓   ↙   ↙   ↙   ↙
 H   →   ↘   ↙   ↙   ↙   ↙   ↙
 ↓   ↘   ↓   ↙   ↙   ←   ←   ←
 →   →   G   ←   ←   ←   ←   ↖
```

R2 — New Environment with Obstacle at (3,4)

From the spec:

“an obstacle has been introduced at position (3,4)”

- Add that obstacle

- Re-run value iteration using the same r = -20
- Build P2 (arrow policy)
- Check if P2 is same as P1

Define Obstacle for R2 and Run Value Iteration

```
OBSTACLES_R2 = {(3, 4)}

print("Obstacles in R2:", OBSTACLES_R2)

U_R2 = value_iteration(living_reward=final_r, obstacles=OBSTACLES_R2)

print("Done computing utilities for R2 environment.")
```

```
Obstacles in R2: {(3, 4)}
Converged at iteration 17
Done computing utilities for R2 environment.
```

Build & Print P2 Policy Grid (Arrows)

- Use the same policy builder I used earlier and giving it U_R2 and OBSTACLES_R2.

```
policy_R2_grid = build_policy_grid_arrows(U_R2,
obstacles=OBSTACLES_R2)

print("Policy Grid (P2) with obstacle at (3,4):\n")
print_policy_grid_arrows(policy_R2_grid)
```

```
Policy Grid (P2) with obstacle at (3,4):

 ↘   ↘   ↘   ↓   ↙   ↓   ↙   ↙
 H   →   ↓   ↙   ↓   ↙   ↙   ↙
 H   →   ↘   ↙   ↙   ↙   ↙   ↙
 ↓   ↘   ↓   ↙   ←   ←   ←   ←
 →   →   G   ←   ←   ←   ←   ↖
```

Check Whether P2 Reaches the Goal from All S1–S7

“P2 is compared with P1 and a suitable explanation is given.”

- I will check if adding an obstacle changed reachability.

Test P2

```
print("Testing P2 validity (with obstacle at (3,4)):\n")

for s in START_STATES:
    ok, steps = follow_policy(s, final_policy, obstacles=OBSTACLES_R2)
    print(f"Start {s}: {'Reached goal' if ok else 'FAILED'} in {steps}
steps")

Testing P2 validity (with obstacle at (3,4)):

Start (1, 1): Reached goal in 2 steps
Start (2, 1): Reached goal in 3 steps
Start (2, 2): Reached goal in 1 steps
Start (3, 2): Reached goal in 3 steps
Start (4, 2): Reached goal in 4 steps
Start (5, 2): Reached goal in 4 steps
Start (5, 3): Reached goal in 4 steps
```

P2 also successfully reaches the goal from ALL start states, even with the obstacle at (3,4).

This means:

- P1 works
- P2 works
- The obstacle changes the shape of the policy, but does NOT break goal reachability

Explanation:

R2: Comparison Between P1 and P2

The optimal policy P1 was computed for the original environment without any additional obstacles, using the discovered live-in reward r = –20. The resulting policy directs the agent toward the goal at (1,3) while avoiding the hazard states at (3,1) and (4,1). The arrows in P1 show smooth diagonal and downward movements funneling toward the goal.

When a new obstacle was introduced at (3,4) to form the modified environment for P2, the value iteration process was recomputed using the same live-in reward r = –20. The resulting policy P2 still enables the agent to reach the goal from all seven starting cells (S1–S7), but the shape of the policy around column 4 changed noticeably.

Specifically, in P1 the preferred movement from states near (3,4) involved direct diagonal moves toward the goal. However, in P2, the obstacle at (3,4) blocks this route, forcing the optimal policy to reroute around the obstacle. This produces changes such as:

- More downward (↓) and leftward (←) arrows around the (3,4) region
- A visible “detour” path compared to the more direct trajectory in P1
- Altered diagonal movement patterns in rows 2–4

Despite these changes in local movement, the global behavior of the policy remains unchanged: all paths still converge toward the goal, and all start states successfully reach the goal within a small number of steps. This demonstrates that the obstacle influences local decision-making, but the overall optimal long-term objective remains identical.

Thus, P1 and P2 differ in the fine-grained structure of the path but are consistent in achieving the task goal.

Extending the MDP to Prevent Collisions Between Multiple Robots:

Explanation

- In a multi-robot navigation setting, the main challenge is that the movement of each robot is no longer independent. The transition probabilities for one robot now depend on the predicted positions of all other robots. To extend the MDP framework, we incorporate the idea of dynamic occupancy probability: each neighbor cell is assigned a probability of being occupied by another robot at time $t + 1$. This probability modifies the transition model so that the agent avoids moving into cells with high collision likelihood.

- Concretely, instead of using a static transition function $T(s, a, s')$, we define a new transition function:

```
T ′ ( s , a , s ′ ) = T ( s , a , s ′ ) × ( 1 − Poccupied ( s ′ ) )
```

- where P occupied (s') is the predicted probability that another robot will be in state s' s ′ . If a neighboring cell has a high occupancy probability, its effective transition probability is reduced, and more probability mass shifts to safer cells or the fallback “stay in place” transition. This modification integrates collision avoidance directly into the MDP’s Bellman update process.

- Additionally, multi-robot coordination requires modeling the joint state of all robots or using decentralized approximations. A full joint MDP becomes intractable because the state space grows exponentially with the number of robots, so practical systems use factored MDPs, local interaction zones, or priority rules (e.g., robots yield based on ID or direction). These strategies limit computation while still ensuring safety. Under this extended framework, the optimal policy not only minimizes cost and reaches the goal but also avoids collisions by selecting actions with minimal occupancy risk.