Cut Off Trees for Golf Event

You are asked to cut off trees in a forest for a golf event. The forest is represented as a non-negative 2D map, in this map:

1. `0` represents the `obstacle` can't be reached.
2. `1` represents the `ground` can be walked through.
3. `The place with number bigger than 1` represents a `tree` can be walked through, and this positive number represents the tree's height.

You are asked to cut off **all** the trees in this forest in the order of tree's height - always cut off the tree with lowest height first. And after cutting, the original place has the tree will become a grass (value 1).

You will start from the point (0, 0) and you should output the minimum steps **you need to walk** to cut off all the trees. If you can't cut off all the trees, output -1 in that situation.

You are guaranteed that no two `trees` have the same height and there is at least one tree needs to be cut off.

Example 1:

```
Input:
[
 [1,2,3],
 [0,0,4],
 [7,6,5]
]
Output: 6
```

Example 2:

```
Input:
[
 [1,2,3],
 [0,0,0],
 [7,6,5]
]
Output: -1
```

Example 3:

```
Input:
[
 [2,3,4],
 [0,0,5],
 [8,7,6]
]
Output: 6
Explanation: You started from the point (0,0) and you can cut off the tree in (0,0) d
irectly without walking.
```

Hint: size of the given matrix will not exceed 50x50.

Solution 1

1. Since we have to cut trees in order of their height, we first put trees (int[] {row, col, height}) into a priority queue and sort by height.
2. Poll each tree from the queue and use BFS to find out steps needed.

The worst case time complexity could be O(m^2 * n^2) (m = number of rows, n = number of columns) since there are m * n trees and for each BFS worst case time complexity is O(m * n) too.

```
class Solution {
    static int[][] dir = {{0,1}, {0, -1}, {1, 0}, {-1, 0}};

    public int cutOffTree(List<List<Integer>> forest) {
        if (forest == null || forest.size() == 0) return 0;
        int m = forest.size(), n = forest.get(0).size();

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

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (forest.get(i).get(j) > 1) {
                    pq.add(new int[] {i, j, forest.get(i).get(j)});
                }
            }
        }

        int[] start = new int[2];
        int sum = 0;
        while (!pq.isEmpty()) {
            int[] tree = pq.poll();
            int step = minStep(forest, start, tree, m, n);

            if (step < 0) return -1;
            sum += step;

            start[0] = tree[0];
            start[1] = tree[1];
        }

        return sum;
    }

    private int minStep(List<List<Integer>> forest, int[] start, int[] tree, int m,
int n) {
        int step = 0;
        boolean[][] visited = new boolean[m][n];
        Queue<int[]> queue = new LinkedList<>();
        queue.add(start);
        visited[start[0]][start[1]] = true;

        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                int[] curr = queue.poll();
```

```
                if (curr[0] == tree[0] && curr[1] == tree[1]) return step;

                for (int[] d : dir) {
                    int nr = curr[0] + d[0];
                    int nc = curr[1] + d[1];
                    if (nr < 0 || nr >= m || nc < 0 || nc >= n
                        || forest.get(nr).get(nc) == 0 || visited[nr][nc]) continue
;
                    queue.add(new int[] {nr, nc});
                    visited[nr][nc] = true;
                }
            }
            step++;
        }

        return -1;
    }
}
```

