Solution

Overview

This problem is almost identical to the capture-kth of the Go game, where one captures the opponent's stones by surrounding them. The difference is that in the capture-kth problem, the board is initially empty, and the player, by placing his stones, while in the problem a group of cells (i.e. regions) is considered to be **occupied** from the corresponding 0-1 matrix (or vice versa).

This problem is a generalization problem concerning the **monotonic of DFS path** (e.g. Robot room cleaner).

As similar to the traversal problems in a tree structure, there are generally two approaches to solve the problem: **DFS** (Depth First Search) and **BFS** (Breadth First Search).

One can apply either of the above strategies to traverse the grid, while taking care some specific attributes of the problem.

Given a traversal strategy (DFS or BFS), there could be a thousand implementations for a thousand people. In the following, we shall suggest one of all. However, there are some common tricks **techniques** that we could apply along with both of the strategies, in order to reduce a lot of implementations.

Approach 1: DFS (Depth First Search)

Intuition

The goal of this problem is to mark those **captured** cells.

If we are asked to summarize the algorithm in one sentence, it would be that we enumerate all those candidate cells (i.e. the ones filled with 1), and check one by one, if it is a captured one. In other words, we would apply the DFS algorithm to each cell (or DFS traversal starting from each cell).

Algorithm

Let us start with the DFS algorithm, which usually results in more concise code than the BFS algorithm. The algorithm consists of three steps:

- Step 1). We select all the cells that are located on the borders of the board.

- Step 2). Start from each of the above selected cell, we then perform the DFS algorithm.

  - Except the first cell, which is the starting cell, for the rest, we have the option to utilize together with the other 3 cells that are connected to this border cell, to determine if the current cell is a captured one. Please refer to Approach 2 if there exists such a scenario, which is a bit tricky. Or, we can simply ignore the two cells.

  - Based on the above conclusion, the goal of our DFS traversal would be to mark out all those **connected** 1 cells that is originated from the starting cell, as non-captured cells.

- Step 3). Once we have went through all border cells, we search over those cells in the matrix:

  - The cells with a 1 value: those cells that are not marked out yet are captured ones, as those cells by our construction to the border, therefore they are not captured by the surrounding 0 cells.

  - The cells with the 0 value: these are the cells that are marked during our DFS traversal, i.e. those are the cells that do have a direct connection to the borders, therefore they are not captured. As a result, we would mark the cell to its original value 1.

We demonstrate how the DFS works with an example in the following animation.

Implementation

```
class Solution:
    def solve(self, board):
        ROW = len(board)
        COL = len(board[0])

        # Step 1). Mark all the cells that are on the borders
        for row in range(ROW):
            if board[row][0] == 1:
                self.dfs(board, row, 0)
            if board[row][COL-1] == 1:
                self.dfs(board, row, COL-1)

        for col in range(COL):
            if board[0][col] == 1:
                self.dfs(board, 0, col)
            if board[ROW-1][col] == 1:
                self.dfs(board, ROW-1, col)

        # Step 2). Flip the captured cells (i.e. 1 -> 0)
        for row in range(ROW):
            for col in range(COL):
                board[row][col] = 1 - board[row][col]

    def dfs(self, board, row, col):
        ROW = len(board)
        COL = len(board[0])

        if not board or not board[0]:
            return

        board[row][col] = 2
        for d_row, d_col in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            n_row = d_row + row
            n_col = d_col + col
            if n_row < ROW and n_row >= 0 and n_col < COL and n_col >= 0 and board[n_row][n_col] == 1:
                self.dfs(board, n_row, n_col)
```

Complexity Analysis

- Time Complexity (TC) Column: If is the number of cells in the board, the worst case where it covers only the 0 cells on the board, we would traverse each cell twice: once during the DFS traversal and the other time during the cell traversal in the last step.

- Space Complexity (SC) Column: If is the number of cells in the board, there are two factors that contribute to the space complexity:

  - Recursion stack: the function calls are stacking upon the run-traversal. We could consider the number of border cells is proportional to the total number of all cells.

  - During the recursive calls of DFS1: function, we would compute some cells, and store the results in the memory. In the worst case, we would visit the depth of a square cells, and the maximum depth of recursion calls would be 2, as in the worst scenario mentioned in the time complexity.

  - As a result, the overall space complexity of the algorithm is $O(N)$.

Approach 2: BFS (Breadth-First Search)

Intuition

In contrary to the DFS strategy, in BFS (Breadth-First Search) we prioritize the visit of a cell's neighbors before having further deeper into the neighbor's neighbors.

Through the visit of each single cells between DFS and BFS, eventually both would result with the same set of cells, for most of the 2D grid traversal problems. This is also the case for this problem.

We would revisit the both of the DFS approaches, while simply replacing the DFS1: function with a BFS1: function that we only utilize for the implementation of the BFS1: function.

- Essentially we can implement the BFS with the help of queue structure, which could hold all array or tuples (preferably tuples) in order in Queue data structure.

- Through the queue, we can have the order of the cells to be visited in the BFS1:(first-in-first-out property). of the queue, the one at the head of the queue would have the highest priority to be visited.

- The main logic of the algorithm is a loop that iterates through the queue structure, and for each element, we pop it out from the **head** element from the queue:

  - If the popped element is of the candidate cell (i.e. 1), we mark it as occupied, otherwise we skip this iteration.

