<https://leetcode.com/problems/minimum-obstacle-removal-to-reach-corner/description/>

You are given a 0-indexed 2D integer array grid of size m x n. Each cell has one of two values:

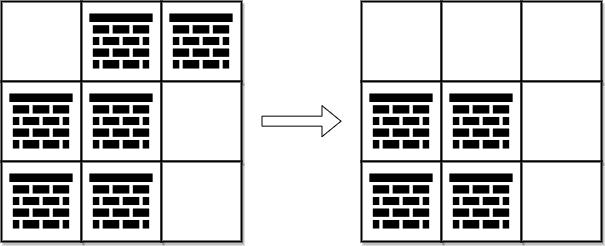
0 represents an **empty** cell,

1 represents an **obstacle** that may be removed.

You can move up, down, left, or right from and to an empty cell.

Return the **minimum** number of **obstacles** to **remove** so you can move from the upper left corner (0, 0) to the lower right corner (m - 1, n - 1).

**Example 1:**

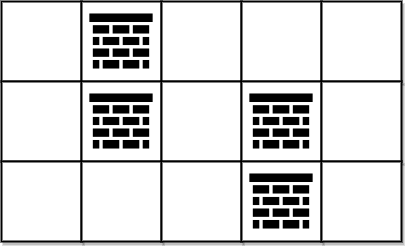


**Input:** grid = [[0,1,1],[1,1,0],[1,1,0]]

**Output:** 2

**Explanation:** We can remove the obstacles at (0, 1) and (0, 2) to create a path from (0, 0) to (2, 2).It can be shown that we need to remove at least 2 obstacles, so we return 2.Note that there may be other ways to remove 2 obstacles to create a path.

**Example 2:**



**Input:** grid = [[0,1,0,0,0],[0,1,0,1,0],[0,0,0,1,0]]

**Output:** 0

**Explanation:** We can move from (0, 0) to (2, 4) without removing any obstacles, so we return 0.

**Constraints:**

m == grid.length

n == grid[i].length

1 <= m, n <= 10^5

2 <= m \* n <= 10^5

grid[i][j] is either 0 **or** 1.

grid[0][0] == grid[m - 1][n - 1] == 0

**与L1293的区别还是挺大的，L1293是避让障碍上限为k，L2290是要求最小避让障碍数目**

**Attempt 1: 2024-06-03**

**Solution 1: Modified BFS + No need visited array (the biggest difference between L2290 & L1293) (180 min)**

**Round 1: TLE 32/54**

class Solution {

public int minimumObstacles(int[][] grid) {

int[] dx = new int[] {0,0,1,-1};

int[] dy = new int[] {1,-1,0,0};

int m = grid.length;

int n = grid[0].length;

int[][] eliminateCount = new int[m][n];

for(int i = 0; i < m; i++) {

for(int j = 0; j < n; j++) {

eliminateCount[i][j] = m \* n;

}

}

boolean[][] visited = new boolean[m][n];

Queue<int[]> q = new LinkedList<int[]>();

// Initially have to eliminate m \* n obstacles

//eliminateCount[0][0] = m \* n;

q.offer(new int[] {0, 0, 0});

visited[0][0] = true;

int distance = m \* n + 1;

while(!q.isEmpty()) {

int size = q.size();

for(int i = 0; i < size; i++) {

int[] cur = q.poll();

int x = cur[0];

int y = cur[1];

int currPathEliminateCount = cur[2];

if(x == m - 1 && y == n - 1) {

distance = Math.min(distance, currPathEliminateCount);

continue;

}

for(int j = 0; j < 4; j++) {

int new\_x = x + dx[j];

int new\_y = y + dy[j];

// Two conditions able to add into queue for next move:

// 1. Not visited before

// 2. Visited before on other path but encounter more obstacles

// than current path, which means the requires more removes

// than current path, if that happened, we will replace

// previous path with current path, also update the 2D array

// used to record

|| eliminateCount[new\_x][new\_y] > currPathEliminateCount)) {

/\* if(grid[new\_x][new\_y] == 0) {

// Since current cell value is 0 not consuming chance

eliminateCount[new\_x][new\_y] = currPathEliminateCount;

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

} else if(grid[new\_x][new\_y] == 1 && eliminateCount[new\_x][new\_y] > currPathEliminateCount) {

// Since current cell value is 1 consuming 1 more remove

eliminateCount[new\_x][new\_y] = currPathEliminateCount + 1;

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

}\*/

eliminateCount[new\_x][new\_y] = currPathEliminateCount + grid[new\_x][new\_y];

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

visited[new\_x][new\_y] = true;

}

}

}

//distance++;

}

return distance;

}

}

**Round 2: Removing the redundant 'visited' array, because eventually we have situation need to revisit 'visited' cells, which differ to L1293 where 'visited' array is mandatory, still TLE 32/54**

class Solution {

public int minimumObstacles(int[][] grid) {

int[] dx = new int[] {0,0,1,-1};

int[] dy = new int[] {1,-1,0,0};

int m = grid.length;

int n = grid[0].length;

int[][] eliminateCount = new int[m][n];

for(int i = 0; i < m; i++) {

for(int j = 0; j < n; j++) {

eliminateCount[i][j] = m \* n;

}

}

//boolean[][] visited = new boolean[m][n];

Queue<int[]> q = new LinkedList<int[]>();

// Initially have to eliminate m \* n obstacles

//eliminateCount[0][0] = m \* n;

q.offer(new int[] {0, 0, 0});

//visited[0][0] = true;

int distance = m \* n + 1;

while(!q.isEmpty()) {

int size = q.size();

for(int i = 0; i < size; i++) {

int[] cur = q.poll();

int x = cur[0];

int y = cur[1];

int currPathEliminateCount = cur[2];

if(x == m - 1 && y == n - 1) {

distance = Math.min(distance, currPathEliminateCount);

continue;

}

for(int j = 0; j < 4; j++) {

int new\_x = x + dx[j];

int new\_y = y + dy[j];

// Two conditions able to add into queue for next move:

