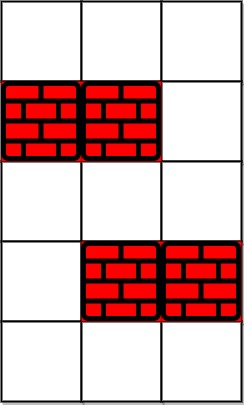
<https://leetcode.com/problems/shortest-path-in-a-grid-with-obstacles-elimination/>

You are given an m x n integer matrix grid where each cell is either 0 (empty) or 1 (obstacle). You can move up, down, left, or right from and to an empty cell in one step.

Return the minimum number of steps to walk from the upper left corner (0, 0) to the lower right corner (m - 1, n - 1) given that you can eliminate at most k obstacles. If it is not possible to find such walk return -1.

**Example 1:**

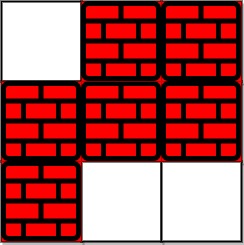


**Input:** grid = [[0,0,0],[1,1,0],[0,0,0],[0,1,1],[0,0,0]], k = 1

**Output:** 6

**Explanation:** The shortest path without eliminating any obstacle is 10.The shortest path with one obstacle elimination at position (3,2) is 6. Such path is (0,0) -> (0,1) -> (0,2) -> (1,2) -> (2,2) -> **(3,2)** -> (4,2).

**Example 2:**



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

**Output:** -1

**Explanation:** We need to eliminate at least two obstacles to find such a walk.

**Constraints:**

m == grid.length

n == grid[i].length

1 <= m, n <= 40

1 <= k <= m \* n

grid[i][j] is either 0 **or** 1.

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

**Attempt 1: 2024-06-02**

**Solution 1: BFS**

**Style 1: Additional 2D array to store status for deserving 're-visiting' cell (60 min)**

class Solution {

    public int shortestPath(int[][] grid, int k) {

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

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

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

        // Initially have k chances to eliminate obstacles

        eliminateChanceRemains[0][0] = k;

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

        visited[0][0] = true;

        int distance = 0;

        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 currPathRemainChance = cur[2];

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

                    return distance;

                }

                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 remains chances less

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

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

                    //    used to record

                            || eliminateChanceRemains[new\_x][new\_y] < currPathRemainChance)) {

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

                            // Since current cell value is 0 not consuming chance

                            eliminateChanceRemains[new\_x][new\_y] = currPathRemainChance;

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

                        } else if(grid[new\_x][new\_y] == 1 && currPathRemainChance > 0) {

                            // Since current cell value is 1 consuming 1 chance

                            eliminateChanceRemains[new\_x][new\_y] = currPathRemainChance - 1;

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

                        }

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

                    }

                }

            }

            distance++;

        }

        return -1;

    }

}

**Refer to**

<https://leetcode.com/problems/shortest-path-in-a-grid-with-obstacles-elimination/discuss/452184/Clean-Java-BFS-with-comments.>

At every index (x,y) of the matrix along with the visited information, we also have to maintain the maximum amount of obstacles that can be eliminated(obstacleCount) and choose/update that path with highest obstacleCount.

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

public int shortestPath(int[][] grid, int k) {

int m = grid.length;

if(m==0) {

return 0;

}

int n = grid[0].length;

int[][] obstacleThatCanBeEliminated = new int[m][n]; // Number of obstacles that can be eliminated.

boolean[][] visited = new boolean[m][n];// Check if the index has been visited.

