Graph

Mustafa Muhammad

5th October 2021

1. Graph basics + Representation
2. Breadth First Search
3. Depth First Search
4. Detect cycle in undirected graph
5. Topological Sort
6. Dijkstra's Algorithm
7. Bellman Ford ALgorithm
8. Floyd Warshall's Alrogrithm
9. Kosaraju's Algorithms
10. Tarjan's Algorithm
11. Prim's Algorithm
12. Kruskal's Algorithm
13. Mother Vertex in graph
14. Flood Fill algorithm
15. Count source to destination paths in a graph
16. Rotten oranges
17. Steps by knight
18. Bipartite graph
19. Hamiltonian path
20. Travelling Salesman Problem
21. Graph coloring problem
22. Alien Dictionary
23. Clone a graph
24. Articulation point in a graph
25. Bridges in a graph

1 Graph Algorithms

Graph = Set of vertices and edges

Undirected Edge = Path between two vertices with no direction

Directed Edge = Path betwen two vertices with direction

Complete graph = Every vertex has an edge to every other vertex

Indegree = How many edges go into the vertex

Outdegree = How many edges go out of a particular vertex

Bridge = An edge if removed can divide the graph into two parts

Aritculation point = Vertex if removed divides the graph into two parts

Spanning Tree = Graph with no cycles, V vertes and Edges (V-1)

2 Representation of graphs

Adjacency Matrix = Matrix of VxV

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

In case of weighted matrix, we replace 1 with weight.

Construct = $O(V^2)$ Neighbors = $O(V)$ Space = $O(V^2)$

Adjacency List

0: [] 1: [0, 2, 3] 2: [0, 3] 3: [0]

3 BFS

```
class Solution:
    def bfsOfGraph(self, V, adj):
        res = []
        queue = []
        queue.append(0)
        visited = set()
        visited.add(V)
        while(len(queue) != 0):
            pop = queue.pop(0)
            res.append(pop)
            for node in adj[pop]:
                if node not in visited:
                    queue.append(node)
                    visited.add(node)
        return res
```

4 DFS

```
class Solution:
    def dfsOfGraph(self, V, adj):
        visited = set()
        res = []
        def dfs(node, adj):
            res.append(node)
            visited.add(node)
            for child in adj[node]:
                if child not in visited:
                    dfs(child, adj)
            return
        dfs(0, adj)
        return res
```

Time complexity of both approaches:

1, DFS TimeComplexity = O(V+E) Space = O(V+E) including stack space.

2, BFS TimeComplexity = O(V+E) Space = O(V+E)

Cycle is defined if we have a back edge in our graph

5 Detect cycle in undirected graph

```
class Solution:
        def isCycle(self, V, adj):
            visited = set()
            def dfs_for_cycle(node, parent, adj):
                visited.add(node)
                for curr in adj[node]:
                    if curr not in visited:
                        if dfs_for_cycle(curr, node, adj):
                            return True
                    elif curr != parent:
                        return True
                return False
            return_bool = False
            for i in range(0, V):
                if i not in visited:
                    return_bool = return_bool or dfs_for_cycle(i,-1,adj)
            return return_bool
```

6 Detect cycle in a directed graph

The are two parts to this question. A graph contains a cycle if it visits a node that it has previously visited. Therefore we keep track of two sets, one telling us the nodes that we have currently visited and the second one telling us the ancestors of that node.

We return true if our node is found among the ancestors, otherwise we just traverse the subgraph using DFS.

```
class Solution:
    def isCyclic(self, V, adj):

        visited = set()
        ancestor = set()

        def dfs(node, adj):
            visited.add(node)
            ancestor.add(node)

            for child in adj[node]:
                if child not in visited:
                    if dfs(child, adj):
                        return True
                elif child in ancestor:
                    return True

            ancestor.remove(node)
            return False

        res = False
        for i in range(0, V):
            if i not in visited:
                res = res or dfs(i, adj)
        return res
```

7 Topological Sort

Simple DFS but add node into stack before returning.

```
class Solution:

    def topoSort(self, V, adj):
        visited = set()
        res = []
        def top_dfs(node, adj):
            if node in visited:
                return

            visited.add(node)

            for child in adj[node]:
                top_dfs(child, adj)
            res.append(node)
            return

        for i in range(0, V):
            if i not in visited:
                top_dfs(i, adj)

        res.reverse()
        return res
```

8 Dijkstra's Algorithm

It is a variation on BFS, as we are using a priority queue instead of a normal queue and searching through frontiers. The total time complexity is O(E(Log(V))).