// 1. Not visited before

// 2. Visited before on other path but encounter more obstacles

// than current path, which means the requires more removes

// than current path, if that happened, we will replace

// previous path with current path, also update the 2D array

// used to record

// || eliminateCount[new\_x][new\_y] > currPathEliminateCount)) {

&& eliminateCount[new\_x][new\_y] > currPathEliminateCount) {

/\*

if(grid[new\_x][new\_y] == 0) {

// Since current cell value is 0 not consuming chance

eliminateCount[new\_x][new\_y] = currPathEliminateCount;

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

} else if(grid[new\_x][new\_y] == 1 && eliminateCount[new\_x][new\_y] > currPathEliminateCount) {

// Since current cell value is 1 consuming 1 more remove

eliminateCount[new\_x][new\_y] = currPathEliminateCount + 1;

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

}\*/

eliminateCount[new\_x][new\_y] = currPathEliminateCount + grid[new\_x][new\_y];

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

//visited[new\_x][new\_y] = true;

}

}

}

//distance++;

}

return distance;

}

}

**Round 3: Removing the redundant 'visited' array, because eventually we have situation need to revisit 'visited' cells, which differ to L1293 where 'visited' array is mandatory, adding additional prune condition as "if(currPathEliminateCount > eliminateCount[x][y]) {continue;}" to skip faster, still TLE 51/54**

class Solution {

public int minimumObstacles(int[][] grid) {

int[] dx = new int[] {0,0,1,-1};

int[] dy = new int[] {1,-1,0,0};

int m = grid.length;

int n = grid[0].length;

int[][] eliminateCount = new int[m][n];

for(int i = 0; i < m; i++) {

for(int j = 0; j < n; j++) {

eliminateCount[i][j] = m \* n;

}

}

//boolean[][] visited = new boolean[m][n];

Queue<int[]> q = new LinkedList<int[]>();

// Initially have to eliminate m \* n obstacles

//eliminateCount[0][0] = m \* n;

q.offer(new int[] {0, 0, 0});

//visited[0][0] = true;

int distance = m \* n + 1;

while(!q.isEmpty()) {

int size = q.size();

for(int i = 0; i < size; i++) {

int[] cur = q.poll();

int x = cur[0];

int y = cur[1];

int currPathEliminateCount = cur[2];

if(x == m - 1 && y == n - 1) {

distance = Math.min(distance, currPathEliminateCount);

continue;

}

if(currPathEliminateCount > eliminateCount[x][y]) {

continue;

}

for(int j = 0; j < 4; j++) {

int new\_x = x + dx[j];

int new\_y = y + dy[j];

// Two conditions able to add into queue for next move:

// 1. Not visited before

// 2. Visited before on other path but encounter more obstacles

// than current path, which means the requires more removes

// than current path, if that happened, we will replace

// previous path with current path, also update the 2D array

// used to record

// || eliminateCount[new\_x][new\_y] > currPathEliminateCount)) {

&& eliminateCount[new\_x][new\_y] > currPathEliminateCount) {

/\*

if(grid[new\_x][new\_y] == 0) {

// Since current cell value is 0 not consuming chance

eliminateCount[new\_x][new\_y] = currPathEliminateCount;

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

} else if(grid[new\_x][new\_y] == 1 && eliminateCount[new\_x][new\_y] > currPathEliminateCount) {

// Since current cell value is 1 consuming 1 more remove

eliminateCount[new\_x][new\_y] = currPathEliminateCount + 1;

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

}\*/

eliminateCount[new\_x][new\_y] = currPathEliminateCount + grid[new\_x][new\_y];

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

//visited[new\_x][new\_y] = true;

}

}

}

//distance++;

}

return distance;

}

}

**Round 4: Removing the redundant 'visited' array, because eventually we have situation need to revisit 'visited' cells, which differ to L1293 where 'visited' array is mandatory, adding additional prune condition as "if(currPathEliminateCount > eliminateCount[x][y]) {continue;}" to skip faster, and change condition from "eliminateCount[new\_x][new\_y] > currPathEliminateCount" to "eliminateCount[new\_x][new\_y] > currPathEliminateCount + grid[new\_x][new\_y]", finally accept**

class Solution {

public int minimumObstacles(int[][] grid) {

int[] dx = new int[] {0,0,1,-1};

int[] dy = new int[] {1,-1,0,0};

int m = grid.length;

int n = grid[0].length;

int[][] eliminateCount = new int[m][n];

for(int i = 0; i < m; i++) {

for(int j = 0; j < n; j++) {

eliminateCount[i][j] = m \* n;

}

}

//boolean[][] visited = new boolean[m][n];

Queue<int[]> q = new LinkedList<int[]>();

// Initially have to eliminate m \* n obstacles

//eliminateCount[0][0] = m \* n;

q.offer(new int[] {0, 0, 0});

//visited[0][0] = true;

int distance = m \* n + 1;

while(!q.isEmpty()) {

int size = q.size();

for(int i = 0; i < size; i++) {

int[] cur = q.poll();

int x = cur[0];

int y = cur[1];

int currPathEliminateCount = cur[2];

if(x == m - 1 && y == n - 1) {

distance = Math.min(distance, currPathEliminateCount);

continue;

}

if(currPathEliminateCount > eliminateCount[x][y]) {

continue;

}

for(int j = 0; j < 4; j++) {

int new\_x = x + dx[j];

int new\_y = y + dy[j];

// Two conditions able to add into queue for next move:

// 1. Not visited before

// 2. Visited before on other path but encounter more obstacles

// than current path, which means the requires more removes

// than current path, if that happened, we will replace

// previous path with current path, also update the 2D array

// used to record

// || eliminateCount[new\_x][new\_y] > currPathEliminateCount)) {

&& eliminateCount[new\_x][new\_y] > currPathEliminateCount + grid[new\_x][new\_y]) {

/\*

if(grid[new\_x][new\_y] == 0) {

// Since current cell value is 0 not consuming chance

eliminateCount[new\_x][new\_y] = currPathEliminateCount;

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

} else if(grid[new\_x][new\_y] == 1 && eliminateCount[new\_x][new\_y] > currPathEliminateCount) {

// Since current cell value is 1 consuming 1 more remove

eliminateCount[new\_x][new\_y] = currPathEliminateCount + 1;

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

}\*/

eliminateCount[new\_x][new\_y] = currPathEliminateCount + grid[new\_x][new\_y];

q.offer(new int[] {new\_x, new\_y, eliminateCount[new\_x][new\_y]});

//visited[new\_x][new\_y] = true;

