<https://leetcode.com/problems/disconnect-path-in-a-binary-matrix-by-at-most-one-flip/description/>

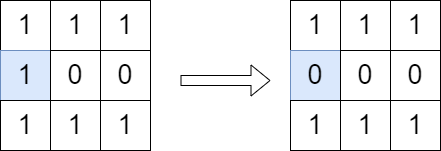
You are given a 0-indexed m x n binary matrix grid. You can move from a cell (row, col) to any of the cells (row + 1, col) or (row, col + 1) that has the value 1. The matrix is disconnected if there is no path from (0, 0) to (m - 1, n - 1).

You can flip the value of at most one (possibly none) cell. You cannot flip the cells (0, 0) and (m - 1, n - 1).

Return true if it is possible to make the matrix disconnect or false otherwise.

Note that flipping a cell changes its value from 0 to 1 or from 1 to 0.

**Example 1:**

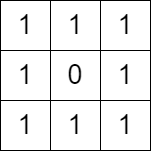


**Input:** grid = [[1,1,1],[1,0,0],[1,1,1]]

**Output:** true

**Explanation:** We can change the cell shown in the diagram above. There is no path from (0, 0) to (2, 2) in the resulting grid.

**Example 2:**



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

**Output:** false

**Explanation:** It is not possible to change at most one cell such that there is not path from (0, 0) to (2, 2).

**Constraints:**

m == grid.length

n == grid[i].length

1 <= m, n <= 1000

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

grid[i][j] is either 0 or 1.

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

**Attempt 1: 2024-12-16**

**Solution 1: DFS two times (360 min)**

**Style 1: Use Short-Circuiting statement as helper(grid, x + 1, y, m, n) || helper(grid, x, y + 1, m, n)**

class Solution {

    public boolean isPossibleToCutPath(int[][] grid) {

        int m = grid.length;

        int n = grid[0].length;

        // Perform a DFS from the top-left corner before the cut

        boolean pathExistBeforeFlip = helper(grid, 0, 0, m, n);

        // If no path initially at all, no flip required and already disconnected

        if(!pathExistBeforeFlip) {

            return true;

        }

        // Restore start and end position to make it accessible

        grid[0][0] = 1;

        grid[m - 1][n - 1] = 1;

        // Perform a DFS from the top-left corner after making the cut

        boolean pathExistAfterFlip = helper(grid, 0, 0, m, n);

        // If before flip we can find path, but after flip not, then

        // able to make the matrix disconnect

        return pathExistBeforeFlip && !pathExistAfterFlip;

    }

    private boolean helper(int[][] grid, int x, int y, int m, int n) {

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

            return true;

        }

        if(x >= m || y >= n || grid[x][y] == 0) {

            return false;

        }

        // Flip all ecountering cell on path from 1 to 0

        grid[x][y] = 0;

        // Short-Circuiting statement

// Recursively search in the right and down directions, return true if a path is found

        return helper(grid, x + 1, y, m, n) || helper(grid, x, y + 1, m, n);

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(1)

**Style 2: Use regular for loop statement to represent helper(grid, x + 1, y, m, n) || helper(grid, x, y + 1, m, n)**

class Solution {

    public boolean isPossibleToCutPath(int[][] grid) {

        int m = grid.length;

        int n = grid[0].length;

        // Perform a DFS from the top-left corner before the cut

        boolean pathExistBeforeFlip = helper(grid, 0, 0, m, n);

        // If no path initially at all, no flip required and already disconnected

        if(!pathExistBeforeFlip) {

            return true;

        }

        // Restore start and end position to make it accessible

        grid[0][0] = 1;

        grid[m - 1][n - 1] = 1;

        // Perform a DFS from the top-left corner after making the cut

        boolean pathExistAfterFlip = helper(grid, 0, 0, m, n);

        // If before flip we can find path, but after flip not, then

        // able to make the matrix disconnect

        return pathExistBeforeFlip && !pathExistAfterFlip;

    }

    private boolean helper(int[][] grid, int x, int y, int m, int n) {

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

            return true;

