Previous Next

130. Surrounded Regions

Jan. 23, 2020 | 22.3K views

Average Rating: 4.74 (38 votes)

Given a 2D board containing `'X'` and `'O'` (**the letter O**), capture all regions surrounded by `'X'`.

A region is captured by flipping all `'O'`s into `'X'`s in that surrounded region.

Example:

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

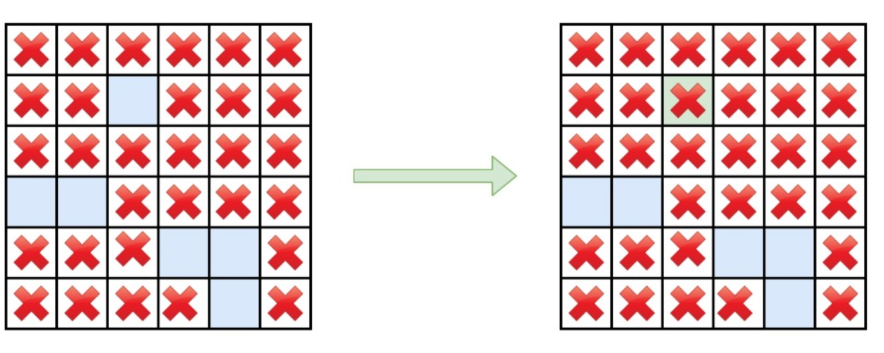
Explanation:

Surrounded regions shouldn't be on the border, which means that any `'O'` on the border of the board are not flipped to `'X'`. Any `'O'` that is not on the border and it is not connected to an `'O'` on the border will be flipped to `'X'`. Two cells are connected if they are adjacent cells connected horizontally or vertically.

Solution

Overview

This problem is *almost* identical as the capture rule of the Go game, where one captures the opponent's stones by surrounding them. The difference is that in the Go game the borders of the board are considered to the walls that surround the stones, while in this problem a group of cells (*i.e.* region) is considered to be ***escaped*** from the surrounding if it reaches any border.



This problem is yet another problem concerning the ***traversal of 2D grid***, *e.g.* Robot room cleaner.

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

One can apply either of the above strategies to traverse the 2D grid, while taking some specific actions to resolve the problems.

Given a traversal strategy (*DFS* or *BFS*), there could be a thousand implementations for a thousand people, if we indulge ourselves to exaggerate a bit. However, there are some common neat ***techniques*** that we could apply along with both of the strategies, in order to obtain a more optimized solution.

Approach 1: DFS (Depth-First Search)

Intuition

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

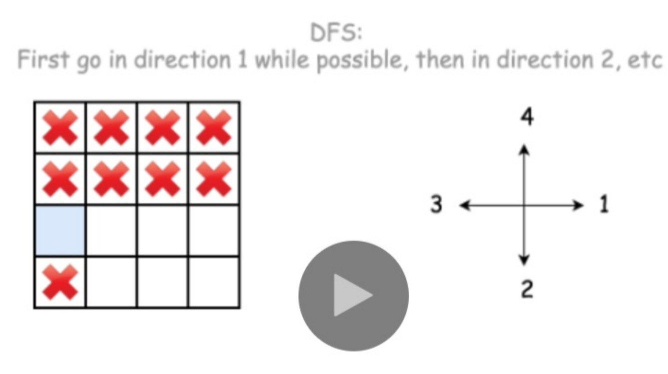
If we are asked to summarize the algorithm in one sentence, it would be that we enumerate all those candidate cells (*i.e.* the ones filled with `O`), and check *one by one* if they are *captured* or not, *i.e.* we start with a candidate cell (`O`), and then apply either DFS or BFS strategy to explore its surrounding cells.

Algorithm

Let us start with the DFS algorithm, which usually results in a 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* traversal.
  - If a cell on the border happens to be `O`, then we know that this cell is *alive*, together with the other `O` cells that are *connected* to this border cell, based on the description of the problem. Two cells are *connected*, if there exists a path consisting of only `O` letter that bridges between the two cells.
  - Based on the above conclusion, the goal of our DFS traversal would be to *mark* out all those ***connected*** `O` cells that is originated from the border, with any distinguished letter such as `E`.
- Step 3). Once we iterate through all border cells, we would then obtain three types of cells:
  - The one with the `X` letter: the cell that we could consider as the wall.
  - The one with the `O` letter: the cells that are spared in our *DFS* traversal, *i.e.* these cells has no connection to the border, therefore they are ***captured***. We then should replace these cell with `X` letter.
  - The one with the `E` letter: these are the cells that are marked during our DFS traversal, *i.e.* these are the cells that has at least one connection to the borders, therefore they are not *captured*. As a result, we would revert the cell to its original letter `O`.

