LeetCode Training Day 23 BFS

BFS is a typical graph search pattern which is discussed in chapter 7. Here we just spend more time to look at some typical pattern.

BFS pattern is that we put our starting point into queue, set the visited flag for the starting point then we run the queue until empty. For every round we take a node from queue and search its neighbors, inserted unvisited node to queue and mark them visited. When the queue is empty, our search complete.

Please remember that for each round, it means our steps increase by 1, so if we care about how many steps we reach to the destination, we should get the size of the current queue and loop that each node, on each round we will insert nodes in the next round in the queue.

The search in graph is same as what we discussed in chapter 7, if we do it in grid, we run 4 directions; if we do connected graph, we should convert edges to neighbors.

## 1765. Map of Highest Peak

Medium

You are given an integer matrix isWater of size m x n that represents a map of **land** and **water** cells.

* If isWater[i][j] == 0, cell (i, j) is a **land** cell.
* If isWater[i][j] == 1, cell (i, j) is a **water** cell.

You must assign each cell a height in a way that follows these rules:

* The height of each cell must be non-negative.
* If the cell is a **water** cell, its height must be 0.
* Any two adjacent cells must have an absolute height difference of **at most** 1. A cell is adjacent to another cell if the former is directly north, east, south, or west of the latter (i.e., their sides are touching).

Find an assignment of heights such that the maximum height in the matrix is **maximized**.

Return *an integer matrix*height*of size*m x n*where*height[i][j]*is cell*(i, j)*'s height. If there are multiple solutions, return****any****of them*.

**Example 1:**

**A picture containing text, clock

Description automatically generated**

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

**Output:** [[1,0],[2,1]]

**Explanation:** The image shows the assigned heights of each cell.

The blue cell is the water cell, and the green cells are the land cells.

**Example 2:**

**Table

Description automatically generated with medium confidence**

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

**Output:** [[1,1,0],[0,1,1],[1,2,2]]

**Explanation:** A height of 2 is the maximum possible height of any assignment.

Any height assignment that has a maximum height of 2 while still meeting the rules will also be accepted.

**Constraints:**

* m == isWater.length
* n == isWater[i].length
* 1 <= m, n <= 1000
* isWater[i][j] is 0 or 1.
* There is at least **one** water cell.

### Analysis:

This is a google problem. We start from water and use BFS to find its neighbor land then get more neighbors. If the land heights not marked, we will mark it as neighbor height + 1. Eventually we will mark all nodes. Because in this problem we set the original height as an impossible value INT\_MAX (you can do -1 as well), then we do not need the visited matrix to remember which positions are visited, i.e. any position without specific height are unvisited.

/// <summary>

/// Leet code 1765. Map of Highest Peak

///

/// Medium

///

/// You are given an integer matrix isWater of size m x n that represents

/// a map of land and water cells.

///

/// If isWater[i][j] == 0, cell (i, j) is a land cell.

/// If isWater[i][j] == 1, cell (i, j) is a water cell.

/// You must assign each cell a height in a way that follows these rules:

///

/// The height of each cell must be non-negative.

/// If the cell is a water cell, its height must be 0.

/// Any two adjacent cells must have an absolute height difference of at

/// most 1. A cell is adjacent to another cell if the former is directly

/// north, east, south, or west of the latter (i.e., their sides are

/// touching).

/// Find an assignment of heights such that the maximum height in the

/// matrix is maximized.

///

/// Return an integer matrix height of size m x n where height[i][j] is

/// cell (i, j)'s height. If there are multiple solutions, return any of

/// them.

///

/// Example 1:

/// Input: isWater = [[0,1],[0,0]]

/// Output: [[1,0],[2,1]]

/// Explanation: The image shows the assigned heights of each cell.

/// The blue cell is the water cell, and the green cells are the land

/// cells.

///

/// Example 2:

/// Input: isWater = [[0,0,1],[1,0,0],[0,0,0]]

/// Output: [[1,1,0],[0,1,1],[1,2,2]]

/// Explanation: A height of 2 is the maximum possible height of any

/// assignment.

/// Any height assignment that has a maximum height of 2 while still meeting

/// the rules will also be accepted.

///

/// Constraints:

/// 1. m == isWater.length

/// 2. n == isWater[i].length

/// 3. 1 <= m, n <= 1000

/// 4. isWater[i][j] is 0 or 1.

/// 5. There is at least one water cell.