        }

        if(x >= m || y >= n || grid[x][y] == 0) {

            return false;

        }

        // Flip all ecountering cell on path from 1 to 0

        grid[x][y] = 0;

        // Use regular for loop statement to represent

        // helper(grid, x + 1, y, m, n) || helper(grid, x, y + 1, m, n)

        int[] dx = {0, 1};

        int[] dy = {1, 0};

        for(int k = 0; k < 2; k++) {

            if(helper(grid, x + dx[k], y + dy[k], m, n)) {

                return true;

            }

        }

        return false;

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/disconnect-path-in-a-binary-matrix-by-at-most-one-flip/solutions/3141656/explained-run-dfs-2-times-very-simple-and-easy-to-understand-solution/>

**Approach**

There are two cases:

When there is no path at all from (0,0) to (m-1, n-1) position, so simply return true;

When there is one path but after fliping one value we will not able to reach the destination.

In the first traversal we will simply flip **ALL** travesed node to 0

By doing this, if there is any alternate path thats not depends on the first one then its not possible to flip a single value to 0 and we won't be reaching.

**Complexity**

Time complexity: O(M \* N)

Space complexity: O(1)

class Solution {

public:

bool dfs(vector<vector<int>> &grid, int i, int j){

if(i+1 == grid.size() && j+1 == grid[0].size()) return true;

if(i >= grid.size() || j >= grid[0].size() || grid[i][j] == 0) return false;

grid[i][j] = 0;

return dfs(grid, i+1, j) || dfs(grid, i, j+1);

}

bool isPossibleToCutPath(vector<vector<int>>& grid) {

if(dfs(grid, 0, 0) == false) return true;

grid[0][0] = 1;

return !dfs(grid, 0, 0);

}

};

**Java version**

class Solution {

    private int[][] grid; // Store the provided grid

    private int rows; // Number of rows in the grid

    private int cols; // Number of columns in the grid

    // Determines if it's possible to cut a path from top-left to bottom-right

    public boolean isPossibleToCutPath(int[][] grid) {

        this.grid = grid;

        rows = grid.length; // Initialize number of rows

        cols = grid[0].length; // Initialize number of columns

        // Perform a DFS from the top-left corner before the cut

        boolean pathExistsBeforeCut = depthFirstSearch(0, 0);

        // Simulate the cut by marking the start and end points as obstacles

        grid[0][0] = 1;

        grid[rows - 1][cols - 1] = 1;

        // Perform a DFS from the top-left corner after making the cut

        boolean pathExistsAfterCut = depthFirstSearch(0, 0);

        // If the path exists before and not after, then it is possible to cut the path

        return !(pathExistsBeforeCut && pathExistsAfterCut);

    }

    // Helper method to perform depth-first search

    private boolean depthFirstSearch(int row, int col) {

        // Check boundaries and whether the current cell is an obstacle

        if (row >= rows || col >= cols || grid[row][col] == 0) {

            return false;

        }

        // Check if we've reached the bottom-right corner

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

            return true;

        }

        // Mark the current cell as an obstacle to avoid revisiting

        grid[row][col] = 0;

        // Recursively search in the right and down directions, returning true if a path is found

        return depthFirstSearch(row + 1, col) || depthFirstSearch(row, col + 1);

    }

}

**The above implementation is Correct, but the logic is Incorrect!**

**Any random path removing logic is incorrect, the real reason behind why flip DFS1 whole path (not like description required only flip 1 cell allowed) still works because we can only move to right and down, NOT a random 4 directions conventional DFS, the DFS1 path is deterministic, either go right or go down first and leads two major group of potential paths (either stick to top & right or stick to bottom & left), if they have NO single mutual used (critical) cell, then no way to block all these paths by just flipping one cell, so just flip all cells on DFS1 path, which cancel any potential mutual used (critical) cell, then when run DFS again as DFS2 if still have a path till target cell, it means no mutual used (critical) cell exist, then no way to block.**

**Refer to**

<https://leetcode.com/problems/disconnect-path-in-a-binary-matrix-by-at-most-one-flip/solutions/3141656/explained-run-dfs-2-times-very-simple-and-easy-to-understand-solution/comments/1787822>