}

}

}

//distance++;

}

return distance;

}

}

**Solution 2: Dijkstra (60 min)**

**Style 1:**

class Solution {

    public int minimumObstacles(int[][] grid) {

        int[] dx = new int[]{0,0,1,-1};

        int[] dy = new int[]{1,-1,0,0};

        int m = grid.length;

        int n = grid[0].length;

        // Initialize dist with Integer.MAX\_VALUE, and use minObstaclesRemoveCount[i][j]

        // to indicate the currently minimum obstacles need to remove to reach (i, j);

        int[][] minObstaclesRemoveCount = new int[m][n];

        for(int[] a : minObstaclesRemoveCount) {

            Arrays.fill(a, Integer.MAX\_VALUE);

        }

        minObstaclesRemoveCount[0][0] = grid[0][0];

        // min-heap storing {i, j, # obstacles eliminated}, sorted by # of obstacles eliminated

        // Similar strategy as L743.Network Delay Time

        PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> a[2] - b[2]);

        minPQ.offer(new int[]{0, 0, minObstaclesRemoveCount[0][0]});

        // Using Dijkstra algorithm with PriorityQueue, no 'visited' array required

        while(!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

            int x = cur[0];

            int y = cur[1];

            int curObstaclesRemoveCount = cur[2];

            if(x == m - 1 && y == n - 1) {

                return curObstaclesRemoveCount;

            }

            for(int i = 0; i < 4; i++) {

                int new\_x = x + dx[i];

                int new\_y = y + dy[i];

                // Continue if we have more optimal result:

                // For {new\_x, new\_y} the new path (curObstaclesRemoveCount + grid[new\_x][new\_y])

                // remove less obstacles than old path (minObstaclesRemoveCount[new\_x][new\_y]),

                // then we update old path to new path, which means addding the node into minPQ

                && curObstaclesRemoveCount + grid[new\_x][new\_y] < minObstaclesRemoveCount[new\_x][new\_y]) {

                    minObstaclesRemoveCount[new\_x][new\_y] = curObstaclesRemoveCount + grid[new\_x][new\_y];

                    minPQ.offer(new int[]{new\_x, new\_y, minObstaclesRemoveCount[new\_x][new\_y]});

                }

            }

        }

        return minObstaclesRemoveCount[m - 1][n - 1];

    }

}

**Style 2: From ChatGPT**

class Solution {

    public int minimumObstacles(int[][] grid) {

        int[] dx = new int[]{0,0,1,-1};

        int[] dy = new int[]{1,-1,0,0};

        int m = grid.length;

        int n = grid[0].length;

        // Initialize dist with Integer.MAX\_VALUE, and use minObstaclesRemoveCount[i][j]

        // to indicate the currently minimum obstacles need to remove to reach (i, j);

        int[][] minObstaclesRemoveCount = new int[m][n];

        for(int[] a : minObstaclesRemoveCount) {

            Arrays.fill(a, Integer.MAX\_VALUE);

        }

        minObstaclesRemoveCount[0][0] = grid[0][0];

        // min-heap storing {i, j, # obstacles eliminated}, sorted by # of obstacles eliminated

        // Similar strategy as L743.Network Delay Time

        PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> a[2] - b[2]);

        minPQ.offer(new int[]{0, 0, minObstaclesRemoveCount[0][0]});

        // Using Dijkstra algorithm with PriorityQueue, no 'visited' array required

        while(!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

            int x = cur[0];

            int y = cur[1];

            int curObstaclesRemoveCount = cur[2];

            if(x == m - 1 && y == n - 1) {

                return curObstaclesRemoveCount;

            }

            for(int i = 0; i < 4; i++) {

                int new\_x = x + dx[i];

                int new\_y = y + dy[i];

                // Continue if we have more optimal result:

                // For {new\_x, new\_y} the new path (curObstaclesRemoveCount + grid[new\_x][new\_y])

                // remove less obstacles than old path (minObstaclesRemoveCount[new\_x][new\_y]),

                // then we update old path to new path, which means addding the node into minPQ

                && curObstaclesRemoveCount + grid[new\_x][new\_y] < minObstaclesRemoveCount[new\_x][new\_y]) {

                    minObstaclesRemoveCount[new\_x][new\_y] = curObstaclesRemoveCount + grid[new\_x][new\_y];

                    minPQ.offer(new int[]{new\_x, new\_y, minObstaclesRemoveCount[new\_x][new\_y]});

                }

            }

        }

        return - 1;

    }

}

**Refer to ChatGPT**

Leetcode problem 2290, "Minimum Obstacle Removal to Reach Corner," can be solved using Dijkstra's algorithm. In this problem, we need to find the minimum number of obstacles that need to be removed to get from the top-left corner to the bottom-right corner of a grid.

**Problem Explanation**

Grid: The grid consists of cells with values 0 (empty cell) and 1 (obstacle).

Objective: Find the path from the top-left corner (0, 0) to the bottom-right corner (n-1, m-1) that requires removing the fewest obstacles.

Moves: You can move up, down, left, or right.

**Approach**

We'll use Dijkstra's algorithm where the priority queue will manage states based on the number of obstacles removed. Each state is represented as (row, col, obstacles\_removed).

**Steps**

Initialization:

Use a priority queue to always expand the state with the fewest obstacles removed.

Use a 2D array to track the minimum number of obstacles removed to reach each cell.

Priority Queue:

Always expand the node with the smallest number of obstacles removed first.

For each neighboring cell, calculate the new number of obstacles removed and update if it is smaller than the known value.

Graph Exploration:

From each cell, consider moving to the neighboring cells (up, down, left, right). If moving to a cell with an obstacle, increase the count of obstacles removed.

Termination:

The algorithm terminates when the target cell (n-1, m-1) is reached.