You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k. Return the time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1

```
import heapq
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        #Step 1 create an adjacency list
        adjacency_map = {}
        for arr in times:
            start_node = arr[0]
            end_node = arr[1]
            weight = arr[2]
            if start_node in adjacency_map:
                adjacency_map[start_node].append((weight, end_node))
            else:
                adjacency_map[start_node] = [(weight, end_node)]
        #initialize the min_heap
        min_heap = [(0, k)]
        t = -1
        visited = set()
        #initialize a visited set to avoid going into a cycle
        while len(min_heap) != 0:
            w1, node1 = heapq.heappop(min_heap)
            if node1 in visited:
                continue
            visited.add(node1)
            t = max(t, w1)
            if node1 in adjacency_map:
                for w2, node2 in adjacency_map[node1]:
                    if node2 not in visited:
                        heapq.heappush(min_heap, (w2+w1, node2))
        return t if len(visited) == n else -1
```

9 Dijkstra's returning an array of distances

```
import heapq
class Solution:
    def dijkstra(self, V, adj, S):
        priority_queue = [(0, S)]
        visited = set()
        distance = [-1]*V
        distance[S] = 0

        while len(priority_queue) != 0:
            w1, node1 = heapq.heappop(priority_queue)
            if node1 in visited:
                continue
            visited.add(node1)

            for node2, w2 in adj[node1]:
                new_d = w2+w1
                if distance[node2] == -1:
                    distance[node2] = new_d
                    heapq.heappush(priority_queue, (w2+w1, node2))

                elif new_d < distance[node2]:
                    distance[node2] = new_d
                    heapq.heappush(priority_queue, (w2+w1, node2))

        return distance
```

10 Bellman Ford algorithm

Used to handle negative edge weights which can't be done using Djikstra's.

Trivial to code, involves going through all the edges and vertices.

```
class Solution:
        def isNegativeWeightCycle(self, n, edges):
                distances = [float("inf")]*n
                distances[0] = 0
                for i in range(0, n):
                    temp = distances.copy()
                    for s, d, p in edges:
                        if distances[s] == float("inf"):
                            continue
                        if distances[s] + p < temp[d]:
                            temp[d] = distances[s]+p
                    distances = temp
        return distances[-1]
```

11 Detecting Negative cycle (Bellman-Ford)

To detect a negative cycle, we run the algorithm again and see if there are any changes from the previous distances array.

```
class Solution:
        def isNegativeWeightCycle(self, n, edges):
                distances = [float("inf")]*n
                distances[0] = 0
                for i in range(0, n):
                    temp = distances.copy()
                    for s, d, p in edges:
                        if distances[s] == float("inf"):
                            continue
                        if distances[s] + p < temp[d]:
                            temp[d] = distances[s]+p
                    distances = temp
                for j in range(0, n):
                    for s, d, p in edges:
                        temp = distances.copy()
                        if distances[s] == float('inf'):
                            continue
                        if distances[s] + p < temp[d]:
                            return True
                return False
```

12 Floyd-Washall's Algorithm

Dijkstra's

Shortest path from one node to all nodes

Bellman-Ford

Shortest path from one node to all nodes - negative weights allowed.

Floyd-Washall's Algorithm

Shortest path of all vertices, negative edges are allowed.

```
class Solution:
        def shortest_distance(self, matrix):
                for k in range(0, len(matrix)):
                    for i in range(0, len(matrix)):
                        for j in range(0, len(matrix)):
                            #-1 corresponds to an edge not existing
                            if matrix[i][k]==-1 or matrix[k][j]==-1:
                                continue
                            elif matrix[i][j] == -1:
                                matrix[i][j] = matrix[i][k] + matrix[k][j]
                            elif matrix[i][j] > matrix[i][k] + matrix[k][j]:
                                matrix[i][j] = matrix[i][k] + matrix[k][j]
                return matrix
```

13 Prim's algorithm

You are given an array points representing integer coordinates of some points on a 2D-plane, where points[i] = [xi, yi].

Return the minimum cost to make all points connected. All points are connected if there is exactly one simple path between any two points.

Input: points = [[0,0],[2,2],[3,10],[5,2],[7,0]] Output: 20 Explanation:

We can connect the points as shown above to get the minimum cost of 20. Notice that there is a unique path between every pair of points.

