**Graphs Algo’s**

**BFS of Graph**

from collections import deque

def bfsOfGraph(self, adj):

    queue = deque([0])  # Start BFS from node 0

    visited = set()

    visited.add(0)

    ans = []

    while queue:

        node = queue.popleft()  # O(1) operation

        ans.append(node)

        for neighbor in adj[node]:

            if neighbor not in visited:

                visited.add(neighbor)

                queue.append(neighbor)

    return ans

**Depth-First Search (DFS)**

def dfsOfGraph(self, adj):

    visited = set()

    ans = []

    def dfs(node):

        visited.add(node)  # Mark node as visited

        ans.append(node)  # Add to DFS traversal order

        for neighbor in adj[node]:

            if neighbor not in visited:

                dfs(neighbor)  # Recursive DFS call

    dfs(0)  # Start DFS from node 0

    return ans

**Kah’n algorithm**

**Detect cycle in directed graph (Topological sort with BFS)**

from collections import deque

class Solution:

    def isCyclic(self, V, adj):

        queue = deque()

        visited = 0

        topo = []

        indegree = [0] \* V  # Stores in-degree of each vertex

        # Calculate in-degree of each vertex

        for u in range(V):

            for vertex in adj[u]:

                indegree[vertex] += 1

        # Enqueue vertices with 0 in-degree

        for u in range(V):

            if indegree[u] == 0:

                queue.append(u)

        # BFS traversal

        while queue:

            node = queue.popleft()

            visited += 1

            for u in adj[node]:

                indegree[u] -= 1

                if indegree[u] == 0:

                    queue.append(u)

        return visited != V  # Cycle exists if not all vertices are visited

**Detect cycle in directed graph (using DFS)**

class Solution:

    def isCyclic(self, V, adj):

        path\_visited = [0] \* V

        visited = set()

        def dfs(node):

            nonlocal visited, path\_visited

            visited.add(node)

            path\_visited[node] = 1

            for neighbor in adj[node]:

                if neighbor not in visited:

                    if dfs(neighbor):

                        return True

                elif path\_visited[neighbor]:  # Already visited and in the current path

                    return True

            path\_visited[node] = 0  # Remove node from the current path

            return False

        for i in range(V):

            if i not in visited:

                if dfs(i):

                    return True

        return False

**Detect cycle in undirected graph(DFS)**

from typing import List

class Solution:

    def isCycle(self, V: int, adj: List[List[int]]) -> bool:

        visited = [False] \* V  # Use a list instead of a set for O(1) access

        def dfs(node, parent):

            visited[node] = True  # Mark as visited

            for neighbor in adj[node]:

                if not visited[neighbor]:  # If neighbor is unvisited, continue DFS

                    if dfs(neighbor, node):

                        return True

                elif neighbor != parent:  # If visited and not the parent, cycle found

                    return True

            return False

        for i in range(V):

            if not visited[i]:  # If node is unvisited, start DFS

                if dfs(i, -1):  # Parent is set to -1 for the first call

                    return True

        return False

**Bellmanford algorithm (Directed Graph) Shortest path.**

1. Helps in detecting negative cycles.
2. Calculate the indegree so that your update propogates outside

3 . Single souce shortest path algorithm.

1. **Find the source then start [Important].**

Time complexity: O(n\*E)

def isNegativeWeightCycle(self, n, edges):

    # Initialize distances

    dist = [float('inf')] \* n

    # Assuming the source vertex is 0

    dist[0] = 0

    # Relax all edges (n-1) times

    for \_ in range(n - 1):

        for u, v, w in edges:

            if dist[u] + w < dist[v]:

                dist[v] = dist[u] + w

    # Check for negative weight cycles

    for u, v, w in edges:

        if dist[u] + w < dist[v]:

            return 1  # Negative weight cycle found

    return 0  # No negative weight cycle found

**Floyd Warshall**

1. Multisource shortest path to all other nodes.
2. Helps in detecting negative cycles as well.
3. Time complexity v\*\*3 (V Cube )

class Solution:

    def isNegativeWeightCycle(self, n, edges):

        # Initialize the distance matrix

        dist = [[float("inf")] \* n for \_ in range(n)]

        for i in range(n):

            dist[i][i] = 0  # Distance from a node to itself is 0

        # Populate the distance matrix with given edges

        for u, v, w in edges:

            dist[u][v] = w

        # Floyd-Warshall algorithm to find shortest paths

        for k in range(n):

            for i in range(n):

                for j in range(n):

                    if dist[i][k] != float("inf") and dist[k][j] != float("inf"):