**Java Implementation**

Here is the Java code to implement this approach:

class Solution {

    static class State implements Comparable<State> {

        int row, col, obstaclesRemoved;

        public State(int row, int col, int obstaclesRemoved) {

            this.row = row;

            this.col = col;

            this.obstaclesRemoved = obstaclesRemoved;

        }

        @Override

        public int compareTo(State other) {

            return Integer.compare(this.obstaclesRemoved, other.obstaclesRemoved);

        }

    }

    public int minimumObstacles(int[][] grid) {

        int rows = grid.length;

        int cols = grid[0].length;

        int[][] dist = new int[rows][cols];

        for (int[] row : dist) {

            Arrays.fill(row, Integer.MAX\_VALUE);

        }

        PriorityQueue<State> pq = new PriorityQueue<>();

        pq.add(new State(0, 0, 0));

        dist[0][0] = 0;

        int[] dr = {-1, 1, 0, 0};

        int[] dc = {0, 0, -1, 1};

        while (!pq.isEmpty()) {

            State current = pq.poll();

            if (current.row == rows - 1 && current.col == cols - 1) {

                return current.obstaclesRemoved;

            }

            for (int i = 0; i < 4; i++) {

                int newRow = current.row + dr[i];

                int newCol = current.col + dc[i];

                    int newObstaclesRemoved = current.obstaclesRemoved + grid[newRow][newCol];

                    if (newObstaclesRemoved < dist[newRow][newCol]) {

                        dist[newRow][newCol] = newObstaclesRemoved;

                        pq.add(new State(newRow, newCol, newObstaclesRemoved));

                    }

                }

            }

        }

        return -1;  // In case the target is not reachable (which should not happen)

    }

}

**Key Points:**

State: Each state in the priority queue includes the row, column, and number of obstacles removed.

Priority Queue: The priority queue ensures that the state with the fewest obstacles removed is processed first.

Distance Array: The 2D distance array keeps track of the minimum number of obstacles removed to reach each cell.

**Benefits:**

Optimal Path: Dijkstra’s algorithm ensures the path with the minimum number of obstacles removed is found.

Efficiency: The priority queue and distance array help manage the exploration efficiently, avoiding unnecessary revisits to states that have already been processed optimally.

By using this modified Dijkstra’s approach, the problem is efficiently solved while ensuring the minimum number of obstacles are removed to reach the target cell.

**Solution 3: 0-1 BFS (60 min)**

class Solution {

public int minimumObstacles(int[][] grid) {

// Get the dimensions of the grid

int rows = grid.length, cols = grid[0].length;

// Create a deque to hold the positions and the current obstacle count

Deque<int[]> queue = new ArrayDeque<>();

// Start from the upper left corner (0,0) with 0 obstacles

queue.offer(new int[] {0, 0, 0});

// Array to iterate over the 4 possible directions (up, right, down, left)

int[] directions = {-1, 0, 1, 0, -1};

// Visited array to keep track of positions already visited

boolean[][] visited = new boolean[rows][cols];

// Process cells until the queue is empty

while (!queue.isEmpty()) {

// Poll the current position and the number of obstacles encountered so far

int[] position = queue.poll();

int currentRow = position[0];

int currentCol = position[1];

int obstacles = position[2];

// Check if we have reached the bottom-right corner

if (currentRow == rows - 1 && currentCol == cols - 1) {

// If we reached the destination, return the number of obstacles encountered

return obstacles;

}

// If we have already visited this cell, skip it

if (visited[currentRow][currentCol]) {

continue;

}

// Mark the current cell as visited

visited[currentRow][currentCol] = true;

// Explore the neighboring cells

for (int h = 0; h < 4; ++h) {

int nextRow = currentRow + directions[h];

int nextCol = currentCol + directions[h + 1];

// Check the boundaries of the grid

// If the next cell is free (no obstacle)

if (grid[nextRow][nextCol] == 0) {

// Add it to the front of the queue to be processed with the same obstacle count

queue.offerFirst(new int[] {nextRow, nextCol, obstacles});

} else {

// If there's an obstacle, add it to the end of the queue with the obstacle count incremented by 1

queue.offerLast(new int[] {nextRow, nextCol, obstacles + 1});

}

}

}

}

// We include a return statement to satisfy the compiler, although the true return occurs inside the loop

return -1; // This will never be reached as the problem guarantees a path exists

}

}

**参考 1**

**Refer to**

<https://leetcode.com/problems/minimum-obstacle-removal-to-reach-corner/solutions/2086036/let-s-solve-this-problem-based-on-never-give-up-explained-failure-point/>

While giving the contest the first intuition which came to my mind is dp. Yes, off-course after solving a bunch of dp problems in the beginning I am also the person who wants to see dp everywhere.

So, Here is my code for using dp which failed 2nd test case though. But still, I have written something which I can analyze.

class Solution {

int [][]grid;

int n,m;

boolean [][]seen;

int []dx = new int[]{0,0,1,-1};

int []dy = new int[]{1,-1,0,0};

int [][]dp;

int finalres;

private boolean isValid(int i, int j) {

}

private int solve(int i, int j, int cnt) {

if(cnt>=finalres) return finalres;

if(i == n-1 && j == m-1) {

return cnt;

}

if(dp[i][j]!=Integer.MAX\_VALUE) return dp[i][j];

int res = n\*m+1;

seen[i][j]=true;

for(int k=0;k<4;k++) {

int newI = i+dx[k], newJ = j+dy[k];

if(isValid(newI, newJ)) {

res = Math.min(res, solve(newI, newJ, cnt+grid[i][j]));

}

}

seen[i][j]=false;

return dp[i][j]=Math.min(dp[i][j], res);

}

public int minimumObstacles(int[][] grid) {

this.grid = grid;

this.n = grid.length;

this.m = grid[0].length;

this.seen = new boolean[n][m];

dp = new int[n][m];

finalres = n\*m+1;

for(int []row:dp) Arrays.fill(row, Integer.MAX\_VALUE);

return solve(0,0,0);

}

}

**But It didn't work.**

**Why?**

since once I store the result of (i,j) using path let's say "path" then whenever I again come to (i,j), I just used the already saved result. So here is what my code assumes once we get the result for (i,j) then it's the optimal. But there is the possibility that a path from a different path that contributed to saving the dp result will be coming with less cost(i.e. cost == the number of blockers that needs to remove). That's why it's failed.

Ok so let's try another way

So at this point of time, I was confirmed that, dp wouldn't help here. So what next I can try?

Since from (0,0) to (n-1,m-1) I can have multiple paths and I need the path which has the minimum number of blockers(not exactly the path just count). So yeah here I thought let's do bfs so that each path will be running in the queue with their context and wouldn't merge as in dp.