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

queue.add(new int[]{0,0,k});// x,y,obstacleCount

obstacleThatCanBeEliminated[0][0]=k; // Initially we can eliminate k obstacles.

int step=1;

while(!queue.isEmpty()) {

int size=queue.size();

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

int[] poll = queue.poll();

if(poll[0]==m-1 && poll[1]==n-1) {

return step-1;

}

int currentObstacleCount = poll[2];

for(int[] d : dir) {

int nextX = poll[0]+d[0];

int nextY = poll[1]+d[1];

/\*Add the next element to the queue if it has not been visited yet or it has

been visited but the number of obstacles encountered are greater than the current path,

hence we can replace it with the current path.\*/

(obstacleThatCanBeEliminated[nextX][nextY]<currentObstacleCount || !visited[nextX][nextY])

&& (grid[nextX][nextY]==0 || (grid[nextX][nextY]==1 && currentObstacleCount>0))

/\*currentObstacleCount>0 it means the current obstacle can also be eliminated\*/) {

if(grid[nextX][nextY]==1) {

queue.add(new int[]{nextX,nextY,currentObstacleCount-1});

obstacleThatCanBeEliminated[nextX][nextY]=currentObstacleCount-1;

} else {

queue.add(new int[]{nextX,nextY,currentObstacleCount});

obstacleThatCanBeEliminated[nextX][nextY]=currentObstacleCount;

}

visited[nextX][nextY]=true;

}

}

}

step++;

}

return -1;

}

**Refer to**

**O(m\*n\*k) BFS Solution with Explanation**

<https://leetcode.com/problems/shortest-path-in-a-grid-with-obstacles-elimination/solutions/451787/python-o-m-n-k-bfs-solution-with-explanation/>

**Solution Explanation**

Because we are trying to find the shortest path, use BFS here to exit immediately when a path reaches the bottom right most cell.

Use a set to keep track of already visited paths. We only need to keep track of the row, column, and the eliminate obstacle usage count. We don't need to keep track of the steps because remember we are using BFS for the shortest path. That means there is no value storing a 4th piece of the data, the current steps. This will reduce the amount of repeat work.

m = rows

n = columns

k = allowed elimination usages

**Time Complexity**

O(m**n**k) time complexity

This is because for every cell (m\*n), in the worst case we have to put that cell into the queue/bfs k times.

Runtime: 68 ms, faster than 33.33% of Python3 online submissions

**Space Complexity**

O(m**n**k) space complexity

This is because for every cell (m\*n), in the worst case we have to put that cell into the queue/bfs k times which means we need to worst case store all of those steps/paths in the visited set.

Memory Usage: 13.9 MB, less than 100.00% of Python3 online submissions

Code

from collections import deque

class Solution:

def shortestPath(self, grid: List[List[int]], k: int) -> int:

if len(grid) == 1 and len(grid[0]) == 1:

return 0

queue = deque([(0,0,k,0)])

visited = set([(0,0,k)])

if k > (len(grid)-1 + len(grid[0])-1):

return len(grid)-1 + len(grid[0])-1

while queue:

row, col, eliminate, steps = queue.popleft()

for new\_row, new\_col in [(row-1,col), (row,col+1), (row+1, col), (row, col-1)]:

if (new\_row >= 0 and

new\_row < len(grid) and

new\_col >= 0 and

new\_col < len(grid[0])):

if grid[new\_row][new\_col] == 1 and eliminate > 0 and (new\_row, new\_col, eliminate-1) not in visited:

visited.add((new\_row, new\_col, eliminate-1))

queue.append((new\_row, new\_col, eliminate-1, steps+1))

if grid[new\_row][new\_col] == 0 and (new\_row, new\_col, eliminate) not in visited:

if new\_row == len(grid)-1 and new\_col == len(grid[0])-1:

return steps+1

visited.add((new\_row, new\_col, eliminate))

queue.append((new\_row, new\_col, eliminate, steps+1))

return -1

**Some analysis of the problem:**

All the cells would be visited more than once as we could reach same cell with more distance but less obstacles which could be helpful later in traversal so we need to consider all the paths passing through same cell even with more distance.

Using PQ doesn't make sense here due to reason no 1. so we are better off with simple queue without any comparator.