/// </summary>

vector<vector<int>> LeetCodeGraph::highestPeak(vector<vector<int>>& isWater)

{

int n = isWater.size();

int m = isWater[0].size();

vector<vector<int>> result(n, vector<int>(m, INT\_MAX));

queue<pair<int, int>> queue;

for (int i = 0; i < n; i++)

{

for (int j = 0; j < m; j++)

{

if (isWater[i][j] == 1)

{

result[i][j] = 0;

queue.push(make\_pair(i, j));

}

}

}

vector<vector<int>> directions = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

while (!queue.empty())

{

pair<int, int> pos = queue.front();

queue.pop();

for (size\_t i = 0; i < directions.size(); i++)

{

pair<int, int> next = pos;

next.first += directions[i][0];

next.second += directions[i][1];

if (next.first < 0 || next.first >= n || next.second < 0 || next.second >= m)

{

continue;

}

if (result[next.first][next.second] != INT\_MAX) continue;

result[next.first][next.second] = result[pos.first][pos.second] + 1;

queue.push(next);

}

}

return result;

}

## 582. Kill Process

Medium

You have n processes forming a rooted tree structure. You are given two integer arrays pid and ppid, where pid[i] is the ID of the ith process and ppid[i] is the ID of the ith process's parent process.

Each process has only **one parent process** but may have multiple children processes. Only one process has ppid[i] = 0, which means this process has **no parent process** (the root of the tree).

When a process is **killed**, all of its children processes will also be killed.

Given an integer kill representing the ID of a process you want to kill, return *a list of the IDs of the processes that will be killed. You may return the answer in****any order****.*

**Example 1:**

A picture containing text, clipart

Description automatically generated

**Input:** pid = [1,3,10,5], ppid = [3,0,5,3], kill = 5

**Output:** [5,10]

**Explanation:** The processes colored in red are the processes that should be killed.

**Example 2:**

**Input:** pid = [1], ppid = [0], kill = 1

**Output:** [1]

**Constraints:**

* n == pid.length
* n == ppid.length
* 1 <= n <= 5 \* 104
* 1 <= pid[i] <= 5 \* 104
* 0 <= ppid[i] <= 5 \* 104
* Only one process has no parent.
* All the values of pid are **unique**.
* kill is **guaranteed** to be in pid.

### Analysis:

This is a typical problem. First, we need to convert the parent relationship to children relationship, from the starting process, then its children, then children’s children. This is a simplified version of topology sort, you just do not need to calculate ingress degree.

/// <summary>

/// Leet code #582. Kill Process

///

/// Given n processes, each process has a unique PID (process id) and its

/// PPID (parent process id).

/// Each process only has one parent process, but may have one or more

/// children processes. This is just like a tree structure. Only one

/// process has PPID that is 0, which means this process has no parent

/// process. All the PIDs will be distinct positive integers.

///

/// We use two list of integers to represent a list of processes, where

/// the first list contains PID for each process and the second list

/// contains the corresponding PPID.

///

/// Now given the two lists, and a PID representing a process you want

/// to kill, return a list of PIDs of processes that will be killed in

/// the end. You should assume that when a process is killed,

/// all its children processes will be killed.

/// No order is required for the final answer.

/// Example 1:

/// Input:

/// pid = [1, 3, 10, 5]

/// ppid = [3, 0, 5, 3]

/// kill = 5

/// Output: [5,10]

/// Explanation:

/// 3

/// / \

/// 1 5

/// /

/// 10

/// Kill 5 will also kill 10.

/// Note:

/// 1. The given kill id is guaranteed to be one of the given PIDs.

/// 2. n >= 1.

/// </summary>

vector<int> LeetCodeGraph::killProcess(vector<int>& pid, vector<int>& ppid, int kill)

{

unordered\_map<int, set<int>> process\_map;

queue<int> process\_queue;

vector<int> result;

for (size\_t i = 0; i < pid.size(); i++)

{

process\_map[ppid[i]].insert(pid[i]);

}

process\_queue.push(kill);

while (!process\_queue.empty())

{

int process\_id = process\_queue.front();

process\_queue.pop();

result.push\_back(process\_id);

for (int p : process\_map[process\_id])

{

process\_queue.push(p);

}

}

return result;

}

## 994. Rotting Oranges

Medium

You are given an m x n grid where each cell can have one of three values:

* 0 representing an empty cell,
* 1 representing a fresh orange, or
* 2 representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return *the minimum number of minutes that must elapse until no cell has a fresh orange*. If *this is impossible, return* -1.