<https://leetcode.com/problems/disconnect-path-in-a-binary-matrix-by-at-most-one-flip/solutions/3141656/explained-run-dfs-2-times-very-simple-and-easy-to-understand-solution/comments/1813905>

**Example why any random path removing logic is incorrect**

This Matrix - Rows (N) = 3 , Cols (M) = 4

**1** 1 0 0

**1 1 1 1**

1 1 1 **1**

We will have to reach from (0, 0) to (2, 3)

Assume the following path for this Matrix for DFS1

(0, 0), **(1, 0), (1, 1), (1, 2), (1, 3)**, (2, 3)

So now you are setting these positions (except (0,0) and (2,3) to Zero according to DFS1 logic.

Now the matrix would be

1 1 0 0

0 0 0 0

1 1 1 1

**So for DFS2 iteration it will not get any valid path and will return TRUE**

**but it should return FALSE.**

**But WHY your implementation is correct ?**

**It is because your Code is not VISITING NODES RANDOMLY.**

You are picking like DOWN DOWN RIGHT ( if and only if valid DOWN not found) ..... and so on. or (VICE VERSA)

**This BOUNDARY WISE FIRST VALID PATH CHECKING making sure that your will yield correct result.**

The reason that always going down first works because we're always sticking to the side.

In the example shown, we're not choosing to stick to the "edge" of our graph. We're cutting clean through the middle of the graph where it'll cut off other paths, which is the issue.

We don't want to unnecessary cut off other paths. We want a path that avoids others paths as much as possible.

This is done through always picking going down first over going right, meaning that your graph is basically sticking to the edge of the graph as much as possible, meaning it steers clear of other paths as much as possible.

**Priority of directions**

**Refer to**

<https://leetcode.com/problems/disconnect-path-in-a-binary-matrix-by-at-most-one-flip/solutions/3154733/priority-of-directions/>

Our search starts from points [0, 1] - right, and [1, 0] - down. We search for two non-intersecting paths from those points.

For the first search, we try to go right first. We go down only if we cannot reach the last cell by going right.

For the second search, the priority of direction is flipped - we go down first.

That way, the first path is closest to the top-right borders of the grid, and the second path - to the bottom-left border.

If we find both paths, then we cannot make the matrix disconnected.

When we traverse a path, we change values in the matrix to zero, so we cannot reuse the same cell the second time.

As Chasey noted, since we flip values for the first path, we do not even need to flip directions for the second one.

This simplifies the code, and paves a way for generalized problem when we need to find more than two paths.

bool dfs(vector<vector<int>>& g, int i, int j) {

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

if (i == m || j == n || g[i][j] == 0)

return false;

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

return true;

if (i != 0 || j != 0)

g[i][j] = 0;

return dfs(g, i, j + 1) || dfs(g, i + 1, j);

}

bool isPossibleToCutPath(vector<vector<int>>& g) {

return !(dfs(g, 0, 0) && dfs(g, 0, 0));

}

**Refer to chatGPT**

i have below 2 code section ?

**Section 1:**

int[] dx = {0, 1};

int[] dy = {1, 0};

boolean result = false;

for(int k = 0; k < 2; k++) {

result |= helper(grid, x + dx[k], y + dy[k], m, n);

}

return result;

**Section 2:**

return helper(grid, x + 1, y, m, n) || helper(grid, x, y + 1, m, n);

**Looks like Section 1 not equal to Section 2 ?**

You are correct—**Section 1 and Section 2 are not equivalent** in terms of behavior due to the difference in how they evaluate results and short-circuiting.

**Section 1:**

int[] dx = {0, 1};

int[] dy = {1, 0};

boolean result = false;

for (int k = 0; k < 2; k++) {

result |= helper(grid, x + dx[k], y + dy[k], m, n);

}

return result;

Logical OR with **|=**:

The |= operator evaluates both sides of the operation, even if the first result is true.