Here we can also use 2D array for visited[m][n] = obstacles\_till\_here, then you will have to check if you find some cell with lesser no of obstacles reaching this point then consider that path. 3D array makes life easier in contest, but in interview its better to discuss about space constraints

**Style 2: 3D array without explicit declare new 'currPathRemainChance' (60 min)**

class Solution {

    public int shortestPath(int[][] grid, int k) {

        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;

        boolean[][][] visited = new boolean[m][n][k + 1];

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

        // Initially have k chances to eliminate obstacles

        q.offer(new int[] {0, 0, k});

        visited[0][0][k] = true;

        int distance = 0;

        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 currPathRemainChance = cur[2];

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

                    return distance;

                }

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

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

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

                        if(grid[new\_x][new\_y] == 1 && currPathRemainChance > 0

&& !visited[new\_x][new\_y][currPathRemainChance - 1]) {

                            visited[new\_x][new\_y][currPathRemainChance - 1] = true;

                            q.offer(new int[] {new\_x, new\_y, currPathRemainChance - 1});

                        }

                        if(grid[new\_x][new\_y] == 0 && !visited[new\_x][new\_y][currPathRemainChance]) {

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

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

                        }

                    }

                }

            }

            distance++;

        }

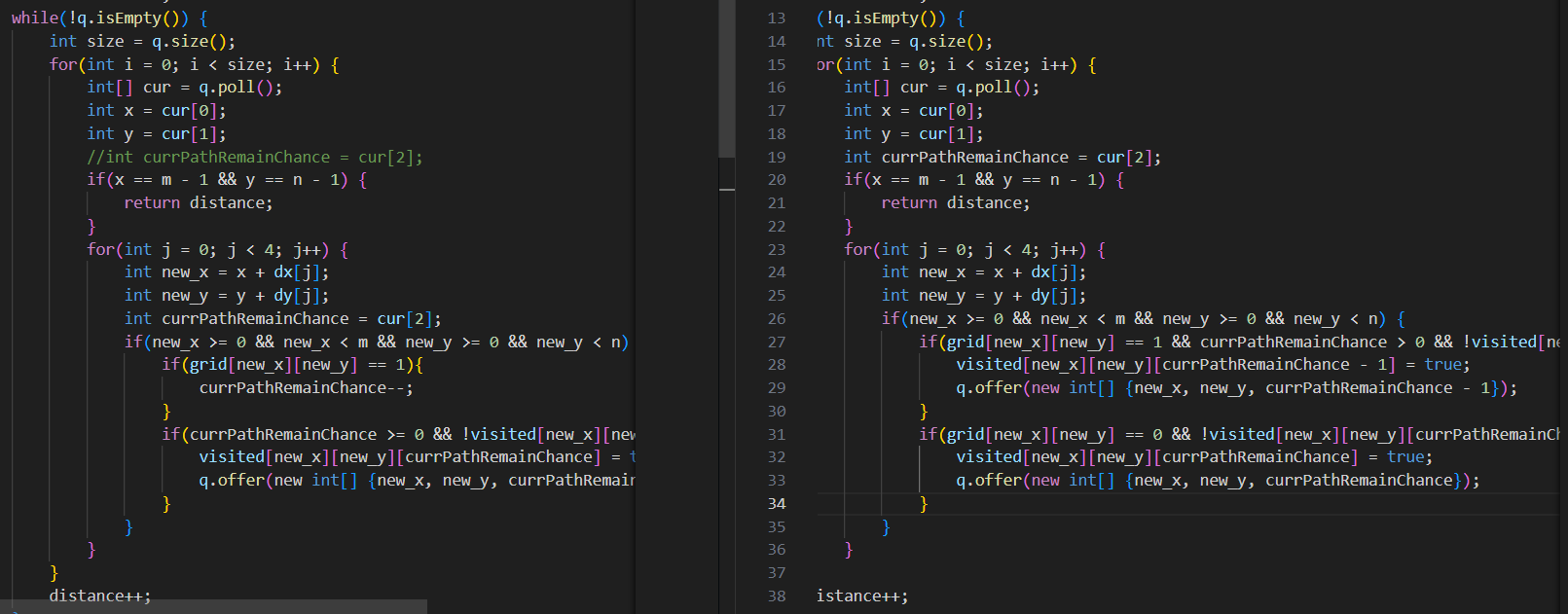
        return -1;