**Example 1:**

A picture containing scatter chart

Description automatically generated

**Input:** grid = [[2,1,1],[1,1,0],[0,1,1]]

**Output:** 4

**Example 2:**

**Input:** grid = [[2,1,1],[0,1,1],[1,0,1]]

**Output:** -1

**Explanation:** The orange in the bottom left corner (row 2, column 0) is never rotten, because rotting only happens 4-directionally.

**Example 3:**

**Input:** grid = [[0,2]]

**Output:** 0

**Explanation:** Since there are already no fresh oranges at minute 0, the answer is just 0.

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 10
* grid[i][j] is 0, 1, or 2.

### Analysis:

Starting from rotten oranges, walk 4 directions to get more oranges to rotten. At very beginning push all the rotten oranges to queue and count all fresh oranges.

/// <summary>

/// Leet code #994. Rotting Oranges

///

/// In a given grid, each cell can have one of three values:

///

/// the value 0 representing an empty cell;

/// the value 1 representing a fresh orange;

/// the value 2 representing a rotten orange.

/// Every minute, any fresh orange that is adjacent (4-directionally) to a

/// rotten orange becomes rotten.

///

/// Return the minimum number of minutes that must elapse until no cell has

/// a fresh orange. If this is impossible, return -1 instead.

///

/// Example 1:

///

/// Input: [[2,1,1],[1,1,0],[0,1,1]]

/// Output: 4

///

/// Example 2:

///

/// Input: [[2,1,1],[0,1,1],[1,0,1]]

/// Output: -1

/// Explanation: The orange in the bottom left corner (row 2, column 0)

/// is never rotten, because rotting only happens 4-directionally.

///

/// Example 3:

///

/// Input: [[0,2]]

/// Output: 0

/// Explanation: Since there are already no fresh oranges at minute 0, the

/// answer is just 0.

///

///

/// Note:

///

/// 1. 1 <= grid.length <= 10

/// 2. 1 <= grid[0].length <= 10

/// 3. grid[i][j] is only 0, 1, or 2.

/// </summary>

int LeetCodeBFS::orangesRotting(vector<vector<int>>& grid)

{

queue<pair<int, int>> search;

int count = 0;

for (size\_t i = 0; i < grid.size(); i++)

{

for (size\_t j = 0; j < grid[i].size(); j++)

{

if (grid[i][j] == 2)

{

search.push(make\_pair(i, j));

}

else if (grid[i][j] == 1)

{

count++;

}

}

}

int result = 0;

vector<vector<int>> directions = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

while (!search.empty())

{

size\_t size = search.size();

for (size\_t i = 0; i < size; i++)

{

pair<int, int> pos = search.front();

search.pop();

for (size\_t d = 0; d < directions.size(); d++)

{

pair<int, int> next\_pos = pos;

next\_pos.first += directions[d][0];

next\_pos.second += directions[d][1];

if (next\_pos.first < 0 || next\_pos.first >= (int)grid.size() ||

next\_pos.second < 0 || next\_pos.second >= (int)grid[0].size())

{

continue;

}

if (grid[next\_pos.first][next\_pos.second] == 1)

{

count--;

search.push(make\_pair(next\_pos.first, next\_pos.second));

grid[next\_pos.first][next\_pos.second] = 2;

}

}

}

if (!search.empty()) result++;

}

if (count != 0) result = -1;

return result;

}

## 286. Walls and Gates

Medium

You are given an m x n grid rooms initialized with these three possible values.

* -1 A wall or an obstacle.
* 0 A gate.
* INF Infinity means an empty room. We use the value 231 - 1 = 2147483647 to represent INF as you may assume that the distance to a gate is less than 2147483647.

Fill each empty room with the distance to *its nearest gate*. If it is impossible to reach a gate, it should be filled with INF.

**Example 1:**

A picture containing text, crossword puzzle

Description automatically generated

**Input:** rooms = [[2147483647,-1,0,2147483647],[2147483647,2147483647,2147483647,-1],[2147483647,-1,2147483647,-1],[0,-1,2147483647,2147483647]]

**Output:** [[3,-1,0,1],[2,2,1,-1],[1,-1,2,-1],[0,-1,3,4]]

**Example 2:**

**Input:** rooms = [[-1]]

**Output:** [[-1]]

**Constraints:**

* m == rooms.length
* n == rooms[i].length
* 1 <= m, n <= 250
* rooms[i][j] is -1, 0, or 231 - 1.

