Finding the Safest Path in a Grid: A Detailed Solution with Python Implementation and Analysis

Problem Explanation: Finding the Safest Path in a Grid

Problem Statement

Given a 2D grid of size (n \times n), where each cell can either contain a thief (1) or be empty (0), the task is to find the maximum safeness factor among all possible paths from the top-left corner to the bottom-right corner of the grid. The safeness factor of a path is the minimum Manhattan distance from any cell in the path to any thief in the grid.

Examples

- **Example 1**:
  - Input: `grid = [[1,0,0],[0,0,0],[0,0,1]]`
  - Output: `0`
  - Explanation: All paths from (0, 0) to (n-1, n-1) go through the thieves in cells (0, 0) and (n-1, n-1).
- **Example 2**:
  - Input: `grid = [[0,0,1],[0,0,0],[0,0,0]]`
  - Output: `2`
  - Explanation: The path has a safeness factor of 2 since the closest cell of the path to the thief at cell (0, 2) is cell (0, 0) with a distance of 2.
- **Example 3**:
  - Input: `grid = [[0,0,0,1],[0,0,0,0],[0,0,0,0],[1,0,0,0]]`
  - Output: `2`
  - Explanation: The path has a safeness factor of 2 with the closest cell to the thief being at a distance of 2.

Objective

To determine the maximum safeness factor of any path from the top-left corner to the bottom-right corner of the grid. The solution should be efficient given the constraint of (n \leq 400).

Approach to Solution

1. **Calculate Distance to Nearest Thief**:
   - Use BFS starting from all thief locations simultaneously.
   - This calculates the minimum distance to a thief for every cell in the grid.
2. **Find the Safest Path Using Priority Queue**:
   - Use a max-heap (priority queue) to explore paths from the start cell (0, 0) to the end cell (n-1, n-1).
   - The priority queue helps expand the path with the highest current safeness factor.

Approach:

1. **Calculate Distance to Nearest Thief**:
   - Use Breadth-First Search (BFS) to find the minimum distance from each cell to the nearest thief.
   - Initialize a queue with all thief cells.
   - Start BFS from each thief cell simultaneously, updating the distance to each cell as we traverse.
   - Store the minimum distance to each cell in a separate grid.
2. **Find the Safest Path Using Priority Queue**:
   - Use a max-heap (priority queue) to explore paths from the start cell to the end cell.

- Use a max heap (priority queue) to explore paths from the start cell to the end cell.
- Initialize the priority queue with the starting cell and its safeness factor (minimum distance to a thief).
- Pop the cell with the highest safeness factor from the priority queue.
- Explore its neighboring cells and push them into the priority queue with updated safeness factors.
- Repeat until reaching the end cell or exhausting all possible paths.

Pseudocode:

```
plaintext
Function maximumSafenessFactor(grid):
    Initialize a grid to store minimum distances to thieves
    Calculate minimum distances to thieves using BFS

    Initialize a priority queue with starting cell and its safeness
factor

    While priority queue is not empty:
        Pop cell with highest safeness factor from priority queue
        If cell is the destination, return its safeness factor
        Explore neighboring cells and update their safeness factors
        Push updated cells into priority queue

    If no path found, return -1
```

Step-by-Step Solution:

1. **Calculate Distance to Nearest Thief**:
   - Initialize an empty queue `q` and a grid `score` to store minimum distances.
   - Iterate through each cell in the grid:
     - If the cell contains a thief (1), set its distance to 0 and add it to the queue.
   - While the queue is not empty:
     - Pop a cell `(x, y)` from the queue.
     - Retrieve its current distance `s` from `score`.
     - Explore neighboring cells:
       - If a neighboring cell is within the grid boundaries and its distance is greater than `s + 1`, update its distance to `s + 1` and add it to the queue.
2. **Find the Safest Path Using Priority Queue**:
   - Initialize an empty priority queue `pq` with tuples `(safeness_factor, x, y)` where `safeness_factor` is the minimum distance to a thief.
   - Push the starting cell `(0, 0)` with its safeness factor to `pq`.
   - While `pq` is not empty:
     - Pop the cell `(x, y)` with the highest safeness factor from `pq`.
     - If `(x, y)` is the destination `(n-1, n-1)`, return its safeness factor.
     - Explore neighboring cells:
       - If a neighboring cell is within the grid boundaries and has not been visited:
         - Calculate its updated safeness factor as the minimum of its current safeness factor and the minimum distance to a thief.
         - Push the neighboring cell with its updated safeness factor to `pq`.
   - If no path is found, return -1.

Time Complexity Analysis:

1. **Calculate Distance to Nearest Thief (BFS)**:
   - Each cell in the grid may need to be visited once to find the minimum distance to a thief.
   - Time Complexity: (O(n^2)), where (n) is the size of the grid.
2. **Find the Safest Path Using Priority Queue**:
   - In the worst-case scenario, each cell may need to be explored and pushed into the priority queue.

- In the worst case scenario, each cell may need to be explored and pushed into the priority queue.
- Priority queue operations take (O(\log n)) time.
- Time Complexity: (O(n^2 \log n)).