                        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

        # Check for negative weight cycles on diagonal

        for i in range(n):

            if dist[i][i] < 0:

                return 1  # Negative weight cycle exists

        return 0  # No negative weight cycle

**Dijkshtra’s Algorithm**

1. it is not applicable for negative weight edge cycles.
2. Doesn’t support negative weight edges. [It will fall in infinite loop]
3. Time complexity : E(log V)
4. Find shortest path from a given node to all other vertices.
5. Time complexity : V\*V (vertices \* Total number of vertices it is relaxing)

from collections import defaultdict

import heapq

class Solution:

    def shortestPath(self, V: int, E: int, edges: List[List[int]]) -> List[int]:

        adj = defaultdict(list)

        for u, v, w in edges:

            adj[u].append([v, w])

        dist = [float("inf")] \* V

        dist[0] = 0

        pq = []

        heapq.heappush(pq, [0, 0])

        while pq:

            edge\_weight, node = heapq.heappop(pq)

            for v, w in adj[node]:

                if w + edge\_weight < dist[v]:

                    dist[v] = w + edge\_weight

                    heapq.heappush(pq, [dist[v], v])

        for i in range(V):

            if dist[i] == float("inf"):

                dist[i] = -1

        return dist

**Prims Algorithm**

1. Time complexity :v(loge)
2. It is used for find the minimum spanning tree

from typing import List

import heapq

from collections import defaultdict

class Solution:

    def spanningTree(self, V: int, adj: List[List[int]]) -> int:

        sum\_ = 0

        pq = []

        heapq.heappush(pq, [0, 0])

        visited = set()

        while pq:

            node\_weight, node = heapq.heappop(pq)

            if node in visited:

                continue

            visited.add(node)

            sum\_ += node\_weight

            for neighbor, weight in adj[node]:

                if neighbor not in visited:

                    heapq.heappush(pq, [weight, neighbor])

        return sum\_

**Krushkal Algorithm (Using union find or Disjoint set)**

1.Time Complexity :O(E log E) or vlog E

2. Used for finding minimum spanning tree

from typing import List

class UnionFind:

    def \_\_init\_\_(self, n):

        self.parent = list(range(n))

        self.rank = [0] \* n

    def find(self, i):

        if self.parent[i] != i:

            self.parent[i] = self.find(self.parent[i])  # Path compression

        return self.parent[i]

    def union(self, i, j):

        i\_root = self.find(i)

        j\_root = self.find(j)

        if i\_root == j\_root:

            return False  # Already in the same set

        if self.rank[i\_root] < self.rank[j\_root]:

            self.parent[i\_root] = j\_root

        elif self.rank[i\_root] > self.rank[j\_root]:

            self.parent[j\_root] = i\_root

        else:

            self.parent[j\_root] = i\_root

            self.rank[i\_root] += 1

        return True

class Solution:

    def spanningTree(self, V: int, adj: List[List[int]]) -> int:

        edges = []

        for u in range(V):

            for v, w in adj[u]:

                edges.append((w, u, v))  # (weight, source, destination)

        edges.sort()  # Sort edges by weight in ascending order

        uf = UnionFind(V)

        mst\_cost = 0

        for weight, u, v in edges:

            if uf.union(u, v):

                mst\_cost += weight

        return mst\_cost

**Tarjan’s Algorithm [for articulation point]**

1. For finding whether a graph has a articulation point or not which make it disconnected
2. Graph can be connected or disconnected
3. import sys
4. sys.setrecursionlimit(10\*\*6)
5. class Solution:
6. def dfs(self, node, parent, vis, adj, articulation\_points, timer, low\_time,
7. insertion\_time):
8. vis[node] = 1
9. low\_time[node] = insertion\_time[node] = timer[0]
10. timer[0] += 1
11. children = 0  # Count of children in DFS tree
12. for neighbor in adj[node]:
13. if neighbor == parent:
14. continue
15. if not vis[neighbor]:
16. children += 1
17. self.dfs(neighbor, node, vis, adj, articulation\_points, timer,
18. low\_time, insertion\_time)
19. low\_time[node] = min(low\_time[node], low\_time[neighbor])
20. # Check for articulation point
21. if parent != -1 and low\_time[neighbor] >= insertion\_time[node]:
22. if node not in articulation\_points:
23. articulation\_points.append(node)
24. else:
25. low\_time[node] = min(low\_time[node], insertion\_time[neighbor])
26. # Special case for root node
27. if parent == -1 and children > 1:
28. if node not in articulation\_points:
29. articulation\_points.append(node)
30. def articulationPoints(self, V, adj):
31. vis = [0] \* V
32. low\_time = [0] \* V
33. insertion\_time = [0] \* V
34. articulation\_points = []
35. timer = [0]
36. # Handle disconnected graph
37. for i in range(V):
38. if not vis[i]:
39. self.dfs(i, -1, vis, adj, articulation\_points, timer,
40. low\_time, insertion\_time)
41. return sorted(articulation\_points) if articulation\_points else [-1]

