BFS, DFS, and Flood Fill

Daniel Liu

December 3, 2018

1 BFS and DFS

DFS stands for depth-first search, and BFS stands for breadth-first search. As their name suggests, they are different searching algorithms that solve problems by searching for the answer instead of just directly calculating it. This may sound similar to brute-forcing answers using recursion. Indeed, BFS and DFS can be used to, for example, print out every combination of 3 digit numbers that do not contain all 3s, which sounds like a recursive problem.

We will only examine a subset of these searching problems that involves traversing a rectangular grid of square cells. In general, problems consisting of grids (or graphs) tend to require extensive use of BFS and DFS.

A very natural query on a grid is finding a path between a starting cell and an ending cell by only moving up, down, left, or right from one cell to the next (eg. Manhattan/taxicab distance). For an empty grid, there are many such paths, and it is fairly simple to come up with one. As such, we will increase the difficulty with a few cells that are blocked. Blocked cells cannot be accessed in our path from the starting cell to the ending cell.

This type of problem maps very nicely onto some sort of recursive function—our state is uniquely determined by the row and column of our current cell, and we can reach another state (cell) by going up, down, left, or right. This leads directly to the DFS algorithm. Note that we have to be careful to stop a recursive call early if the current cell is blocked or if we already visited that cell. Returning early is extremely important to make sure that the overall run time is $O(n)$, as each of the n cells is only visited once.

Algorithm 1 DFS on grid

$start \leftarrow$ starting row and column
$end \leftarrow$ ending row and column
$g_{r,c} \leftarrow 1$ if $g_{r,c}$ is blocked, else 0
$v_{r,c} \leftarrow 0$
$res \leftarrow ?$
function DFS$(r, c, path)$
    if $((r, c)$ out of bounds) or $(g_{r,c} = 1)$ or $(v_{r,c} = 1)$ **then**
        return
    end if
    $v_{r,c} \leftarrow 1$
    if $(r, c) = end$ **then**
        $res \leftarrow path$
        return
    end if
    for $(a, b) \in \{(0, 1), (0, -1), (1, 0), (-1, 0)\}$ **do**
        DFS$(r + a, c + b, path + [(r + a, c + b)])$
    end for
end function
DFS$(start_r, start_c, [start])$
res contains a path from $start$ to end

Note that DFS is not guaranteed to find the shortest or the longest path from start to end. It tries to go as far as possible in one direction until it gets stuck. Then it will backtrack and attempt other paths.

BFS uses an iterative method to go through empty cells in the grid. Each iteration, it expands a queue by choosing a cell in the queue and adding its neighbors that are not in the queue to the queue. In contrast to DFS, BFS handles cells in a "first come first serve" manner, as the queue returns earlier cells before later cells. This allows the first valid path found by BFS to go from start to end to be one of the shortest paths possible, if path is measured by Manhattan distance. Since each cell is only added to the queue and removed from the queue once, the worse case run time complexity is the same as DFS: $O(n)$.

Algorithm 2 BFS on grid

```
1: function BFS(g, start, end)
2:     q ← []
3:     vis_{r,c} ← 0
4:     p_{r,c} ← ?
5:     add start to q
6:     vis_{start} ← 1
7:     while q is not empty do
8:         u ← next in q
9:         if u = end then
10:            break out of while loop
11:        end if
12:        for v ∈ {(0,1),(0,−1),(1,0),(−1,0)} do
13:            if (v not out of bounds) and (vis_v = 0) and (g_v = 0) then
14:                vis_v ← 1
15:                p_v = u
16:                add v to q
17:            end if
18:        end for
19:    end while
20:    return the path by following the values in p, starting from p_{end}, and
       ending at p_{start}
21: end function
```

In the DFS example, the path is tracked through a list of cells visited in order. However, for BFS, we keep track of the previous cell for each current cell, and that allows us to get back trace the path in reverse at the end.

2 Flood Fill

One very useful application of DFS and BFS is finding connected "islands". For example, a certain area of the grid might be completely cut off from the outside by blocked cells. To find out which cells inside or outside the blocked region, we can run either DFS or BFS starting from a certain cell, and checking which cells are visited after the algorithm terminates.

What if there are multiple blocked regions? Then we can assign a "color" (a number would be more practical) to each cell. The assignment of colors must satisfy the following property: if it is possible to travel from one cell to another, then those two cells must have the same color. To figure out the colors for each cell, we can run flood fill starting from every single cell and skipping the ones that are already colored. Each time we color in a new region, we can use a different color to distinguish them. By precomputing the colors using flood fill, we can easily figure out whether two cells are connected by some path in $O(1)$ time.