This is equivalent to:

result = result | helper(grid, x + dx[k], y + dy[k], m, n);

Here, **both directions** (right and down) are explored **even if one returns true**.

Implication:

The loop explores **all possible directions** (right → k=0 and down → k=1).

It does not stop when a true result is found; it continues evaluating both paths.

**Section 2:**

return helper(grid, x + 1, y, m, n) || helper(grid, x, y + 1, m, n);

Short-Circuiting **||**:

The || (logical OR) operator short-circuits.

If helper(grid, x + 1, y, m, n) evaluates to true, the second condition helper(grid, x, y + 1, m, n) **is not evaluated** at all.

Implication:

This code stops as soon as one of the recursive calls returns true.

It does not explore both paths if one path is already successful.

**Summary of Differences:**

|  |  |  |
| --- | --- | --- |
| Aspect | Section 1 | Section 2 |
| Operator Used | ` | =` (bitwise OR) |
| Path Exploration | Explores  **both directions**  always | Stops after the  **first true**  path |
| Efficiency | Less efficient due to redundant calls | More efficient with short-circuiting |

**Impact on Behavior:**

If the helper function has side effects or is computationally expensive, Section 1 will **evaluate both directions unnecessarily**, whereas Section 2 avoids unnecessary calls by short-circuiting.

In some recursive problems, Section 1 may work correctly but perform redundant computations, while Section 2 is optimized for early termination.

**Example to Illustrate:**

Suppose helper has print statements to demonstrate the calls:

**Code:**

boolean helper(int[][] grid, int x, int y, int m, int n) {

System.out.println("Checking: " + x + ", " + y);

return (x == m - 1 && y == n - 1); // Base case

}

**Section 1 Output:**

Checking: 1, 0

Checking: 0, 1

Both calls are made.

**Section 2 Output:**

Checking: 1, 0

Second call is skipped because the first path already returned true.

**Conclusion:**

Section 2 is **more efficient** due to short-circuiting.

Section 1 explores both paths fully, which can be redundant and inefficient in cases where early termination is desired.

**How to update Section 1 to make it equal to Section 2 ?**

To make **Section 1** equivalent to **Section 2**, you need to introduce **short-circuit behavior** into the loop. This can be done by using a return statement inside the loop when a true value is found, or by explicitly checking the result.

Here's the updated version of **Section 1**:

**Updated Section 1:**

int[] dx = {0, 1};

int[] dy = {1, 0};

for (int k = 0; k < 2; k++) {

if (helper(grid, x + dx[k], y + dy[k], m, n)) {

return true; // Short-circuit: Return immediately if `true` is found

}

}

return false; // Return false if neither path returns true

**Key Changes:**

Short-Circuiting:

If helper returns true for any direction, you immediately return true and stop further exploration.

This replicates the behavior of || in Section 2.

Avoid Redundant Exploration:

The loop stops as soon as one path succeeds, preventing unnecessary recursive calls.

**Why It Works:**

The if condition ensures that as soon as true is encountered, the method returns true immediately.

If no direction leads to success, the loop completes, and the method returns false, just like Section 2.

**Behavior Comparison After Update:**

|  |  |  |
| --- | --- | --- |
| Aspect | Updated Section 1 | Section 2 |
| Path Exploration | Stops after the first  true  path | Stops after the first  true  path |
| Short-Circuiting | if  condition inside the loop | ` |
| Efficiency | Optimized, avoids redundant calls | Optimized, avoids redundant calls |

**Final Thoughts:**

This updated version of Section 1 now behaves identically to Section 2 and achieves the same short-circuit efficiency.

**Refer to**

[L1074.Number of Submatrices That Sum to Target (Ref.L560)](note://WEB47e31ed01fd0cb1f8aa407face798345)

[L1368.Minimum Cost to Make at Least One Valid Path in a Grid](note://WEB4b3aa9003dbf5915537540decf09b434)

[L1568.Minimum Number of Days to Disconnect Island](note://WEBe1f9dea2f2571376df84b64404041136)

[L2203.Minimum Weighted Subgraph With the Required Paths](note://WEBa77cc0b31467a4612ff75d4806548614)