**Graphs**

1. Depth first search
2. Breadth first search
3. Topological sort
4. Detect cycle in a directed graph using DFS
5. Detect cycle in a directed graph using BFS
6. Detect cycle in a undirected graph using DFS
7. Detect cycle in a undirected graph using BFS
8. Course schedule
9. Course schedule 2
10. Alien dictionary
11. Prim’s algo
12. Min cost between cities
13. Number of connected components in undirected graph
14. Dijkstra’s algo
15. Floyd warshall algo
16. Evaluate division
17. Clone a graph
18. Bipartite graph (graph coloring)
19. Longest increasing path in a matrix
20. Path with min max value

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def addEdge(self,u,v):

        self.graph[u].append(v)

    # 1. Visit one vertex, go to one of its adjacent vertex and then repeat

    def DFS(self,s):

        visited = set()

        self.DFSUtil(s, visited)

    def DFSUtil(self, v, visited):

        if v not in visited:

            visited.add(v)

        print(v, end=" ")

        for i in self.graph[v]:

            if i not in visited:

                visited.add(i)

                self.DFSUtil(i, visited)

g = Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

g.DFS(1)

Bread first search

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.list = defaultdict(list)

        # print(self.list)

    def addEdge(self,u,v):

        self.list[u].append(v)

    def BFS(self,start\_node):

        # initialize visited list

        visited = set()

        # For BFS, queue is used

        queue = []

        # 1. Mark the node as visited and add to queue

        queue.append(start\_node)

        visited.add(start\_node)

        while queue:

            s = queue.pop(0) # pop 2

            print(s, end=" ")

            # add vertices which are edges to start node(2)

            for i in self.list[s]:

                if i not in visited:

                    queue.append(i)

                    visited.add(i)

g = Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

g.BFS(2)

Detect cycle in directed graph using DFS

Detect cycle in directed graph using BFS

Detect cycle in undirected graph using DFS

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def addEdge(self,u,v):

        self.graph[u].append(v)

        self.graph[v].append(u)

    def hasCycle(self, node):

        visited = set()

        print(self.dfs(node, visited, -1))

    def dfs(self,node, visited, parent):

        visited.add(node)

        for neighbor in self.graph[node]:

            if neighbor not in visited:

                # visited.add(node)

                if self.dfs(neighbor, visited, node):

                    return True

            elif parent != neighbor:

                return True

        return False

        # print(visited)

g = Graph()

g.addEdge(1, 2)

g.addEdge(2, 3)

g.addEdge(3, 4)

g.addEdge(4, 5)

g.addEdge(5, 2)

g.hasCycle(1)

Topological sort

'''

1. a vertex is pushed to stack only when all of its adjacent vertices

(and their adjacent vertices and so on) are already in stack.

Time complexity: O(V+E)

Space complexity: O(V)

Algorithm:

1. Visit one vertex and then recursively visit its neighbors

2. Push the neighbor which has no more neighbors into the stack

3. Do it for all vertices

'''

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self, vertices):

        self.graph = defaultdict(list)

        self.V = vertices

    def addEdge(self, u,v):

        self.graph[u].append(v)

    def topologicalUtil(self,v,visited,result):

        # set vertex as visited

        visited[v] = True

        # look for its adjacent vertices

        for i in self.graph[v]:

            if visited[i] == False:

                self.topologicalUtil(i,visited,result)

        result.insert(0,v)

    def topological\_sort(self):

        visited = [False] \* self.V

        result = []

        for i in range(self.V):

            if visited[i] == False:

                self.topologicalUtil(i,visited, result)

        print(result)

g= Graph(6)

g.addEdge(5, 2);

g.addEdge(5, 0);

g.addEdge(4, 0);

g.addEdge(4, 1);

g.addEdge(2, 3);

g.addEdge(3, 1);

g.topological\_sort()

Course schedule

'''

Approach: Detect cycle using BFS

Difficulties faced:

Steps to resolve Difficulties:

Time complexity: O(V+E)

Space complexity: O(V)

Algorithm:

1. Detect a cycle in the graph

'''

from collections import defaultdict

class Solution:

    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:

        # num of courses = num of vertices

        # prerequisites is the [u,v] (edges)

        # detect a cycle in the graph

        indegree = [0] \* numCourses

        graph = defaultdict(list)

        # create adjacency list

        for pr in prerequisites:

            # pr[1] = vertex

            # pr[0] = adj vertex

            graph[pr[1]].append(pr[0])

            indegree[pr[0]] += 1

        # push only those vertices whose indegree is 0

        queue = []

        for vertex in range(len(indegree)):

            # if indegree[i] is 0, add to queue

            if indegree[vertex] == 0:

                queue.append(vertex)

        count = 0

        while queue:

            # pop

            v = queue.pop(0)

            # check for popped vertex neighbors and decrement their indegree

            for n in graph[v]:

                indegree[n] -= 1

                if indegree[n] == 0:

                    queue.append(n)

            # as we visited a node

            count += 1

        return count == numCourses

Course schedule 2

'''

Approach: Topological sort

Difficulties faced:

Steps to resolve Difficulties:

Time complexity: O(V+E)

Space complexity: O(V)

Algorithm:

'''

from collections import defaultdict

class Solution:

    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List:

        # num of courses = num of vertices

        # prerequisites is the [u,v] (edges)

        # detect a cycle in the graph

        indegree = [0] \* numCourses

        graph = defaultdict(list)

        result = []

        # create adjacency list

        for pr in prerequisites:

            # pr[1] = vertex

            # pr[0] = adj vertex

            graph[pr[1]].append(pr[0])

            indegree[pr[0]] += 1

        # push only those vertices whose indegree is 0

        queue = []

        for vertex in range(len(indegree)):

            # if indegree[i] is 0, add to queue

            if indegree[vertex] == 0:

                queue.append(vertex)

        count = 0

        while queue:

            # pop

            v = queue.pop(0)

            result.append(v)

            # check for popped vertex neighbors and decrement their indegree

            for n in graph[v]:

                indegree[n] -= 1

                if indegree[n] == 0:

                    queue.append(n)

            # as we visited a node

            count += 1

        #print(result)

        return result if count == numCourses else []

332 Reconstruct itinerary

'''

Approach:

Difficulties faced:

Steps to resolve Difficulties:

Time complexity: O(E/2 \* log E/2)

Space complexity: O(V + 2E)

Algorithm:

1. Sort the neighbors of the graph in reverse order

2. Pop the neighbor and do DFS

3. Add the node to the result when its neighbor list is empty

4. Reverse the result list

'''

class Solution:

    def findItinerary(self, tickets: List[List[str]]) -> List[str]:

        from collections import defaultdict

        self.flightMap = defaultdict(list)

        for ticket in tickets:

            origin, dest = ticket[0], ticket[1]

            self.flightMap[origin].append(dest)

        for origin, itinerary in self.flightMap.items():

        # Note that we could have multiple identical flights, i.e. same origin and destination.

            itinerary.sort(reverse=True)

        self.result = []

        self.DFS('JFK')

        return self.result[::-1]

    def DFS(self, origin):

        destlist = self.flightMap[origin]

        while destlist:

            nextDest = destlist.pop()

            self.DFS(nextDest)

        self.result.append(origin)

261 Graph Valid Tree

class Solution:

    def validTree(self, n: int, edges: List[List[int]]) -> bool:

        self.graph = collections.defaultdict(list)

        for e in edges:

            self.graph[e[0]].append(e[1])

            self.graph[e[1]].append(e[0])

        visited = set()

        # if cycle then not a tree

        if self.hasCycle(0, visited, -1):

            return False

        # if all nodes are not connected

        if len(visited) != n:

            return False

        return True

    def hasCycle(self,node, visited, parent):

        visited.add(node)

        for neighbor in self.graph[node]:

            if neighbor not in visited:

                if self.hasCycle(neighbor, visited, node):

                    return True

            elif parent != neighbor:

                return True

        return False

**1135. Connecting Cities With Minimum Cost**

**A screenshot of a cell phone

Description automatically generated**

class Solution:

    def minimumCost(self, N: int, connections: List[List[int]]) -> int:

        graph = collections.defaultdict(list)

        for city1, city2, cost in connections:

            graph[city1].append((cost,city2))

            graph[city2].append((cost,city1))

        queue = [(0,N)]

        visited = set()

        total\_cost = 0

        path = {}

        while queue and len(visited) != N:

            cost, city = heapq.heappop(queue)

            # if (cost,city) not in path:

            #     path[(cost,city)] = N

            if city not in visited:

                visited.add(city)

                total\_cost += cost

                for cost, city in graph[city]:

                    heapq.heappush(queue, (cost,city))

        return total\_cost if len(visited) == N else -1

Clone graph

"""

# Definition for a Node.

class Node:

    def \_\_init\_\_(self, val = 0, neighbors = []):

        self.val = val

        self.neighbors = neighbors

"""

'''

Approach:

Difficulties faced:

Steps to resolve Difficulties:

Time complexity: O(V+E)

Space complexity: O(V)

Algorithm:

1. Using BFS, clone each node in the graph. Store the map between old\_node and new node

2. Iterate through the hashmap and update the neighbors of the new nodes by referencing old\_node's neighbors

Node1 {

val = 1

neighbors = [Node2, Node3]

}

hashmap= {'Node1':new\_Node1, 'Node2': new\_Node2}

'''

class Solution:

    def cloneGraph(self, node: 'Node') -> 'Node':

        # BFS

        if not node:

            return

        def BFS(node):

            visited = set()

            queue = []

            # add node to the queue

            queue.insert(0,node)

            # update visited

            visited.add(node)

            while queue:

                node = queue.pop()

                # clone the node

                clone\_node = Node()

                clone\_node.val = node.val

                hashmap[node] = clone\_node

                for nei in  node.neighbors:

                    if nei not in visited:

                        visited.add(nei)

                        queue.insert(0,nei)

        hashmap = {}

        BFS(node)

        for key, val in hashmap.items():

            for nei in key.neighbors:

                c\_node = hashmap[nei]

                val.neighbors.append(c\_node)

        return hashmap[node]