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

```python
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
```