    }

}

**Style 3: 3D array with explicit declare new 'currPathRemainChance' (60 min)**

**与Style 2的重大不同来自于改变了 currPathRemainChance 定义的位置，必须在对4个方向开启扫描的时候分别定义1次（意味着总共初始化4次），而不是在开启扫描之前定义1次，而4次扫描中每一次 都是cur[2]，必须通过这个办法来保持每次扫描的 currPathRemainChance 不变**



class Solution {

    public int shortestPath(int[][] grid, int k) {

        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;

        boolean[][][] visited = new boolean[m][n][k + 1];

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

        // Initially have k chances to eliminate obstacles

        q.offer(new int[] {0, 0, k});

        visited[0][0][k] = true;

        int distance = 0;

        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 currPathRemainChance = cur[2];

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

                    return distance;

                }

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

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

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

                    int currPathRemainChance = cur[2];

                        if(grid[new\_x][new\_y] == 1){

                            currPathRemainChance--;

                        }

                        if(currPathRemainChance >= 0 && !visited[new\_x][new\_y][currPathRemainChance]) {

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

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

                        }

                    }

                }

            }

            distance++;

        }

        return -1;

    }

}

**Refer to**

<https://algo.monster/liteproblems/1293>

**Brute Force**

First, we might think to try all possible eliminations of at most k obstacles.

Then, on every possible elimination, we can run a BFS/flood fill algorithm on the new grid to find the length of the shortest path. Our final answer will be the minimum of all of these lengths.

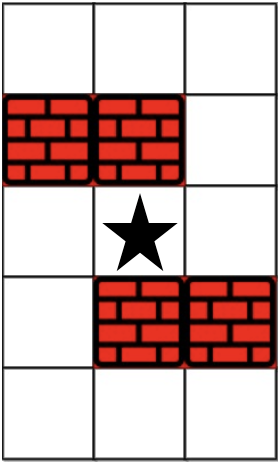
However, this is way too inefficient and complicated.

**Full Solution**

Instead of thinking of first removing obstacles and then finding the shortest path, we can find the shortest path and remove obstacles along the way when necessary.

To accomplish this, we'll introduce an additional state by adding a counter for the number of obstacles we removed so far in our path. For any path, we can extend that path by moving up, left, down, or right. If the new cell we move into is blocked, we will remove that obstacle and add 1 to our counter. However, since we can remove no more than K obstacles, we can't let our counter exceed K.

**Example**



Let's look at our destination options if we started in the cell grid[2][1] with the obstacle counter at 0.

|  |  |  |  |
| --- | --- | --- | --- |
| Cell | Change In    Row | Change In    Column | Change In    Obstacle Counter |
| grid[1][1] | -1 | +0 | +1 |
| grid[2][2] | +0 | +1 | +0 |
| grid[3][1] | +1 | +0 | +1 |
| grid[2][0] | +0 | -1 | +0 |

We can also make the observation that each position/state (row, column, obstacle counter) can act as a node in a graph and each destination option can act as a directed edge in a graph.

In this specific example with the node (2,1,0), we have 4 directed edges with the destinations being (1,1,1), (2,2,0), (3,1,1), and (2,0,0).

Since each edge has length 1, we can run a BFS/flood fill algorithm on the graph to find the answer. Using BFS to find the shortest path will work in this case as the graph is unweighted. While running the algorithm, we'll look for the first instance we traverse through a node u which is located in the bottom right corner (i.e. row = m - 1 and column = n - 1). Since BFS/flood fill traverses through nodes in non-decreasing order by the distance from the start node (0,0,0), our answer will be the distance from the start node (0,0,0) to node u. This is always true no matter what the obstacle counter is for that node (assuming it doesn't exceed k). If no such node is traversed, then our answer is −1.

Essentially, we'll create a graph with nodes having 3 states (row, column, obstacle counter) and run a BFS/flood fill on it to find the minimum distance between the start and end nodes.

**Time Complexity**

Let's think of how different nodes exist in our graph. There are O(MN) cells in total, and in each cell, our current counter of obstacles ranges from 0 to K, inclusive. This gives us O(K) options for our obstacle counter, yielding O(MNK) nodes. From each node, we have a maximum of 4 other destinations we can visit (i.e. edges), which is equivalent to O(1). From all O(MNK) nodes, we also obtain O(MNK) total edges.