So here is the BFS code :

class Solution {

int n,m;

int []dx = new int[]{0,0,1,-1};

int []dy = new int[]{1,-1,0,0};

int [][]dp;

private boolean isValid(int i, int j) {

}

public int minimumObstacles(int[][] grid) {

this.n = grid.length;

this.m = grid[0].length;

dp = new int[n][m];

for(int []row:dp)Arrays.fill(row,Integer.MAX\_VALUE);

Queue<int[]> queue = new LinkedList<>();

queue.add(new int[]{0,0,0});

int res = n\*m+1;// At most i have to remove all the blockers.

while(!queue.isEmpty()) {

int thisLevel = queue.size();

while(thisLevel-->0 ) {

int []temp = queue.remove();

int i=temp[0], j=temp[1];

if(i==n-1 && j == m-1) {

res = Math.min(res, temp[2]);

continue;

}

for(int k=0;k<4;k++) {

int newI = i+dx[k], newJ = j+dy[k];

// if newi and newj is valid and which is not increasing the cost for newi and newj which already i have achieved.

if(isValid(newI, newJ) && dp[newI][newJ]>temp[2]+grid[newI][newJ]) {

dp[newI][newJ]=temp[2]+grid[newI][newJ];

queue.add(new int[]{newI, newJ, dp[newI][newJ]});

}

}

}

}

return res;

}

}

**But again this code is given tle.**

**Why?**

So after doing analysis, I come to the point that, I was checking for any node (newI, newJ) which is going to be added to the queue that is, whether it is increasing the blocker count that I have already achieved for this node or not? Yeah, this is right and I should do that. But I skipped that If the same node (newI, newJ) is already in the queue with the greater number of blockers that we're getting now.

So why this is making an issue?

Since the node in the queue must be removed because we got a path to reach the (newI, newJ) with less blocker. If we don't remove this path this path again will start adding the node in the queue which gives the result as TLE.

ok so now let's update it.

class Solution {

int n,m;

int []dx = new int[]{0,0,1,-1};

int []dy = new int[]{1,-1,0,0};

int [][]dp;

int [][]grid;

private boolean isValid(int i, int j, int cost) {

if(valid) dp[i][j]=cost+grid[i][j];

return valid;

}

public int minimumObstacles(int[][] grid) {

this.grid = grid;

this.n = grid.length;

this.m = grid[0].length;

dp = new int[n][m];

for(int []row:dp)Arrays.fill(row,Integer.MAX\_VALUE);

Queue<int[]> queue = new LinkedList<>();

queue.add(new int[]{0,0,0});

int res = n\*m+1;// At most i have to remove all the blockers.

while(!queue.isEmpty()) {

int thisLevel = queue.size();

while(thisLevel-->0 ) {

int []temp = queue.remove();

int i=temp[0], j=temp[1];

if(i==n-1 && j == m-1) {

res = Math.min(res, temp[2]);

continue;

}

if(temp[2]>dp[i][j]) {

// This is invalid path because we have achived better which already might be running in the queue

continue;

}

for(int k=0;k<4;k++) {

int newI = i+dx[k], newJ = j+dy[k];

// if newi and newj is valid and which is not increasing the cost for newi and newj which already i have achieved.

if(isValid(newI, newJ, temp[2])) {

queue.add(new int[]{newI, newJ, dp[newI][newJ]});

}

}

}

}

return res;

}

}

**It's accepted**

Later when I came to the discussion page and read some code. **Then I remember this code is most likely the Dijkstra algorithm.**

**So morel is**

Instead of remembering the Algorithms just understand the actual concept and the problem for which any algorithms are made. Later in the future, you'll be building the same algorithms again and again without remembering.

**参考 2**

**这里的解法本质上叫做 0-1 BFS**

**Refer to**

<https://algo.monster/liteproblems/2290>

**Problem Description**

You are given a 2D integer array grid with two possible values in each cell:

0 represents an empty cell you can move through.

1 represents an obstacle that can be removed.

The grid has m rows and n columns, and the goal is to move from the top-left corner (0, 0) to the bottom-right corner (m - 1, n - 1). You can move in four directions from an empty cell: up, down, left, or right.

The problem asks you to find the minimum number of obstacles (1s) that need to be removed to enable this path-finding from the start to the end.

**Intuition**

To solve this problem, we use a breadth-first search (BFS) approach. BFS is a graph traversal method that expands and examines all nodes of a level before moving to the next level. This characteristic makes BFS suitable for finding the shortest path on unweighted graphs, or in this case, to find the minimum number of obstacles to remove.

**The idea is to traverse the grid using BFS, keeping track of the position (i, j) we're currently in and the count k of obstacles that we have removed to reach this position. Each time we can either move to an adjacent empty cell without incrementing the obstacle removal count or move to an adjacent cell with an obstacle by incrementing the removal count by one.**

**We use a deque to facilitate the BFS process. When we encounter an empty cell, we add the position to the front of the deque, giving priority to such moves for expansion before those requiring obstacle removal, effectively ensuring the BFS prioritizes paths with fewer obstacles removed.**

**A visited set vis is used to keep track of the cells we have already processed to avoid re-processing and potentially getting into cycles.**

The search continues until we reach the bottom-right corner (m - 1, n - 1), at which point we return the count of removed obstacles k which represents the minimum number of obstacles that need to be removed.

**Solution Approach**

The provided Python code implements the BFS algorithm using a deque, a double-ended queue that allows the insertion and removal of elements from both the front and the back with constant time complexity O(1).

**BFS (Breadth-First Search)**

At a high level, the BFS algorithm works by visiting all neighbors of a node before moving on to the neighbors' neighbors. It uses a queue to keep track of which nodes to visit next.

**Deque for BFS**

In the Python code, the deque is used instead of a regular queue. When the BFS explores a cell with an obstacle (value 1), it appends the new state to the end of the deque. Conversely, when it explores a cell without an obstacle (value 0), it appends the new state to the front of the deque.

**Implementation Details**

Initialize the deque q with a tuple containing the starting position and the initial number of obstacles removed (0, 0, 0).

Create a vis set to record visited positions and avoid revisiting them.

The dirs tuple is used to calculate the adjacent cells' positions with the help of a helper function like pairwise.

Enter a loop that continues until the end condition is met (reaching (m - 1, n - 1)).