Every node connected together without any cycles.

```
import heapq
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        adj_list = {}
        for i in range(0, len(points)):
            adj_list[i] = []
        for j in range(0, len(points)):
            x1, y1 = points[j]
            for k in range(j+1, len(points)):
                x2, y2 = points[k]
                manhattan_distance = abs(x1-x2) + abs(y1-y2)
                adj_list[k].append((manhattan_distance, j))
                adj_list[j].append((manhattan_distance, k))

        #Prims algorithm
        visited = set()
        res = 0
        min_heap = [(0,0)]
        while len(visited) != len(points):
            popped = heapq.heappop(min_heap)
            if popped[1] in visited:
                continue
            visited.add(popped[1])
            res+= popped[0]
            for neighbors in adj_list[popped[1]]:
                if neighbors[1] not in visited:
                    heapq.heappush(min_heap, neighbors)
        return res
```

14 Course Schedule

Input: numCourses = 2, prerequisites = [[1,0],[0,1]] Output: false

Explanation: There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

Input: numCourses = 2, prerequisites = [[1,0]] Output: true

Explanation: There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        # step 1, create an adjacency list
        # step 2, create a visiting set
        # step 3, run dfs
        adj_list = {}
        for i in range(0, numCourses):
            adj_list[i] = []
        for req in prerequisites:
            adj_list[req[0]].append(req[1])
        visited = set()

        def dfs(course):
            if course in visited:
                return False
            if adj_list[course] == []:
                return True
            visited.add(course)
            for reqs in adj_list[course]:
                    if not dfs(reqs):
                        return False
            visited.remove(course)
            adj_list[course] = []
            return True

        for n in range(numCourses):
            if not dfs(n):
                return False
        return True
```

15 Course Schedule

In short detecting cycle in graph.

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        adj_list = {}
        for i in range(0, numCourses):
            adj_list[i] = []
        for req in prerequisites:
            adj_list[req[0]].append(req[1])

        visited = set()
        anscestor = set()

        def dfs(course):
            visited.add(course)
            anscestor.add(course)
            for reqs in adj_list[course]:
                if reqs not in visited:
                    if dfs(reqs) == False:
                        return False
                elif reqs in anscestor:
                    return False
            anscestor.remove(course)
            return True

        for n in range(numCourses):
            if n not in visited:
                if dfs(n) == False:
                    return False
        return True
```

16 Course Schedule II

Return a topological sort, taking care there are no cycles.

```
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:
        adj_map = {}
        for i in range(numCourses):
            adj_map[i] = []
        for arr in prerequisites:
            start = arr[0]
            end = arr[1]
            adj_map[start].append(end)

        visited = set()
        anscestor = set()
        out = []

        def dfs(node):
            visited.add(node)
            anscestor.add(node)
            for child in adj_map[node]:
                if child not in visited:
                    if dfs(child) == False:
                        return False
                elif child in anscestor:
                    return False
            out.append(node)
            anscestor.remove(node)
            return True

        for n in range(numCourses):
            if n not in visited:
                if dfs(n) == False:
                    return []
        return out
```

17 Number of Islands

Input: grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]]

Output: 1

Input: grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]]

Output: 3

```
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        count = 0
        def dfs(grid, i, j):
            if i >= len(grid) or i<0 or j <0 or j >= len(grid[0]):
                return
            if grid[i][j] == "0":
                return
            grid[i][j] = "0"
            dfs(grid, i+1, j)
            dfs(grid, i-1, j)
            dfs(grid, i, j+1)
            dfs(grid, i, j-1)
            return

        for i in range(0, len(grid)):
            for j in range(0, len(grid[0])):
                if grid[i][j] == "1":
                    dfs(grid,i,j)
                    count += 1
        return count
```

18 Clone Graph

Given a reference of a node in a connected undirected graph.

Return a deep copy (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

```
class Node {

    public int val;

    public List<Node> neighbors;

}
class Solution:
    def cloneGraph(self, node: 'Node') -> 'Node':
        old_to_new = {}
        def dfs(node):
            if node == None:
                return None

            if node in old_to_new:
                return old_to_new[node]

            copy = Node(node.val)
            old_to_new[node] = copy
            for vals in node.neighbors:
                copy.neighbors.append(dfs(vals))
            return copy

        return dfs(node)
```

19 Pacific Atlantic Waterflow

The Pacific Ocean touches the island's left and top edges, and the Atlantic Ocean touches the island's right and bottom edges.