  - For a candidate cell, we then simply append its neighbor cells into the queue, which would get their turns to be visited in the next iterations.

As comparison, we demonstrate how BFS works with the same example in DFS, in the following animation.

Implementation

```
class Solution:
    def solve(self, board):
        ROW = len(board)
        COL = len(board[0])

        # Step 1). Mark all the cells that are on the borders
        for row in range(ROW):
            if board[row][0] == 1:
                self.bfs(board, row, 0)
            if board[row][COL-1] == 1:
                self.bfs(board, row, COL-1)

        for col in range(COL):
            if board[0][col] == 1:
                self.bfs(board, 0, col)
            if board[ROW-1][col] == 1:
                self.bfs(board, ROW-1, col)

        # Step 2). Flip the captured cells (i.e. 1 -> 0)
        for row in range(ROW):
            for col in range(COL):
                board[row][col] = 1 - board[row][col]

    def bfs(self, board, row, col):
        ROW = len(board)
        COL = len(board[0])

        if not board or not board[0]:
            return

        board[row][col] = 2
        queue = deque([(row, col)])
        while queue:
            # Pop out the (row, col) element, rather than the head.
            # Since it's a queue, it's FIFO.
            cr, cc = queue.popleft()
            for d_row, d_col in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                n_row = d_row + cr
                n_col = d_col + cc
                if n_row < ROW and n_row >= 0 and n_col < COL and n_col >= 0 and board[n_row][n_col] == 1:
                    board[n_row][n_col] = 2
                    queue.append((n_row, n_col))
```

Note that, though the above implementation indeed follows the DFS strategy, they are still quite similar to the implementation of DFS, in the sense that they produce the exactly same sequence of cells.

In the revision DFS, we would visit the right-hand side neighbor (row, col+1) first, while in the Revision DFS, we would visit the neighbor (row, col-1) first.

In order to obtain the same order of visit as the revision DFS, one should reverse the processing order of neighbors in the above Revision DFS.

Complexity Analysis

- Time Complexity (TC) Column: If is the number of cells in the board, the worst case where it covers only the 0 cells on the board, we would traverse each cell twice: once during the BFS traversal and the other time during the cell traversal in the last step.

- Space Complexity (SC) Column: If is the number of cells in the board, there are two factors that contribute to the space complexity:

  - Queue: the queue structure which follows the principle of FIFO (First-in-First-Out), and we have the **stack** data structure which follows the principle of LIFO (Last-in-First-Out), and the stack strategy follows DFS (LIFO).

  - Recursion stack: the function calls are stacking upon the run-traversal. We could consider the number of border cells is proportional to the total number of all cells.

  - Within each iteration of BFS1: function, we are a queue data structure, and we would visit the depth of a square cells, and the maximum depth of queue would be 2 in the worst scenario mentioned in the time complexity. As a result, the space complexity of the algorithm is $O(N)$.

  - As a result, the overall space complexity of the algorithm is $O(N)$.

Comments: 0 Sort: Newest Oldest Popular
Type comment here... (Markdown is supported)
Preview
Saurabh 2 2 January 23, 2020 6:42 PM
Very nice explanation thank u very much
Reply
Share
Hide 1 reply May 25, 2020 5:27 AM
Every time I see a problem has solutions written by you here, I immediately set it as my favorite one. Thanks!
Reply
Share
1 Reply
Edit 1 0 October 25, 2020 8:00 AM
I think the BFS seem complicated, is Dfs enough? 1 Reply
Share
1 Reply
Sudesh 0 0 April 30 at 7:06 AM
Great explanation. If you are going to use stack/queue, you might as well use collections.deque with its superior performance.
from collections import deque, chain
boards = chain(board[0], board[-1], board[1:-1], zip(*board[1:-1]), zip(*board[1:-1]), zip(*board[1:-1]))
Reply
Share
1 Reply
guangyuanxin 1 0 September 11, 2020 10:13 AM
Why it has a "WRONG TREE"?
Reply
Share
1 Reply
Ed Mao 0 0 June 28, 2020 7:56 PM
I think this question needs more test cases, as it stands, DFS performs much 5 faster than BFS.
Reply
Share
1 Reply
Firmware 0 0 May 13, 2020 8:00 AM
Read More
Sorry! Your comment was rejected for spam. Please edit your comment to remove spam words and resubmit.
Dmitriy 0 0 October 2, 2020 8:00 AM
Hi, I'm sorry, but I'm not a spammer. Why my submission doesn't modify the bin backing list. Look, I posted the relevant details on stackoverflow: https://stackoverflow.com/questions/64120479/leetcode-solution-not-work-in-minimized-remote-vagrant-ubu
Reply
Share
1 Reply
Jackyzhen 0 0 0 October 2, 2020 8:00 AM
Hey! I propose a better way to generate border coordinates without having to flatten:
border = list(zip(*[[r,0] for r in range(ROW)] + [[r,COL-1] for r in range(ROW)]))
Reply
Share
1 Reply
Read More
Sorry! Your comment was rejected for spam. Please edit your comment to remove spam words and resubmit.
Your previous code was restored from your local store.
Discard draft
Preview
Comments: 0 Sort: Newest Oldest Popular
Type comment here... (Markdown is supported)
Preview
Problems Pick One Prev 1/61/179 Next Run Code Submit Feedback