Use popleft to get the current position and the count k of the obstacles removed.

If the target position is reached, return k.

If the position has been visited before ((i, j) in vis), just continue to the next iteration.

Otherwise, mark the position as visited by adding (i, j) to vis.

Iterate over all possible directions, and calculate the adjacent cell positions (x, y).

If (x, y) is within the grid bounds and is not an obstacle, append it to the front of the deque with the same number of removed obstacles k.

If (x, y) has an obstacle, append it to the end of the deque and increment the count of removed obstacles by 1.

This process prioritizes exploring paths with fewer obstacles, and since BFS is used, it guarantees that the first time we reach the bottom-right corner is via the path that requires the minimum number of obstacles to be removed.

The solution effectively mixes BFS traversal with a priority queue concept by using a deque to manage traversal order, ensuring an efficient search for the minimum obstacle removal path.

**Example Walkthrough**

Let's walk through a small example to illustrate the solution approach. Suppose we have the following grid:

grid = [

[0, 1, 0],

[0, 1, 1],

[0, 0, 0]

]

Here, m = 3 and n = 3. We want to move from (0, 0) to (2, 2) with the minimum number of obstacles removed.

**Step-by-Step Process:**

We start by initializing the deque q with the starting position and the initial number of obstacles removed, which is 0. So q = deque([(0, 0, 0)]).

The vis set is initialized to ensure we don't visit the same positions repeatedly. In the beginning, it's empty: vis = set().

The BFS begins. We dequeue the first element (i, j, k) = (0, 0, 0), where (i, j) represents the current cell, and k is the number of removed obstacles so far.

We have not reached the target, so we check the adjacent cells:

Right (0, 1): It has an obstacle. We increment obstacle count and add it to the end of q: q = deque([(0, 1, 1)]).

Down (1, 0): No obstacle. We add it to the front of q: q = deque([(1, 0, 0), (0, 1, 1)]).

We mark the current cell (0, 0) as visited: vis.add((0, 0)).

Now the deque q has cells to process, so we take the one from the front, which is (1, 0, 0).

From (1, 0), we again check the adjacent cells:

Right (1, 1): It has an obstacle, add it to end: q = deque([(0, 1, 1), (1, 1, 1)]).

Down (2, 0): No obstacle, add it to front: q = deque([(2, 0, 0), (0, 1, 1), (1, 1, 1)]).

We mark the cell (1, 0) as visited: vis.add((1, 0)).

The steps repeat, dequeuing from the front, checking adjacent cells, and enqueueing in the deque according to the rules.

Eventually, we dequeue the cell (2, 0, 0). From here, we can go right to (2, 1) with no obstacle and add it to the front: q = deque([(2, 1, 0), (0, 1, 1), (1, 1, 1)]).

Again, we mark (2, 0) as visited: vis.add((2, 0)).

Dequeuing (2, 1, 0), we can go right to the target (2, 2) with no obstacle. We add it to the front.

Mark (2, 1) as visited and check the next cell from q.

Finally, we reach (2, 2) which is the target. We have not needed to remove any obstacles, so we return k = 0.

Throughout this process, we have explored paths with the fewest obstacles first, using the deque to effectively prioritize cells without obstacles. By doing this, we've ensured that the first time we reach the end, it is the path with the minimum number of obstacles removed. In this case, no obstacles needed to be removed.

**Solution Implementation**

class Solution {

public int minimumObstacles(int[][] grid) {

// Get the dimensions of the grid

int rows = grid.length, cols = grid[0].length;

// Create a deque to hold the positions and the current obstacle count

Deque<int[]> queue = new ArrayDeque<>();

// Start from the upper left corner (0,0) with 0 obstacles

queue.offer(new int[] {0, 0, 0});

// Array to iterate over the 4 possible directions (up, right, down, left)

int[] directions = {-1, 0, 1, 0, -1};

// Visited array to keep track of positions already visited

boolean[][] visited = new boolean[rows][cols];

// Process cells until the queue is empty

while (!queue.isEmpty()) {

// Poll the current position and the number of obstacles encountered so far

int[] position = queue.poll();

int currentRow = position[0];

int currentCol = position[1];

int obstacles = position[2];

// Check if we have reached the bottom-right corner

if (currentRow == rows - 1 && currentCol == cols - 1) {

// If we reached the destination, return the number of obstacles encountered

return obstacles;

}

// If we have already visited this cell, skip it

if (visited[currentRow][currentCol]) {

continue;

}

// Mark the current cell as visited

visited[currentRow][currentCol] = true;

// Explore the neighboring cells

for (int h = 0; h < 4; ++h) {

int nextRow = currentRow + directions[h];

int nextCol = currentCol + directions[h + 1];

// Check the boundaries of the grid

// If the next cell is free (no obstacle)

if (grid[nextRow][nextCol] == 0) {

// Add it to the front of the queue to be processed with the same obstacle count

queue.offerFirst(new int[] {nextRow, nextCol, obstacles});

} else {

// If there's an obstacle, add it to the end of the queue with the obstacle count incremented by 1

queue.offerLast(new int[] {nextRow, nextCol, obstacles + 1});

}

}

}

}

// We include a return statement to satisfy the compiler, although the true return occurs inside the loop

return -1; // This will never be reached as the problem guarantees a path exists

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the algorithm can be estimated by considering the number of operations it performs. The algorithm uses a queue that can have at most every cell (i, j) from the grid enqueued, and the grid has m \* n cells. Each cell is inserted into the queue at most once, since we're using a set of visited nodes to prevent re-visitation.

The deque operations appendleft and popleft have O(1) time complexity. The for loop executes at most four times for each cell, corresponding to the four possible directions one can move in the grid.

Considering all of this, the overall time complexity is O(m \* n), as every cell is considered at most once.

**Space Complexity**

The space complexity consists of the space needed for the queue and the set of visited cells.

The queue can store up to m \* n elements, if all cells are inserted, and the set will at the same time contain a maximum of m \* n elements as well to track visited cells. The sum is 2 \* m \* n but in terms of Big O notation, the constants are ignored.

Thus, the space complexity is O(m \* n) for the queue and the visited set combined.