The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is less than or equal to the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

NOTE: The approach is to move from the pacific to areas that can be reached by the edge blocks, perform the same for the atlantic and return the coordinates that are intersections of both places.

```
class Solution:
    def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:
        pacific = set()
        atlantic = set()
        def dfs(heights, i, j, visited, prev):
            if i<0 or j<0 or i>=len(heights) or j>= len(heights[0]):
                return
            if (i, j) in visited:
                return
            if heights[i][j] < prev:
                return
            visited.add((i,j))
            dfs(heights, i+1, j, visited, heights[i][j])
            dfs(heights, i-1, j, visited, heights[i][j])
            dfs(heights, i, j+1, visited, heights[i][j])
            dfs(heights, i, j-1, visited, heights[i][j])
            return
        #top_row
        for i in range(0, len(heights[0])):
            dfs(heights, 0, i, pacific, float('-inf'))
        #Left column
        for j in range(0, len(heights)):
            dfs(heights, j, 0, pacific, float('-inf'))
        #bottom row
        for k in range(0, len(heights[0])):
            dfs(heights, len(heights)-1, k, atlantic, float('-inf'))
        #right column
        for l in range(0, len(heights)):
            dfs(heights, l, len(heights[0])-1, atlantic, float('-inf'))
        res = []
        for i in range(0, len(heights)):
            for j in range(0, len(heights[0])):
                if (i, j) in pacific and (i,j) in atlantic:
                    res.append((i, j))
        return res
```

20 Network Time delay

Dijkstra's algorithm where we find the minimum distance from source vertex "k".

You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k. Return the time it takes for wall the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2 Output: 2

```
import heapq
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        adj_map = {}
        for n in range(1, n+1):
            adj_map[n] = []
        for arr in times:
            start = arr[0]
            end = arr[1]
            weight = arr[2]
            if start in adj_map:
                adj_map[start].append((weight, end))
            else:
                adj_map[start] = (weight, end)
        min_heap = [(0, k)]
        t = -1
        visited = set()
        while len(min_heap) != 0:
            city = heapq.heappop(min_heap)
            if city[1] in visited:
                continue
            visited.add(city[1])
            t = max(t, city[0])
            for neighbors in adj_map[city[1]]:
                if neighbors not in visited:
                    heapq.heappush(min_heap, (neighbors[0]+city[0], neighbors[1]))
        return t if len(visited) == n else -1
```

21 Word Search

Given an m x n grid of characters board and a string word, return true if word exists in the grid. The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

```
class Solution:
    def exist(self, matrix: List[List[str]], word: str) -> bool:
        visited = set()

        def dfs(matrix, i, j, count):
            if count == len(word):
                return True

            if i < 0 or j< 0 or i >= len(matrix) or j>= len(matrix[0]):
                return False

            if word[count] != matrix[i][j]:
                return False

            if (i, j) in visited:
                return False

            visited.add((i, j))

            a = dfs(matrix, i+1, j, count+1)
            b = dfs(matrix, i-1, j, count+1)
            c = dfs(matrix, i, j+1, count+1)
            d = dfs(matrix, i, j-1, count+1)
            visited.remove((i, j))

            return a or b or c or d

        for i in range(0, len(matrix)):
            for j in range(0, len(matrix[0])):
                if word[0] == matrix[i][j]:
                    if dfs(matrix, i, j, 0):
                        return True
        return False
```

22 Island Perimeter

You are given row x col grid representing a map where grid[i][j] = 1 represents land and grid[i][j] = 0 represents water.

Grid cells are connected horizontally/vertically (not diagonally). The grid is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells).

The island doesn't have "lakes", meaning the water inside isn't connected to the water around the island. One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

Input: grid = [[0,1,0,0],[1,1,1,0],[0,1,0,0],[1,1,0,0]] Output: 16 Explanation: The perimeter is the 16 yellow stripes in the image above.

```
class Solution:
    def islandPerimeter(self, grid: List[List[int]]) -> int:

        visited = set()

        def dfs(matrix, i, j):
            if i < 0 or j < 0 or i >= len(matrix) or j >= len(matrix[0]):
                return 1
            if matrix[i][j] == 0:
                return 1
            if (i, j) in visited:
                return 0
            visited.add((i, j))
            return dfs(matrix, i+1, j)+dfs(matrix, i-1, j)+dfs(matrix, i, j+1)+dfs(matr

        for i in range(0, len(grid)):
            for j in range(0, len(grid[0])):
                if grid[i][j] == 1:
                    return dfs(grid, i, j)
        return -1
```