**Tarjans Algo [for finding bridge]**

1. for Finding whether a graph has bridge or not which make it disconnected
2. Graph can we connected for disconnected
3. class Solution:
4. def dfs(self, node, parent, adj, vis, c, d, low, intime, timer):
5. vis[node] = 1
6. low[node] = intime[node] = timer[0]
7. timer[0] += 1
8. for neighbor in adj[node]:
9. if neighbor == parent:
10. continue
11. if not vis[neighbor]:
12. if self.dfs(neighbor, node, adj, vis, c, d, low, intime, timer):
13. return 1
14. low[node] = min(low[node], low[neighbor])
15. if low[neighbor] > intime[node]:  # Condition for bridge
16. return 1
17. else:
18. low[node] = min(low[node], low[neighbor])
19. return 0
20. #Function to find if the given edge is a bridge in graph.
21. def isBridge(self, V, adj, c, d):
22. vis = [0] \* V
23. timer = [0]
24. low = [0] \* V
25. intime = [0] \* V
26. # Run DFS for all components (important for disconnected graphs)
27. for i in range(V):
28. if not vis[i]:
29. if self.dfs(i, -1, adj, vis, c, d, low, intime, timer):
30. return 1
31. return 0

**Kosaraju algorithm**

**Complexity**

* **Time Complexity:**
  + O(V+E)O(V + E)O(V+E), where VVV is the number of vertices and EEE is the number of edges.
  + Each DFS traversal and graph transposition operation runs in linear time.
* **Space Complexity:**
  + O(V+E)O(V + E)O(V+E) for storing the adjacency list and its transpose.
  + O(V)O(V)O(V) for the visited set and stack.

class Solution:

    def dfs(self,node,adj,vis,stack):

        vis.add(node)

        for neigh in adj[node]:

            if neigh not in vis:

                self.dfs(neigh,adj,vis,stack)

        stack.append(node)

    def kosaraju(self, adj):

        vis = set()

        stack = []

        scc = 0

        for i in range(len(adj)):

            if i not in vis:

                self.dfs(i,adj,vis,stack)

        adjT = [[] for \_ in range(len(adj))]

        for i in range(len(adj)):

            for j in adj[i]:

                adjT[j].append(i)

        vis.clear()

        while stack:

            i = stack.pop()

            if i not in vis:

                self.dfs(i,adjT,vis,[])

                scc+=1

        return scc

**Rotten Oranges (BFS on Matrix)**

from collections import deque

class Solution:

    def orangesRotting(self, grid: List[List[int]]) -> int:

        pq = deque()

        m = len(grid)

        n = len(grid[0])

        delRow = [0,1,0,-1]

        delCol = [1,0,-1,0]

        freshcount = 0

        for i in range(m):

            for j in range(n):

                if grid[i][j] == 2:

                    pq.append([0,i,j])

                if grid[i][j] == 1:

                    freshcount += 1

        time = 0

        cnt = 0

        while pq:

            t,row,col = pq.popleft()

            time = max(time,t)

            for i in range(4):

                newrow = row + delRow[i]

                newcol = col + delCol[i]

                if 0 <= newrow < m and 0 <= newcol < n and grid[newrow][newcol] == 1:

                    grid[newrow][newcol] = 2

                    pq.append([t+1,newrow,newcol])

                    cnt+=1

        return time if cnt == freshcount else -1

**Topological Sort (DFS)**

def topological\_sort\_dfs(graph):

    visited = set()

    stack = []

    def dfs(node):

        visited.add(node)

        # Explore all neighbors

        for neighbor in graph[node]:

            if neighbor not in visited:

                dfs(neighbor)

        # Push node to stack after visiting all its neighbors

        stack.append(node)

    # Perform DFS on all unvisited nodes

    for node in graph:

        if node not in visited:

            dfs(node)

    # The stack now contains the nodes in reverse topological order

    return stack[::-1]  # Reverse the stack to get the correct order

### Summary Table:

| **Feature** | **Dijkstra’s Algorithm** | **Prim’s Algorithm** |
| --- | --- | --- |
| **Purpose** | Shortest path from source to all nodes | Minimum spanning tree |
| **Graph Type** | Directed or undirected, with positive weights | Undirected graph with positive weights |
| **Edge Selection** | Minimum tentative distance to any node | Minimum edge connecting MST to non-MST vertex |
| **Priority Queue** | Uses a priority queue for shortest paths | Uses a priority queue for minimum edge selection |
| **Negative Edge Weights** | Not supported (unless modified, e.g., Bellman-Ford) | Can handle negative weights |
| **Time Complexity** | O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV) with priority queue | O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV) with priority queue |
| **Stopping Condition** | When shortest path to all nodes is found | When all vertices are included in the MST |

Both algorithms have greedy approaches but serve different purposes and are applied in different scenarios.