**参考 3**

**Refer to**

**[Java/Python 3] 2 codes: Shortest Path & BFS, w/ brief explanation, analysis and similar problems.**

<https://leetcode.com/problems/minimum-obstacle-removal-to-reach-corner/solutions/2085640/java-python-3-2-codes-shortest-path-bfs-w-brief-explanation-analysis-and-similar-problems/>

**Q & A**

Q1: Why do we not need to keep track of which nodes we've already visited? Is this code perhaps already implicitly tracking the nodes we've visited? According to the implementation of Lazy Dijkstra here, we need to keep track of which nodes we've already visited.

A1: The PriorityQueue/heap always keep track of the cell that we currently can reach with shortest path, and grid[i][j] + o < dist[i][j] makes sure we don't need to visit the cell again if we can not reach it with less obstacles.

Q2: .Is it possible to optimize backtracking code with dp for this problem?

@Mikey98 @Adithya\_U\_Bhat contributed the following anwer.

A2: No. In this question 1 <= m, n <= 105, which is toooo big for backtracking; You cannot use dp here as there are 4 directions and revisiting a node again can give you minimum answer so dp fails here , remember one thing if its given 4 direction always go with graphs

**End of Q & A**

**Method 1: Shortest Path**

Initialize dist with Integer.MAX\_VALUE/math.inf, and use dist[i][j] to indicate the currently minimum obstacles need to remove to reach (i, j);

Starting from (0, 0), put [grid[0][0], 0, 0] into a PriorityQueue/heap to begin to search by Shortest Path; Once we can reach any (i, j) from its neighbor with fewer obstacles, we update it with the less value and put the corresponding information array [dist[i][j], i, j] into PriorityQueue/heap, repeat till we find a path to (m - 1, n - 1).

private static final int[] d = {0, 1, 0, -1, 0};

public int minimumObstacles(int[][] grid) {

int m = grid.length, n = grid[0].length;

int[][] dist = new int[m][n];

for (int[] di : dist) {

Arrays.fill(di, Integer.MAX\_VALUE);

}

dist[0][0] = grid[0][0];

PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));

pq.offer(new int[]{dist[0][0], 0, 0});

while (!pq.isEmpty()) {

int[] cur = pq.poll();

int o = cur[0], r = cur[1], c = cur[2];

if (r == m - 1 && c == n - 1) {

return o;

}

for (int k = 0; k < 4; ++k) {

int i = r + d[k], j = c + d[k + 1];

dist[i][j] = o + grid[i][j];

pq.offer(new int[]{dist[i][j], i, j});

}

}

}

return dist[m - 1][n - 1];

}

Analysis:

Time: O(m \* n log(m \* n)), space: O(m \* n).

**这里的解法本质上叫做 0-1 BFS**

**Method 2: Modified BFS -- credit to @Doskarin.**

Initialize dist with Integer.MAX\_VALUE/math.inf, and use dist[i][j] to indicate the currently minimum obstacles need to remove to reach (i, j);

Starting from (0, 0), put [grid[0][0], 0, 0] into a Deque to begin BFS, and use dist value to avoid duplicates;

Whenever encountering an empty cell neighbor, the dist value is same and hence we can put it to the front of the Deque; Otherwise, put it to the back of the Deque;

Repeat 2. and 3. and update dist accordingly till the Deque is empty;

return dist[m - 1][n - 1] as solution.

private static final int[] d = {0, 1, 0, -1, 0};

private static final int M = Integer.MAX\_VALUE;

public int minimumObstacles(int[][] grid) {

int m = grid.length, n = grid[0].length;

int[][] dist = new int[m][n];

for (int[] di : dist) {

Arrays.fill(di, M);

}

dist[0][0] = 0;

Deque<int[]> dq = new ArrayDeque<>();

dq.offer(new int[3]);

while (!dq.isEmpty()) {

int[] cur = dq.poll();

int o = cur[0], r = cur[1], c = cur[2];

for (int k = 0; k < 4; ++k) {

int i = r + d[k], j = c + d[k + 1];

if (grid[i][j] == 1) {

dist[i][j] = o + 1;

dq.offer(new int[]{o + 1, i, j});

}else {

dist[i][j] = o;

dq.offerFirst(new int[]{o, i, j});

}

}

}

}

return dist[m - 1][n - 1];

}

Analysis:

Since each cell is visited at most once, therefore

Time & space: O(m \* n).

**参考 4**

**这里的解法本质上叫做 0-1 BFS，并且解释了在什么情况下 0-1 BFS 可以代替 Dijkstra**

**Refer to**

<https://leetcode.com/problems/minimum-obstacle-removal-to-reach-corner/solutions/2086235/0-1-bfs-c/>

**Solution in short :**

Standard 0-1 BFS.

Use deque and push the grid without obstacle to the front and the grid with obstacle to the end and do a normal BFS.

**Intuition**

**The idea is similar to that of Djikstra. Dijkstra uses a set or priority queue because at every iteration edge with smallest cost is required. So an additional time of sorting the edges is involved at each step. Here we have a special weighted graph where every edge has a weight of either 0 or 1. In this kind of graph, Dijkstra can be further optimized to avoid sorting at every iteration.**

For every node that you visit, push all its **unvisited neighbours** to the queue in the following manner.

**Every unvisited neighbour with cost 1 (having an obstacle), push to the end of the queue**

**Every unvisited neighbour with cost 0 (having no obstacle), push to the beginning of the queue.**

By the time you push a node to the queue, compute and store its distance in dp array.

**I think it is enough to push any node only once in the queue using the visited flag. You can easily prove that the queue will always be sorted by the distances**

int dx[] = {1,-1,0,0};

int dy[] = {0,0,1,-1};

class Solution {

public:

int minimumObstacles(vector<vector<int>>& grid) {

int m = grid.size();

int n = grid[0].size();

vector<vector<int>> dp(m, vector<int>(n,INT\_MAX));

vector<vector<bool>> vis(m, vector<bool>(n,0));

deque<pair<int,int>> q;

q.push\_front({0,0});

dp[0][0] = 0;

while(q.size()) {

pair<int,int> p = q.front();

q.pop\_front();

int cx = p.first;

int cy = p.second;

for(int i=0;i<4;i++) {

int tx = cx + dx[i];

int ty = cy + dy[i];

if(!vis[tx][ty]) {

dp[tx][ty] = dp[cx][cy] + (grid[tx][ty] == 1);

if(grid[tx][ty] == 1) {

q.push\_back({tx,ty});