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



1 / 8

Java Python Copy

```python
class Solution(object):
    def solve(self, board):
        """
        :type board: List[List[str]]
        :rtype: None Do not return anything, modify board in-place instead.
        """
        if not board or not board[0]:
            return

        self.ROWS = len(board)
        self.COLS = len(board[0])

        # Step 1). retrieve all border cells
        from itertools import product
        borders = list(product(range(self.ROWS), [0, self.COLS-1])) \
                + list(product([0, self.ROWS-1], range(self.COLS)))

        # Step 2). mark the "escaped" cells, with any placeholder, e.g. 'E'
        for row, col in borders:
            self.DFS(board, row, col)

        # Step 3). flip the captured cells ('O'->'X') and the escaped one ('E'->'O')
        for r in range(self.ROWS):
            for c in range(self.COLS):
                if board[r][c] == 'O':   board[r][c] = 'X'  # captured
                elif board[r][c] == 'E': board[r][c] = 'O'  # escaped

```

Optimizations

In the above implementation, there are a few techniques that we applied ***under the hood***, in order to further optimize our solution. Here we list them one by one.

> Rather than iterating all candidate cells (the ones filled with `O`), we check only the ones on the ***borders***.

In the above implementation, our starting points of *DFS* are those cells that meet two conditions: 1). on the border. 2). filled with `O`.

As an alternative solution, one might decide to iterate all `O` cells, which is less optimal compared to our starting points.

As one can see, during DFS traversal, the alternative solution would traverse the cells that eventually might be captured, which is not necessary in our approach.

> Rather than using a sort of `visited[cell_index]` map to keep track of the visited cells, we simply mark visited cell ***in place***.

This technique helps us gain both in the space and time complexity.

As an alternative approach, one could use a additional data structure to keep track of the visited cells, which goes without saying would require additional memory. And also it requires additional calculation for the comparison. Though one might argue that we could use the hash table data structure for the `visited[]` map, which has the $\mathcal{O}(1)$ asymptotic time complexity, but it is still more expensive than the simple comparison on the value of cell.

> Rather than doing the boundary check within the `DFS()` function, we do it ***before*** the invocation of the function.

As a comparison, here is the implementation where we do the boundary check within the `DFS()` function.

Python Copy

```python
def DFS(self, board, row, col):
    if row < 0 or row >= self.ROWS or col < 0 or col >= self.COLS:
        return
    if board[row][col] != 'O':
        return
    board[row][col] = 'E'
    # jump to the neighbors without boundary checks
    for ro, co in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
        self.DFS(board, row+ro, col+co)
```

This measure reduces the number of recursion, therefore it reduces the overheads with the function calls.

As trivial as this modification might seem to be, it actually reduces the runtime of the Python implementation from *148 ms* to *124 ms*, *i.e.* *16%* of reduction, which beats 97% of submissions instead of 77% at the moment.

Complexity Analysis

- Time Complexity: $\mathcal{O}(N)$ where N is the number of cells in the board. In the worst case where it contains only the `O` cells on the board, we would traverse each cell twice: once during the DFS traversal and the other time during the cell reversion in the last step.
- Space Complexity: $\mathcal{O}(N)$ where N is the number of cells in the board. There are mainly two places that we consume some additional memory.
  - We keep a list of border cells as starting points for our traversal. We could consider the number of border cells is proportional to the total number (N) of cells.
  - During the recursive calls of `DFS()` function, we would consume some space in the function call stack, *i.e.* the call stack will pile up along with the depth of recursive calls. And the maximum depth of recursive calls would be N as in the worst scenario mentioned in the time complexity.
  - As a result, the overall space complexity of the algorithm is $\mathcal{O}(N)$.

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 moving further (deeper) into the neighbor's neighbor.