Overall Time Complexity: [O(n^2) + O(n^2 \log n) = O(n^2 \log n)]

Recurrence Relation:

Let (T(n)) be the time taken to find the maximum safeness factor for a grid of size (n \times n). The recurrence relation can be expressed as:

[T(n) = O(n^2) + O(n^2 \log n) = O(n^2 \log n)]

Implementation

In [11]:

```
import heapq
from collections import deque

class Solution:
    def __init__(self):
        # Define movement directions: up, down, left, right
        self.roww = [0, 0, -1, 1]
        self.coll = [-1, 1, 0, 0]

    def bfs(self, grid, score, n):
        # Initialize a queue for BFS
        q = deque()

        # Iterate through each cell in the grid
        for i in range(n):
            for j in range(n):
                # If the cell contains a thief, mark its distance as 0
                if grid[i][j]:
                    score[i][j] = 0
                    # Add thief cell to the queue
                    q.append((i, j))

        # Perform BFS to find minimum distances
        while q:
            # Pop the current cell from the queue
            x, y = q.popleft()
            # Get the current distance
            s = score[x][y]

            # Explore neighboring cells
            for i in range(4):
                new_x = x + self.roww[i]
                new_y = y + self.coll[i]

                # Check if the neighboring cell is within bounds and has a shorter path
                if 0 <= new_x < n and 0 <= new_y < n and score[new_x][new_y] > s + 1:
                    # Update the distance to the neighboring cell
                    score[new_x][new_y] = s + 1
                    # Add the neighboring cell to the queue for further exploration
                    q.append((new_x, new_y))

    def maximumSafenessFactor(self, grid):
        # Get the size of the grid
        n = len(grid)
        # If either the start or end cell contains a thief, return 0
        if grid[0][0] or grid[n - 1][n - 1]:
            return 0

        # Initialize a grid to store minimum distances
        score = [[float('inf')] * n for _ in range(n)]
        # Calculate minimum distances using BFS
        self.bfs(grid, score, n)

        # Initialize a 2D array to keep track of visited cells
        vis = [[False] * n for _ in range(n)]
        # Initialize a priority queue with the starting cell
        pq = [(-score[0][0], 0, 0)]
        # Heapify the priority queue
        heapq.heapify(pq)
```

```python
        # Perform Dijkstra's algorithm using priority queue
        while pq:
            # Pop the cell with the highest safeness factor from the priority queue
            safe, x, y = heapq.heappop(pq)
            # Convert safeness factor to positive value
            safe = -safe

            # If the current cell is the destination, return its safeness factor
            if x == n - 1 and y == n - 1:
                return safe

            # Mark the current cell as visited
            vis[x][y] = True

            # Explore neighboring cells
            for i in range(4):
                new_x = x + self.roww[i]
                new_y = y + self.coll[i]

                # Check if the neighboring cell is within bounds and has not been visite
                if 0 <= new_x < n and 0 <= new_y < n and not vis[new_x][new_y]:
                    # Calculate the updated safeness factor
                    s = min(safe, score[new_x][new_y])
                    # Push the neighboring cell with its updated safeness factor to the
                    heapq.heappush(pq, (-s, new_x, new_y))
                    # Mark the neighboring cell as visited
                    vis[new_x][new_y] = True

        # If no path is found, return -1
        return -1
    # Author : Diya Maity
```

Testing

We will test the function with provided examples and additional test cases to ensure correctness and efficiency.

In [12]:

```python
test_cases = [
    ([[1,0,0],[0,0,0],[0,0,1]], 0),
    ([[0,0,1],[0,0,0],[0,0,0]], 2),
    ([[0,0,0,1],[0,0,0,0],[0,0,0,0],[1,0,0,0]],2),
    # Add more test cases as needed
]

solution = Solution()
for i, (grid, expected_output) in enumerate(test_cases, 1):
    result = solution.maximumSafenessFactor(grid)
    print(f"Case {i}: {'Passed' if result == expected_output else 'Failed'}")
# Author : Diya Maity
```

```
Case 1: Passed
Case 2: Passed
Case 3: Passed
```

Code Explanation:

1. **Initialization:**
    - In the `__init__` method, the `roww` and `coll` variables are initialized to represent the movement directions: up, down, left, and right.
2. **BFS (Calculate Distance to Nearest Thief):**
    - The `bfs` method performs a Breadth-First Search (BFS) to calculate the minimum distance from each cell to the nearest thief.
    - It initializes a queue `q` and iterates through each cell in the grid.
    - If a cell contains a thief, its distance is marked as 0, and it is added to the queue.
    - The BFS continues until the queue is empty, exploring neighboring cells and updating their distances if a shorter path is found.
3. **Find the Safest Path Using Priority Queue:**

- The `maximumSafenessFactor` method finds the maximum safeness factor using Dijkstra's algorithm with a priority queue.
- It initializes a priority queue `pq` with the starting cell's safeness factor.
- While the priority queue is not empty, it pops the cell with the highest safeness factor.
- If the popped cell is the destination, its safeness factor is returned.
- Neighboring cells are explored, and their safeness factors are updated based on the minimum distance to a thief.
- Updated cells are pushed into the priority queue for further exploration.
- The process continues until the destination cell is reached or no path is found.

4. **Edge Cases Handling:**

    - The code checks if either the start or end cell contains a thief. If so, it returns 0 as there is no safe path.
    - If no path is found, the method returns -1 to indicate that there is no path from the start to the end cell.

This solution efficiently calculates the maximum safeness factor of any path in the grid while considering all possible paths.

In conclusion, the provided solution efficiently tackles the problem of finding the maximum safeness factor in a grid by employing Breadth-First Search (BFS) and Dijkstra's algorithm with a priority queue.

- The BFS algorithm is used to calculate the minimum distance from each cell to the nearest thief, ensuring that every possible path's safeness factor can be determined accurately.
- Dijkstra's algorithm, implemented with a priority queue, then efficiently explores paths from the start