Day 3

Done	☑
Topic	994. Rotting Oranges - Leetcode Prim's Algorithm
Languages	Java
Difficulty	⭐⭐⭐⭐⭐
Date Started	@March 3, 2023 11:50 PM
Date Completed	@March 4, 2023 4:49 AM
Related to Progress (Days)	Your Progress

What I Learned Today

Key Points of Minimum Spanning Tree(MST)

Prim’s Algorithm with Implementation

994. Rotting Oranges(Hard) - LeetCode

Key Concepts

MINIMUM SPANNING TREE

A minimum spanning tree is a tree that spans all the vertices of a weighted, undirected graph, while minimizing the total weight of the edges. Here are some of its key points:

1. Subset of the edges of the graph that connects all vertices with the minimum total weight.

Quick Links

Prim's Algorithm

Documentation

Rotting Oranges

Tutorial

2. Can be found using algorithms such as Kruskal's Algorithm or Prim's Algorithm.
3. Can be used to find the shortest path between any two vertices in the graph.
4. If Vertices is “X” then Edges must be “X-1”.

PRIM’S ALGORITHM

Prim's Algorithm is a greedy algorithm that can be used to find the minimum spanning tree of a weighted, undirected graph. Here are some of its key points:

1. Start with a single vertex and add it to the minimum spanning tree.
2. Find the edge with the smallest weight that connects a vertex in the tree to a vertex not in the tree.
3. Add the new vertex to the tree, along with the edge that connects it to the tree.
4. Repeat steps 2 and 3 until all vertices are in the tree.
5. The resulting tree will be the minimum spanning tree of the graph, which is the tree that connects all vertices with the smallest total weight.
6. Prim's Algorithm is guaranteed to find the correct minimum spanning tree, but it may not be the only one possible.
7. The time complexity of Prim's Algorithm is O(E log V), where E is the number of edges and V is the number of vertices in the graph.

994. Rotting Oranges(Hard) - LeetCode

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.

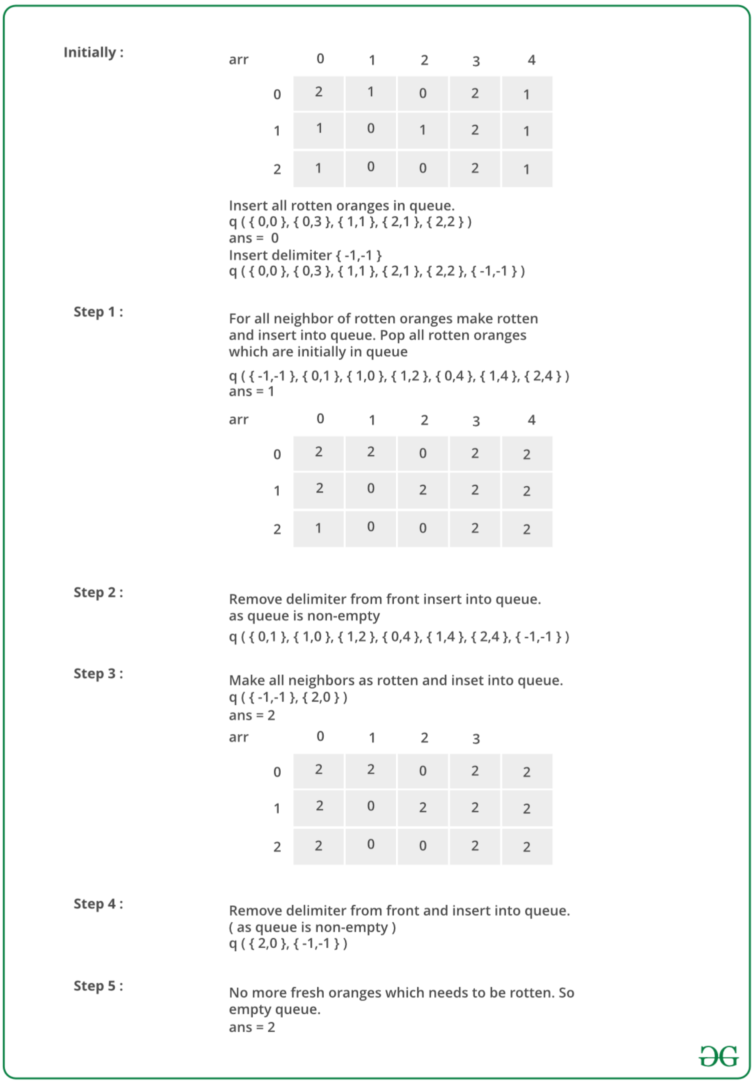
Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