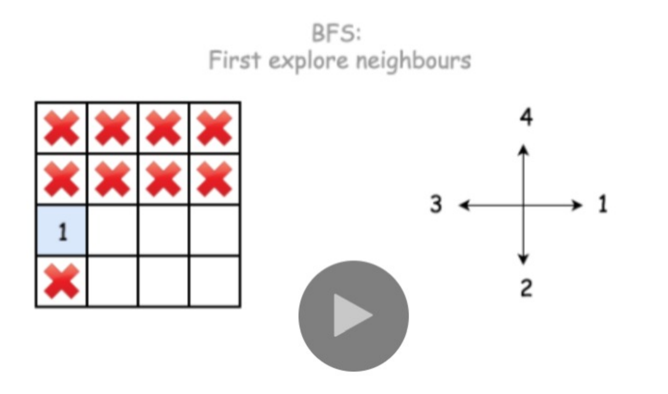
Though the order of visit might differ between DFS and BFS, eventually both strategies would visit the same set of cells, for most of the 2D grid traversal problems. This is also the case for this problem.

Algorithm

We could reuse the bulk of the DFS approach, while simply replacing the `DFS()` function with a `BFS()` function. Here we just elaborate the implementation of the `BFS()` function.

- Essentially we can implement the BFS with the help of queue data structure, which could be of `Array` or more preferably `LinkedList` in Java or `Deque` in Python.
- Through the queue, we maintain the order of visit for the cells. Due to the **FIFO** (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 above-mentioned queue. At each iteration of the loop, we *pop out* the **head** element from the queue.
  - If the popped element is of the candidate cell (*i.e.* `O`), we mark it as escaped, 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.



1 / 9

Java Python Copy

```python
class Solution(object):
    def solve(self, board):
        """
        :type board: List[List[str]]
        :rtype: None Do not return anything, modify board in-place instead.
        """
        if not board or not board[0]:
            return

        self.ROWS = len(board)
        self.COLS = len(board[0])

        # Step 1). retrieve all border cells
        from itertools import product
        borders = list(product(range(self.ROWS), [0, self.COLS-1])) \
                + list(product([0, self.ROWS-1], range(self.COLS)))

        # Step 2). mark the "escaped" cells, with any placeholder, e.g. 'E'
        for row, col in borders:
            #self.DFS(board, row, col)
            self.BFS(board, row, col)

        # Step 3). flip the captured cells ('O'->'X') and the escaped one ('E'->'O')
        for r in range(self.ROWS):
            for c in range(self.COLS):
                if board[r][c] == 'O':   board[r][c] = 'X'  # captured
                elif board[r][c] == 'E': board[r][c] = 'O'  # escaped
```

From BFS to DFS

In the above implementation of BFS, the fun part is that we could easily convert the BFS strategy to DFS by changing one single line of code. And the obtained DFS implementation is done in iteration, instead of recursion.

> The key is that instead of using the ***queue*** data structure which follows the principle of FIFO (First-In First-Out), if we use the ***stack*** data structure which follows the principle of LIFO (Last-In First-Out), we then switch the strategy from BFS to DFS.

Specifically, at the moment we pop an element from the queue, instead of popping out the *head* element, we pop the *tail* element, which then changes the behavior of the container from queue to stack. Here is how it looks like.

Java Python Copy

```python
    def DFS(self, board, row, col):
        from collections import deque
        queue = deque([(row, col)])
        while queue:
            # pop out the _tail_ element, rather than the head.
            (row, col) = queue.pop()
            if board[row][col] != 'O':
                continue
            # mark this cell as escaped
            board[row][col] = 'E'
            # check its neighbour cells
            if col < self.COLS-1: queue.append((row, col+1))
            if row < self.ROWS-1: queue.append((row+1, col))
            if col > 0: queue.append((row, col-1))
            if row > 0: queue.append((row-1, col))
```

Note that, though the above implementations indeed follow the DFS strategy, they are NOT equivalent to the previous ***recursive*** version of DFS, *i.e.* they do not produce the exactly same sequence of visit.

In the recursive DFS, we would visit the *right-hand side* neighbor `(row, col+1)` first, while in the iterative DFS, we would visit the *up* neighbor `(row-1, col)` first.

In order to obtain the same order of visit as the recursive DFS, one should *reverse* the processing order of neighbors in the above iterative DFS.

Complexity

- Time Complexity: $\mathcal{O}(N)$ where N is the number of cells in the board. In the worst case where it contains only the `O` cells on the board, we would traverse each cell twice: once during the BFS traversal