Our graph has O(MNK) nodes and O(MNK) edges. A BFS with O(MNK) nodes and O(MNK) edges will have a final time complexity of O(MNK).

Time Complexity: O(MNK)

Space Complexity

Our graph has O(MNK) nodes so a BFS will have a space complexity of O(MNK) as well.

**Java solution**

class Solution {

public int shortestPath(int[][] grid, int k) {

int m = grid.length;

int n = grid[0].length; // dimensions of the grid

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

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

// nodes are in the form (row, column, obstacles removed so far)

int[][][] dis = new int[m][n][k + 1]; // keeps track of distance of nodes

boolean[][][] vis =

new boolean[m][n][k + 1]; // keeps track of whether or not we visited a node

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

int[] start = {0, 0, 0};

q.add(start); // starting at upper left corner for BFS/floodfill

vis[0][0][0] = true;

while (!q.isEmpty()) {

int[] cur = q.poll();

int curX = cur[0]; // current row

int curY = cur[1]; // current column

int curK = cur[2]; // current obstacles removed

if (curX == m - 1

&& curY == n - 1) { // check if node is in bottom right corner

return dis[curX][curY][curK];

}

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

int newX = curX + deltaX[i]; // row of destination

int newY = curY + deltaY[i]; // column of destination

if (newX < 0 || newX >= m || newY < 0

|| newY >= n) { // check if it's in boundary

continue;

}

int newK = curK; // obstacle count of destination

if (grid[newX][newY] == 1)

newK++;

if (newK > k) { // surpassed obstacle removal limit

continue;

}

if (vis[newX][newY][newK]) { // check if node has been visited before

continue;

}

dis[newX][newY][newK] = dis[curX][curY][curK] + 1;

vis[newX][newY][newK] = true;

int[] destination = {newX, newY, newK};

q.add(destination);

// process destination node

}

}

return -1; // no valid answer found

}

}

**Solution 2: Dijkstra (120 min)**

**Style 1:**

class Solution {

    public int shortestPath(int[][] grid, int k) {

        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[][][] distances = new int[m][n][k + 1];

        for(int[][] distance : distances) {

            for(int[] d : distance) {

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

            }

        }

        // Initialize distance for all potential removing obstacles cases(0 ~ k) as 0

        Arrays.fill(distances[0][0], 0);

        // min-heap storing {i, j, # obstacles eliminated, distance}, sorted by distance to (0,0)

        // Similar strategy as L743.Network Delay Time

        PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> a[3] - b[3]);

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

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

        while(!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

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

                int new\_x = cur[0] + dx[i];

                int new\_y = cur[1] + dy[i];

                // Must in boundary

                    int new\_k = cur[2] + grid[new\_x][new\_y];

                    // Must under # obstacles eliminated limit

                    if(new\_k <= k) {

                        // Continue if we have more optimal result:

                        // Under same number of obstacles removed case, if the

                        // new path(new\_distance) able to reach longer distance

                        // than existing path(distances[new\_x][new\_y][new\_k]),

                        // we will update current path to new path, which means

                        // addding the node into minPQ

                        int new\_distance = cur[3] + 1;

                        if(distances[new\_x][new\_y][new\_k] > new\_distance) {

                            distances[new\_x][new\_y][new\_k] = new\_distance;

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

                        }

                    }

                }

            }

        }

        // Each number of removing obstacle case(0 ~ k) will have a

        // minimum distance, scan all of them to find final minimum one

        int result = distances[m - 1][n - 1][0];

        for(int i = 1; i <= k; i++) {

            result = Math.min(result, distances[m - 1][n - 1][i]);

        }

        // If not able to find a path to reach {m - 1, n - 1}

        // just return -1

        return result == Integer.MAX\_VALUE ? -1 : result;