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

```
    Input: grid = [[2,1,1],
                   [1,1,0],
                   [0,1,1]]
Output: 4
```

```
Input: grid = [[2,1,1],
               [0,1,1],
               [1,0,1]]
Output: -1
Explanation: The orange in the bottom left corner (row 2, col
umn 0) is never rotten, because rotting only happens 4-direct
ionally.
```

- Create an empty queue **Q**.
- Find all rotten oranges and enqueue them to **Q**. Also, enqueue a delimiter to indicate the beginning of the next time frame.
- Run a loop While Q is not empty and do the following while the delimiter in **Q** is not reached
    - Dequeue an orange from the queue, and rot all adjacent oranges.
    - While rotting the adjacent, make sure that the time frame is incremented only once. And the time frame is not incremented if there are no adjacent oranges.
    - Dequeue the old delimiter and enqueue a new delimiter. The oranges rotten in the previous time frame lie between the two delimiters.
- Return the last time frame.



Code Snippets

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.PriorityQueue;

public class PrimsAlgorithm {
    static class Edge {
        int src;
        int dest;
        int wt;

        public Edge(int s, int d, int w) {
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }
        static void createGraph(ArrayList<Edge> graph[]){
```

```java
        for (int i=0; i< graph.length; i++){
            graph[i]=new ArrayList<>();
        }
        graph[0].add(new Edge(0,1,10));
        graph[0].add(new Edge(0,2,15));
        graph[0].add(new Edge(0,3,30));

        graph[1].add(new Edge(1,0,10));
        graph[1].add(new Edge(1,2,40));


        graph[2].add(new Edge(2,3,50));
        graph[2].add(new Edge(2,0,15));

        graph[3].add(new Edge(3,0,30));
        graph[3].add(new Edge(3,1,40));

    }
    static class Pair implements Comparable<Pair>{
        int vertex;
        int cost;
        public Pair(int vertex, int cost) {
            this.vertex = vertex;
            this.cost = cost;
        }
        @Override
        public int compareTo(Pair p2){
            return this.cost-p2.cost;
        }
    }
    public static int primsAlgorithm(ArrayList<Edge> []graph){
        boolean []visited = new boolean[graph.length];
        Arrays.fill(visited,false);
        PriorityQueue<Pair> pq = new PriorityQueue<Pair>();
        pq.add(new Pair(0,0));
        int finalCost = 0;
        while (!pq.isEmpty()){
            Pair current = pq.remove();
            if (!visited[current.vertex]){
                visited[current.vertex]=true;
                finalCost+=current.cost;
                for (int i=0; i<graph[current.vertex].size(); i++){
                        Edge e = graph[current.vertex].get(i);
                        pq.add(new Pair(e.dest,e.wt));

