Approach

- First, I take input for the grid, starting point, ending point, and grid size.
- I call a function and pass the grid, starting point, ending point, and grid size to ensure that neighbors do not go outside the grid.
- I use the Manhattan heuristic for calculating heuristic values.
- I write a function named getNeighbor, which takes a node as input and returns a list of its valid neighbors.
- I maintain a priority queue to ensure that the node with the lowest backward cost and heuristic value is explored first.
- I keep a parent dictionary to track backward traversal and find the actual path.
- I maintain a list called g_score to store the backward cost for each coordinate.
- I maintain a list called f_score to store the heuristic value for each coordinate.
- Then, I pop nodes one by one from the priority queue and check if the current node is the goal. If it is the goal, I return True and the path. To do this, I track the actual parent nodes using the parent dictionary. Since I traverse backward, I get the path from goal to start, so I reverse it before returning.
- If the popped node is not the goal, I find its neighbors:
    i) For each neighbor, I calculate its g_score and f_score and check if they are equal to or less than the parent's values. If they are, I update its parent, g_score, and f_score and push it into the priority queue.
- Finally, if the priority queue becomes empty and the goal condition has not been met, it means there is no valid path. So, I return False and None for the path.