written by shawngao original link here

Solution 2

I thought it is a straightforward BFS search, so I write it like the following. Actually, I have the same question with `Number of Island` problem: https://discuss.leetcode.com/topic/88586/why-python-bfs-get-time-exceed-error

```
import collections
class Solution(object):
    def cutOffTree(self, G):
        """
        :type forest: List[List[int]]
        :rtype: int
        """
        if not G or not G[0]: return -1
        m, n = len(G), len(G[0])
        trees = []
        for i in xrange(m):
            for j in xrange(n):
                if G[i][j] > 1:
                    trees.append((G[i][j], i, j))
        trees = sorted(trees)
        count = 0
        cx, cy = 0, 0
        for h, x, y in trees:
            step = self.BFS(G, cx, cy, x, y)
            if step == -1:
                return -1
            else:
                count += step
                G[x][y] = 1
                cx, cy = x, y
        return count

    def BFS(self, G, cx, cy, tx, ty):
        m, n = len(G), len(G[0])
        visited = [[False for j in xrange(n)] for i in xrange(m)]
        Q = collections.deque()
        step = -1
        Q.append((cx, cy))
        while len(Q) > 0:
            size = len(Q)
            step += 1
            for i in xrange(size):
                x, y = Q.popleft()
                visited[x][y] = True
                if x == tx and y == ty:
                    return step
                for nx, ny in [(x + 1, y), (x - 1, y), (x, y-1), (x, y + 1)]:
                    if nx < 0 or nx >= m or ny < 0 or ny >= n or G[nx][ny] == 0 or
visited[nx][ny]:
                        continue
                    Q.append((nx, ny))
        return -1
```

written by Andrinux original link here

Solution 3

Just my own very similar implementation of @wufangjie's solution (and some terminology from the Hadlock algorithm which @awice mentioned to me), with some more explanation. Gets accepted in about 700 ms.

Basically find the trees, sort them by order, find the distance from each tree to the next, and sum those distances. But how to find the distance from one cell to some other cell? BFS is far to slow for the current test suite. Instead use what's apparently known as "Hadlock's Algorithm" (though I've only seen high-level descriptions of that). First try paths with no detour (only try steps in the direction towards the goal), then if necessary try paths with one detour step, then paths with two detour steps, etc. The distance then is the Manhattan distance plus twice the number of detour steps (twice because you'll have to make up for a detour step with a later step back towards the goal).

How to implement that?

- Round 1: Run a DFS only on cells that you can reach from the start cell with no detour towards the goal, i.e., only walking in the direction towards the goal. If this reaches the goal, we're done. Otherwise...
- Round 2: Try again, but this time try starting from all those cells reachable with one detour step. Collect these in round 1.
- Round 3: If round 2 fails, try again but start from all those cells reachable with two detour steps. Collect these in round 2.
- And so on...

If there are no obstacles, then this directly walks a shortest path towards the goal, which is of course very fast. Much better than BFS which would waste time looking in all directions. With only a few obstacles, it's still close to optimal.

My `distance` function does this searching algorithm. I keep the current to-be-searched cells in my `now` stack. When I move to a neighbor that's closer to the goal, I also put it in `now`. If it's not closer, then that's a detour step so I just remember it on my `soon` stack for the next round.

```
def cutOffTree(self, forest):

    # Add sentinels (a border of zeros) so we don't need index-checks later on.
    forest.append([0] * len(forest[0]))
    for row in forest:
        row.append(0)

    # Find the trees.
    trees = [(height, i, j)
             for i, row in enumerate(forest)
             for j, height in enumerate(row)
             if height > 1]

    # Can we reach every tree? If not, return -1 right away.
    queue = [(0, 0)]
    reached = set()
    for i, j in queue:
        if (i, j) not in reached and forest[i][j]:
            reached.add((i, j))
            queue += (i+1, j), (i-1, j), (i, j+1), (i, j-1)
    if not all((i, j) in reached for (_, i, j) in trees):
        return -1

    # Distance from (i, j) to (I, J).
    def distance(i, j, I, J):
        now, soon = [(i, j)], []
        expanded = set()
        manhattan = abs(i - I) + abs(j - J)
        detours = 0
        while True:
            if not now:
                now, soon = soon, []
                detours += 1
            i, j = now.pop()
            if (i, j) == (I, J):
                return manhattan + 2 * detours
            if (i, j) not in expanded:
                expanded.add((i, j))
                for i, j, closer in (i+1, j, i < I), (i-1, j, i > I), (i, j+1, j <
J), (i, j-1, j > J):
                    if forest[i][j]:
                        (now if closer else soon).append((i, j))

    # Sum the distances from one tree to the next (sorted by height).
    trees.sort()
    return sum(distance(i, j, I, J) for (_, i, j), (_, I, J) in zip([(0, 0, 0)] + tr
ees, trees))
```

written by StefanPochmann original link here