                }
            }
        }
        return finalCost;
    }

    public static void main(String[] args) {
        int V = 4;
        ArrayList<Edge> []graph = new ArrayList[V];
        createGraph(graph);
        System.out.println(primsAlgorithm(graph));
    }
}
```

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.Queue;

public class GraphPractice {
// structure for storing coordinates of the cell
    static class Cordinates {
        int i = 0;
        int j = 0;

        public Cordinates(int i, int j) {
            this.i = i;
            this.j = j;
        }
    }
    // Function to check whether the cell is delimiter
    // which is (-1, -1)
    static boolean isDelimiter(Cordinates temp) {
        return (temp.i == -1 && temp.j == -1);
    }
    // function to check whether a cell is valid / invalid
    static boolean isValid(int i, int j) {
        return (i >= 0 && j >= 0 && i < R && j < C);
    }

    public final static int R = 3;
    public final static int C = 5;
    // Function to check whether there is still a fresh
    // orange remaining
    static boolean checkAll(int arr[][]) {
        for (int i = 0; i < R; i++)
            for (int j = 0; j < C; j++)
                if (arr[i][j] == 1)
                    return true;
        return false;
    }
    // This function finds if it is possible to rot all
    // oranges or not. If possible, then it returns minimum
    // time required to rot all, otherwise returns -1
    public static int orangesRotting(int[][] grid) {
        // Create a queue of cells
        Queue<Cordinates> queue = new LinkedList<>();
        Cordinates curr;
        int ans = 0;
        // Store all the cells having rotten orange in first
        // time frame
        for (int i = 0; i < R; i++) {
            for (int j = 0; j < C; j++) {
                if (grid[i][j] == 2) {
                    queue.add(new Cordinates(i, j));
                }
            }
        }
```

```
        // Separate these rotten oranges from the oranges
        // which will rotten due the oranges in first time
        // frame using delimiter which is (-1, -1)
        queue.add(new Cordinates(-1, -1));
        while (!queue.isEmpty()) {
            // This flag is used to determine whether even a
            // single fresh orange gets rotten due to rotten
            // oranges in the current time frame so we can
            // increase the count of the required time.
            boolean flag = false;
            // Process all the rotten oranges in current
            // time frame.
            while (!isDelimiter(queue.peek())) {
                curr = queue.peek();
                // Check right adjacent cell that if it can
                // be rotten
                if (isValid(curr.i + 1, curr.j) && grid[curr.i + 1][curr.j] == 1) {
                    // if this is the first orange to
                        // get rotten, increase count and
                        // set the flag.
                    if (!flag) {
                        ans++;
                        flag = true;
                    }
                    // Make the orange rotten
                    grid[curr.i + 1][curr.j] = 2;
                    // push the adjacent orange to Queue
                    curr.i++;
                    queue.add(new Cordinates(curr.i, curr.j));
                    // Move back to current cell
                    curr.i--;
                }
                //Left Adjacent Cell
                if (isValid(curr.i - 1, curr.j) && grid[curr.i - 1][curr.j] == 1) {
                    if (!flag) {
                        ans++;
                        flag = true;
                    }
                    grid[curr.i - 1][curr.j] = 2;
                    curr.i--;
                    queue.add(new Cordinates(curr.i, curr.j));
                    curr.i++;
                }
                //Top Adjacent Cell
                if (isValid(curr.i, curr.j + 1) && grid[curr.i][curr.j + 1] == 1) {
                    if (!flag) {
                        ans++;
                        flag = true;
                    }
                    grid[curr.i][curr.j + 1] = 2;
                    curr.j++;
                    queue.add(new Cordinates(curr.i, curr.j));
                    curr.j--;
                }
//                Bottom Adjacent Cell
                if (isValid(curr.i, curr.j - 1) && grid[curr.i][curr.j - 1] == 1) {
                    if (!flag) {
                        ans++;
```

```
                        flag = true;
                    }
                    grid[curr.i][curr.j - 1] = 2;
                    curr.j--;
                    queue.add(new Cordinates(curr.i, curr.j));
                }
                queue.remove();
            }
            // Pop the delimiter
            queue.remove();
            // If oranges were rotten in current frame than
            // separate the rotten oranges using delimiter
            // for the next frame for processing.
            if (!queue.isEmpty()) {
                queue.add(new Cordinates(-1, -1));
            }
            // If Queue was empty than no rotten oranges
            // left to process so exit
        }
        // Return -1 if all arranges could not rot,
        // otherwise ans
        return (checkAll(grid)) ? -1 : ans;
    }
public static void main(String[] args) {
int arr[][] = {{2, 1, 0, 2, 1},
               {1, 0, 1, 2, 1},
               {1, 0, 0, 2, 1}};
        System.out.println(orangesRotting(arr));
    }
}
```

Challenges Experienced

Got better understanding of Queue by taking Objects in Queue.

Comparable topic still confusion for me.

Resources Used

Alpha, YouTube , ChatGPT, GeeksForGeeks