Task 1

Planner Design

1. Design Decisions

- **MDP Parsing and Representation:**
    - The planner reads an MDP file formatted with state, action, transition, and reward information.
    - Terminal states and the MDP type (episodic or continuing) are explicitly parsed, allowing the planner to adjust its solving strategy accordingly.
- **Algorithm Choice:**
    - **Howard's Policy Iteration:**
        - The primary method used is Howard's Policy Iteration, where policy evaluation is performed exactly by solving a system of linear equations.
        - This exact evaluation helps in obtaining precise value functions.
    - **Linear Programming (LP) Fallback:**
        - For episodic MDPs ($\gamma = 1$) or in cases where the linear system is singular (due to cyclic structures), the planner falls back on an LP formulation using PuLP.
        - This approach provides robustness in scenarios where policy iteration may fail to converge.
- **Policy Evaluation Techniques:**
    - An exact policy evaluation function is implemented to solve the Bellman equation directly.
    - Additionally, an iterative evaluation method is provided, which can be useful for comparison or when the discount factor guarantees contraction.
- **policy_evaluation_exact & policy_evaluation:**

  policy_evaluation_exact(): This function sets up the Bellman equations as a linear system $A \cdot V = b$ and solves it exactly using a direct linear solver (i.e. np.linalg.solve). This gives an exact solution in one step, assuming the matrix is non-singular. It is typically more efficient when the system is well-behaved.

  policy_evaluation(): This function uses an iterative approach. It repeatedly updates the value function using the Bellman backup until the values converge (i.e., until the maximum change falls below a small threshold). This method is sometimes used when a direct solution is not feasible or when the discount factor guarantees contraction (ensuring convergence).

  Are both needed?

  In this implementation, they provide two alternative ways to evaluate a policy:
    - The **exact version** is used within the Howard's policy iteration (when possible) to quickly obtain the precise value function.
    - The **iterative version** can be used when an exact solution is not practical or for verification/comparison purposes.

2. Assumptions

- The input MDP file is correctly formatted and contains valid transitions, rewards, and terminal states.
- For continuing MDPs, the discount factor is less than 1, ensuring contraction in the value iteration.
- Episodic problems (with discount factor equal to 1) are more likely to lead to singular systems during exact policy evaluation, hence the fallback to LP.
- The state space is assumed to be fully described in the file, and each state's outgoing transitions (if any) are complete.

3. Observations

- **Singular Matrix Challenge:**
  - When solving the linear system for policy evaluation with $\gamma = 1$, singular matrix errors can occur in cyclic environments.
  - The fallback to LP resolves this by using a different mathematical formulation.
- **Robustness and Efficiency:**
  - Exact policy evaluation tends to converge faster and provides more precise values compared to iterative methods when applicable.
  - LP is particularly effective in episodic scenarios and ensures the planner produces a viable policy even in challenging cases.
- **Numerical Stability:**
  - Minor modifications (e.g., adding a small constant to the diagonal) can further improve the robustness of the exact solution without affecting the overall outcome.

4. Interesting Aspects

- **Algorithm Flexibility:**
  - Implementing both Howard's Policy Iteration and LP allowed exploration of trade-offs between speed and robustness in different MDP settings.
- **Fallback Mechanism:**
  - The decision to fallback to LP in the event of a singular matrix was a key insight, ensuring that the planner remains reliable across various types of MDPs.
- **Real-world Applicability:**
  - The approach taken reflects practical challenges encountered in reinforcement learning and decision-making problems, highlighting the importance of choosing the right solver based on problem structure

TASK 2:ENCODER AND DECODER

1.ENCODER

The purpose of this encoder is to translate a gridworld description into a formal Markov Decision Process (MDP) representation. This encoded MDP is used downstream by a planner to compute optimal policies. The gridworld uses specific characters to denote various entities (e.g., walls, goal, key, door, etc.), and the encoder maps this grid into a state space where each state represents a free cell along with the agent's orientation and whether it has collected the key.

---

2. Grid Parsing and Free Cell Identification

- **Parsing the Grid:**
  The parse_grid function reads a gridworld text file and converts it into a 2D list. Each row in the file is split into individual tokens (characters), which represent elements of the grid.
- **Building the Free Cells Dictionary:**
  The build_free_cells function iterates over the grid and assigns an index to each cell that is *not* a wall (W). This dictionary (free_cells) maps grid coordinates (i, j) to a unique integer index. Walls are excluded because the agent cannot traverse them.

---

3. State Encoding

- **State Space Structure:**
  Each free cell is expanded into multiple states based on:
  - **Orientation:** The agent can face one of four directions (up, right, down, left), coded as 0, 1, 2, 3.
  - **Key Status:** There are two possibilities: key not collected (0) or collected (1).
- **State ID Computation:**
  The state ID is calculated using the formula:

  $$\text{state\_id} = ((\text{cell\_index} \times 4) + \text{orientation}) \times 2 + \text{key\_flag}$$

  This formula ensures that each free cell (determined by its unique index) expands into exactly 8 states (4 orientations × 2 key statuses).