    }

}

**Refer to**

**Dijkstra Solution**

**[Java] Clean O(MNK)-Time BFS Solution || comparing with Dijkstra's**

<https://leetcode.com/problems/shortest-path-in-a-grid-with-obstacles-elimination/solutions/1188835/java-clean-o-mnk-time-bfs-solution-comparing-with-dijkstra-s/>

This most optimal BFS Solution is actually not as straightforward as it seems to be. The **visited** matrix record the minimum obstacles removed to get to that entry. Thus, this Solution is different from the classic BFS Algorithm in these aspects :

**a position (x,y) may be visited more than 1 time**

**a position (newX, newY) should not enter queue if we have a more optimal result**

Namely, we have the following code:

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

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

visited[0][0] = 0;

instead of

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

for (boolean[] b : visited) Arrays.fill(b, false);

visited[0][0] = true;

And in during the BFS process, we have:

if (visited[newX][newY] <= newK) continue;

instead of

if (visited[newX][newY]) continue;

The solution below is a direct analogy to the classical BFS Algorithm:

class Solution {

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

public int shortestPath(int[][] grid, int k) {

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

// storing {i, j, # obstacles eliminated}

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

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

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

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

visited[0][0] = 0;

int dist = 0;

while (!queue.isEmpty()) {

int size = queue.size();

while (size-- > 0) {

int[] curr = queue.poll();

if (curr[0] == m-1 && curr[1] == n-1) return dist;

for (int[] dir : DIRECTIONS) {

int newX = curr[0] + dir[0];

int newY = curr[1] + dir[1];

// 1). continue if out of bound

if (newX < 0 || newY < 0 || newX >= m || newY >= n) continue;

// 2). continue if out of elimation

int newK = curr[2] + grid[newX][newY];

if (newK > k) continue;

// 3). continue if we have more optimal result

if (visited[newX][newY] <= newK) continue;

visited[newX][newY] = newK;

queue.offerLast(new int[]{newX, newY, newK});

}

}

dist++;

}

return -1;

}

}

However, I still want to point out that the reasoning above is really similar to Dijkstra's Algorithm. This Dijkstra-Solution is actually even more straightforward.

Before jump into the Dijkstra-Solution, I recommand comparing these problems:

1102. Path With Maximum Minimum Value

1368. Minimum Cost to Make at Least One Valid Path in a Grid

1631. Path With Minimum Effort

1293. Shortest Path in a Grid with Obstacles Elimination

class Solution {

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

public int shortestPath(int[][] grid, int k) {

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

int[][][] dists = new int[m][n][k+1];

for (int[][] dist : dists)

for (int[] d : dist)

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

Arrays.fill(dists[0][0], 0);

// min-heap storing {i, j, # obstacles eliminated, dist}, sorted by distance to (0,0)

PriorityQueue<int[]> heap = new PriorityQueue<>((a,b) -> a[3] - b[3]);

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

while (!heap.isEmpty()) {

int[] curr = heap.poll();

if (curr[0] == m-1 && curr[1] == n-1) continue;

for (int[] dir : DIRECTIONS) {

int newX = curr[0] + dir[0];

int newY = curr[1] + dir[1];

// 1). continue if out of bound

if (newX < 0 || newY < 0 || newX >= m || newY >= n) continue;

// 2). continue if out of elimation

int newK = curr[2] + grid[newX][newY];

if (newK > k) continue;

// 3). continue if we have more optimal result

int newDist = curr[3] + 1;

if (dists[newX][newY][newK] <= newDist) continue;

dists[newX][newY][newK] = newDist;

heap.offer(new int[]{newX, newY, newK, newDist});

}

}

int res = dists[m-1][n-1][0];

for (int i = 1; i <= k; i++) res = Math.min(res, dists[m-1][n-1][i]);

return (res == Integer.MAX\_VALUE) ? -1 : res;

}

}

**Style 2: From ChatGPT**

class Solution {

    public int shortestPath(int[][] grid, int k) {

        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[][][] distances = new int[m][n][k + 1];

        for(int[][] distance : distances) {

            for(int[] d : distance) {

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

            }

        }

        // Initialize distance for all potential removing obstacles cases(0 ~ k) as 0

        Arrays.fill(distances[0][0], 0);