} else {

q.push\_front({tx,ty});

}

vis[tx][ty] = true;

}

}

}

}

return dp[m-1][n-1];

}

};

**参考 5**

**Refer to**

**Two methods || 0-1 bfs || dijkastras algo ||100% faster|| resources included**

<https://leetcode.com/problems/minimum-obstacle-removal-to-reach-corner/solutions/2088276/two-methods-0-1-bfs-dijkastras-algo-100-faster-resources-included/>

**Dijkastras algo method**

**When do we use dijkastras,**

When there is a weighted graph and there are one or multiple destinations possible

**What do we do in this**

1.We need to store the minimum distance between nodes

2.In order to do this we use a data structure like set or a priority queue

**In questions point of view :-**

**Consider the edge weight from a free node to obstacle node to be 1 and from free node to free node 0**

Now apply dijkastras algo

int m=grid.size(), n=grid[0].size();

vector<int> dir={0,1,0,-1,0};

vector<vector<int>> dist(m, vector<int> (n,INT\_MAX));

dist[0][0]=0;

pq.push({0,{0,0}});

while(!pq.empty())

{

auto v=pq.top();

pq.pop();

int i=v.second.first, j=v.second.second, d=v.first;

for(int k=0;k<4;k++)

{

int x=i+dir[k], y=j+dir[k+1];

if(x<0 || x>=m || y<0 || y>=n) continue;

int wt;

if(grid[x][y]==1)

{

wt=1;

}

else

{

wt=0;

}

if(d+wt<dist[x][y])

{

dist[x][y]=d+wt;

pq.push({dist[x][y],{x,y}});

}

}

}

return dist[m-1][n-1];

}

This approach gives us the answer but its time complexity is (O(V+E)logN)

Extra logN comes because of the use of Priority Queue

TIME TO THINK == > Do we really need dijkastras algo. No we don't because we have only 2 different kinds of weights so we have a better and faster approach i.e 0-1 BFS.

**0-1BFS Method**

Let me try to explain when we use 0-1 bfs and what are the advantages :-

This is used when only two types of states/ weights are present

This works using a deque so the complexity is O(V+E) like a normal bfs traversal

Links to refer for 0-1 BFS

<https://cp-algorithms.com/graph/01_bfs.html>

class Solution {

public:

int minimumObstacles(vector<vector<int>>& grid) {

int m=grid.size(), n=grid[0].size();

vector<int> dir={0,1,0,-1,0};

vector<vector<int>> dist(m, vector<int> (n,INT\_MAX));

vector<vector<bool>> vis(m, vector<bool>(n,0));

deque<pair<int,int>>q;

dist[0][0]=0;

q.push\_front({0,0});

while(!q.empty())

{

auto cur=q.front();

q.pop\_front();

int x=cur.first;

int y=cur.second;

for(int i=0;i<4;i++)

{

int cx=x+dir[i];

int cy=y+dir[i+1];

if(cx>=0 and cy>=0 and cx<m and cy<n)

{

if(!vis[cx][cy])

{

dist[cx][cy]=dist[x][y]+(grid[cx][cy]==1);

if(grid[cx][cy]==1)

q.push\_back({cx,cy});//obstacle cell pushed at the end

else

q.push\_front({cx,cy}); //empty cell pushed on top

vis[cx][cy] = true;

}

}

}

}

return dist[m-1][n-1];

}

};

**Why in above Leetcode 1293 Dijkstra solution we need one more field in 'State' object as 'steps' but Leetcode 2290 Dijkstra solution don't need it ?**

The key difference between the two problems lies in the nature of the constraints and the goal for each problem.

**Leetcode 1293: "Shortest Path in a Grid with Obstacles Elimination"**

**Problem Requirements:**

Goal: Find the shortest path from the top-left to the bottom-right of the grid.

Constraints: You can eliminate up to k obstacles.

Objective: Minimize the number of steps to reach the target cell.

**Why We Need steps:**

Path Length: The primary goal is to minimize the number of steps taken to reach the target. Each state needs to keep track of the number of steps to ensure we are indeed minimizing the path length.

Optimal Path: When we evaluate which state to process next, we need to compare not just the number of obstacles removed but also the number of steps taken to reach that state. The steps field helps us prioritize states that lead to shorter paths.

Priority Queue: The priority queue needs to sort states not only by the number of obstacles removed but also by the number of steps taken. Thus, steps is essential for the ordering in the priority queue.

**Leetcode 2290: "Minimum Obstacle Removal to Reach Corner"**

**Problem Requirements:**

Goal: Find the minimum number of obstacles to remove to get from the top-left to the bottom-right of the grid.

Constraints: The number of obstacles is the primary constraint.

Objective: Minimize the number of obstacles removed.

**Why We Don't Need steps:**

Obstacle Removal Focus: The primary objective is to minimize the number of obstacles removed, not the number of steps taken.

Sufficient to Track Obstacles: The shortest path in terms of steps is inherently considered because we always take the path with the fewest obstacles first. Since each step contributes equally to the path length in an unweighted sense, tracking the number of obstacles is sufficient.

Priority Queue: The priority queue in this problem only needs to order states by the number of obstacles removed, making the steps field unnecessary.

**Summarizing the Differences:**

Leetcode 1293:

Objective: Minimize steps while keeping track of the number of obstacles removed.

State Representation: (row, col, steps, obstaclesRemoved).

Priority Queue: Orders states based on steps to ensure the shortest path in terms of steps.

Leetcode 2290:

Objective: Minimize the number of obstacles removed to reach the target.

State Representation: (row, col, obstaclesRemoved).

Priority Queue: Orders states based on obstaclesRemoved to ensure the minimum obstacles are removed.

In summary, the difference in the requirement of the steps field arises from the different objectives of the two problems: minimizing path length in one case and minimizing obstacle removal in the other.

**Refer to**

[L1293.Shortest Path in a Grid with Obstacles Elimination (Ref.L743,L2290)](note://WEBa01cb15801adc2626262e2f194736bef)

[0-1 BFS](note://WEB28136176b9a074a7d4b4ae45d6b6bc7a)

[Dijkstra Shortest Path Algorithm - A Detailed and Visual Introduction](note://80857119213E49EC840091BB3F7E4356)