---

4. Action Encoding and Transition Formulation

- **Action Space:**
  There are four possible actions:
  1. **Move Forward (Action 0):**
     The agent attempts to move in the direction it faces. However, due to the icy floor, the agent slides 1, 2, or 3 steps with probabilities 0.5, 0.3, and 0.2, respectively.
     - **Sliding Dynamics:** The function get_sliding_outcomes calculates the valid outcomes in the given direction. If obstacles (walls or locked doors without a

key) block further movement, the probability mass is reassigned to the furthest reachable cell.

- **Key Pickup:** If the agent slides into a cell that contains the key (k), and the agent hasn't picked it up yet, the key flag is set to 1 in the next state.

2. **Turn Left (Action 1):**
   The agent turns left with a probability of 0.9 or performs a turn-around (180°) with probability 0.1.
3. **Turn Right (Action 2):**
   Similarly, the agent turns right with a 0.9 probability or a turn-around with a 0.1 probability.
4. **Turn Around (Action 3):**
   The agent turns 180° with a probability of 0.8, and with a 0.1 probability each, it may instead turn left or right.

- **Transition Recording:**
  For each state and for each action, transitions are recorded in the MDP file with a cost of -1 per step. The format for a transition is:
  transition s a s' -1 probability
  where s is the current state, a is the action taken, and s' is the resulting state after the action.

---

5. Terminal States and MDP File Output

- **Terminal States:**
  Terminal states correspond to any state where the underlying cell is a goal cell (g). For these states, no outgoing transitions are added.
- **Final MDP File Format:**
  The encoder writes the following to standard output:
  - The number of states and actions.
  - All transition lines detailing the dynamics of the MDP.
  - A line listing all terminal states (using the computed state IDs).
  - The MDP type (episodic) and the discount factor (1.0).

DECODER

<u>Decoder for the Icy Gridworld MDP</u>

1. Introduction

The decoder translates the output from the planner (the value–policy file) back into a set of optimal actions for snapshots of the gridworld. It reads test files containing multiple grid snapshots, determines the agent's state in each snapshot, and then uses the policy (generated by the planner) to compute and output the corresponding optimal action.

2. Design Decisions

- **Parsing Test Cases:**
  The decoder begins by reading a test file that contains several gridworld snapshots. Each test case is delimited by the marker "Testcase", and each snapshot consists of grid rows where each row is a line of space-separated characters.
- **Free Cell Ordering:**
  - The function build_free_cells_from_test is used to generate an ordering of free cells (i.e., every cell that is not a wall, denoted by 'W').
  - This ordering is critical because the state IDs (used in the MDP formulation) depend on the index of the free cell. The same ordering is assumed to be used by the encoder.
- **Agent and Key Detection:**
  - The find_agent function scans the grid for agent symbols ('>', '<', '^', 'v') and returns the cell coordinates along with the orientation (0 for up, 1 for right, 2 for down, 3 for left). If no agent symbol is found, it falls back to finding the start symbol ('s').
  - The function has_key_in_grid checks whether the key ('k') is still present. If it is, the agent is assumed not to have the key (key flag 0); otherwise, the agent is considered to have picked it up (key flag 1).
- **State ID Computation:**
  - Using the free cell ordering and the detected agent information, the decoder computes a state ID with the formula:

$$\underline{\text{state} = ((\text{cell\_index} \times 4) + \text{orientation}) \times 2 + \text{key\_flag}}$$

  - This computation mirrors the state encoding done in the encoder so that the planner's policy (an array indexed by these state IDs) can be applied directly.
- **Action Extraction:**
  - After computing the state for each test case, the decoder retrieves the optimal action from the policy list generated by the planner. The actions (represented as integers) are then printed space-separated.

3. Assumptions

- **Input File Format:**
  - The test file contains gridworld snapshots where rows are space-separated and each test case is delimited by the line "Testcase".
  - The gridworld format (i.e., use of symbols such as 'W', 's', 'k', 'g', etc.) is consistent with the encoder's expectations.
- **Consistency with Encoder:**

    - It is assumed that the ordering of free cells in the decoder matches that in the encoder, ensuring the state ID computed here correctly corresponds to the state ID used by the planner.

- **Agent Representation:**
    - The agent is represented by one of the symbols '>', '<', '^', 'v'. If none is present, the code defaults to the start symbol ('s') with a default orientation of 0 (up).

4. Observations

- **Robustness:**
    - The decoder includes a fallback mechanism (searching for 's' if no agent symbol is found) to ensure that every snapshot yields a valid state.
    - The functions are designed to ignore walls and only consider free cells, maintaining consistency with the encoder.
- **Modularity:**
    - The design separates concerns by having individual functions for parsing test cases, building the free cell dictionary, detecting the agent, and checking for the key. This makes the code easier to understand and maintain.
- **Mapping Consistency:**
    - The careful computation of state IDs ensures that the optimal action selected from the policy list is correctly mapped to the gridworld snapshot. This consistency is key for the decoder to output the correct actions.
- **Scalability:**
    - Given that gridworlds are small (with no more than 25 cells per side), the simple list-based methods for parsing and cell ordering are sufficient. For larger grids, more efficient data structures might be considered, but they are not necessary for the scope of this assignment.