### Analysis:

Starting from gate and mark every empty room distance.

/// <summary>

/// Leet Code 286. Walls and Gates

///

/// Medium

///

/// You are given an m x n grid rooms initialized with these three

/// possible values.

///

/// -1 A wall or an obstacle.

/// 0 A gate.

/// INF Infinity means an empty room. We use the value

/// 2^31 - 1 = 2147483647 to represent INF as you may assume that the

/// distance to a gate is less than 2147483647.

/// Fill each empty room with the distance to its nearest gate. If it

/// is impossible to reach a gate, it should be filled with INF.

///

/// Example 1:

/// Input: rooms = [[2147483647,-1,0,2147483647],

/// [2147483647,2147483647,2147483647,-1], [2147483647,-1,2147483647,-1],

/// [0,-1,2147483647,2147483647]]

/// Output: [[3,-1,0,1],[2,2,1,-1],[1,-1,2,-1],[0,-1,3,4]]

///

/// Example 2:

/// Input: rooms = [[-1]]

/// Output: [[-1]]

///

/// Constraints:

/// 1. m == rooms.length

/// 2. n == rooms[i].length

/// 3. 1 <= m, n <= 250

/// 4. rooms[i][j] is -1, 0, or 2^31 - 1.

/// </summary>

void LeetCodeGraph::wallsAndGates(vector<vector<int>>& rooms)

{

int n = rooms.size();

int m = rooms[0].size();

queue<vector<int>> process\_queue;

for (int i = 0; i < (int)rooms.size(); i++)

{

for (int j = 0; j < (int)rooms[i].size(); j++)

{

if (rooms[i][j] == 0)

{

process\_queue.push({ i, j });

}

}

}

vector<vector<int>> directions = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

while (!process\_queue.empty())

{

vector<int> pos = process\_queue.front();

process\_queue.pop();

for (size\_t d = 0; d < directions.size(); d++)

{

vector<int> next = pos;

next[0] += directions[d][0];

next[1] += directions[d][1];

if (next[0] < 0 || next[0] >= n || next[1] < 0 || next[1] >= m)

{

continue;

}

if (rooms[next[0]][next[1]] == -1 || rooms[next[0]][next[1]] != INT\_MAX)

{

continue;

}

rooms[next[0]][next[1]] = rooms[pos[0]][pos[1]] + 1;

process\_queue.push(next);

}

}

}

### Analysis:

For this problem you can also do DFS, just make sure check the gate position.

/// <summary>

/// Leet code #286. Walls and Gates

/// </summary>

void LeetCodeGraph::wallsAndGatesDFS(vector<vector<int>>& rooms, int row, int col, int distance)

{

if (row < 0 || col < 0 || row >= (int)rooms.size() || col >= (int)rooms[0].size())

{

return;

}

if (rooms[row][col] <= distance && distance != 0)

{

return;

}

rooms[row][col] = distance;

wallsAndGatesDFS(rooms, row - 1, col, distance + 1);

wallsAndGatesDFS(rooms, row, col - 1, distance + 1);

wallsAndGatesDFS(rooms, row + 1, col, distance + 1);

wallsAndGatesDFS(rooms, row, col + 1, distance + 1);

}

/// <summary>

/// Leet code #286. Walls and Gates

///

/// You are given a m x n 2D grid initialized with these three possible values.

/// 1.-1 - A wall or an obstacle.

/// 2.0 - A gate.

/// 3.INF - Infinity means an empty room. We use the value 231 - 1 = 2147483647 to represent

/// INF as you may assume that the distance to a gate is less than 2147483647.

/// Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, it should be filled with INF.

/// For example, given the 2D grid:

/// INF -1 0 INF

/// INF INF INF -1

/// INF -1 INF -1

/// 0 -1 INF INF

/// After running your function, the 2D grid should be:

/// 3 -1 0 1

/// 2 2 1 -1

/// 1 -1 2 -1

/// 0 -1 3 4

/// </summary>

void LeetCodeGraph::wallsAndGatesDFS(vector<vector<int>>& rooms)

{

for (size\_t i = 0; i < rooms.size(); i++)

{

for (size\_t j = 0; j < rooms[0].size(); j++)

{

if (rooms[i][j] == 0)

{

wallsAndGatesDFS(rooms, i, j, 0);

}

}

}

}