        // min-heap storing {i, j, # obstacles eliminated, distance}, sorted by distance to (0,0)

        // Similar strategy as L743.Network Delay Time

        PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> a[3] - b[3]);

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

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

        while(!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

            // When the target cell {m - 1, n - 1} is reached with any number

            // of obstacles(0 ~ k) removed within the allowed limit, the first

            // case returned will surely be the minimum distance since its

            // reaching the goal as first one

            if(cur[0] == m - 1 && cur[1] == n - 1) {

                return cur[3];

            }

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

                int new\_x = cur[0] + dx[i];

                int new\_y = cur[1] + dy[i];

                // Must in boundary

                    int new\_k = cur[2] + grid[new\_x][new\_y];

                    // Must under # obstacles eliminated limit

                    if(new\_k <= k) {

                        // Continue if we have more optimal result:

                        // Under same number of obstacles removed case, if the

                        // new path(new\_distance) able to reach longer distance

                        // than existing path(distances[new\_x][new\_y][new\_k]),

                        // we will update current path to new path, which means

                        // addding the node into minPQ

                        int new\_distance = cur[3] + 1;

                        if(distances[new\_x][new\_y][new\_k] > new\_distance) {

                            distances[new\_x][new\_y][new\_k] = new\_distance;

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

                        }

                    }

                }

            }

        }

        // If not able to find a path to reach {m - 1, n - 1}

        // just return -1

        return -1;

    }

}

**Refer to ChatGPT**

Leetcode problem 1293, "Shortest Path in a Grid with Obstacles Elimination," requires finding the shortest path in a grid while being allowed to eliminate a certain number of obstacles. Dijkstra's algorithm can be adapted to solve this problem by treating each state (position in the grid, number of obstacles removed) as a node in the graph.

Here’s the approach to solving the problem using a modified version of Dijkstra’s algorithm:

**Explanation**

State Representation:

Each state is represented as (row, col, obstacles\_removed), where row and col are the current position in the grid and obstacles\_removed is the number of obstacles removed so far.

Priority Queue:

Use a priority queue to always expand the node with the smallest path cost first. The path cost here is the number of steps taken.

Distance Array:

Maintain a 3D distance array dist[row][col][obstacles\_removed] to store the shortest path to each state.

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 with any number of obstacles removed within the allowed limit.

**Java Implementation**

Here's the Java code implementing the above approach using a priority queue:

class Solution {

    static class State implements Comparable<State> {

        int row, col, steps, obstaclesRemoved;

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

            this.row = row;

            this.col = col;

            this.steps = steps;

            this.obstaclesRemoved = obstaclesRemoved;

        }

        @Override

        public int compareTo(State other) {

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

        }

    }

    public int shortestPath(int[][] grid, int k) {

        int rows = grid.length;

        int cols = grid[0].length;

        int[][][] dist = new int[rows][cols][k + 1];

        for (int[][] matrix : dist) {

            for (int[] array : matrix) {

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

            }

        }

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

        pq.add(new State(0, 0, 0, 0));

        dist[0][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.steps;

            }

            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 <= k && current.steps + 1 < dist[newRow][newCol][newObstaclesRemoved]) {

                        dist[newRow][newCol][newObstaclesRemoved] = current.steps + 1;

                        pq.add(new State(newRow, newCol, current.steps + 1, newObstaclesRemoved));

                    }

                }

            }

        }

        return -1;  // If the target is not reachable

    }

}

**Key Points:**

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

Priority Queue: The priority queue ensures that the state with the smallest number of steps is processed first.

Distance Array: The 3D distance array keeps track of the shortest path to each cell for each possible number of obstacles removed.

**Benefits:**

Optimal Path: Dijkstra’s algorithm ensures the shortest path 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 respecting the constraints on the number of obstacles that can be eliminated.

**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.

**Refer to**

[L2290.Minimum Obstacle Removal to Reach Corner (Ref.L1293)](note://WEBcd918985c5097f42e558acf5ab25def8)

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

[L743.Network Delay Time](note://17E4C21F015F4605AAD4D6878F0D0A67)