Graph Algorithms Tutorial

This tutorial will guide you through the most fundamental concepts in Graph Theory and its algorithms, with explanations and code snippets in C++ or Python.

Graphs, Representation (Adjacency Matrix/List)

Graphs are mathematical structures used to model pairwise relations between objects. They consist of vertices (nodes) connected by edges (links).

- **Adjacency Matrix:** This is a 2D array where the cell at row i and column j indicates whether there is an edge between vertex i and vertex j. For weighted graphs, the cell contains the weight of the edge. This representation is efficient for dense graphs but consumes $O(V^2)$ space, where V is the number of vertices.
- **Adjacency List:** Each vertex maintains a list of adjacent vertices. This is more space-efficient for sparse graphs, using $O(V+E)$ space, where E is the number of edges. It is the most commonly used representation in practice, especially for large, sparse graphs.

BFS (Breadth-First Search)

Breadth-First Search is an algorithm for traversing or searching tree or graph data structures. It starts from a selected node (source) and explores all its neighbors before moving on to the next level of neighbors. BFS uses a queue to keep track of the next node to visit. It is particularly useful for finding the shortest path in unweighted graphs, level-order traversal, and checking graph connectivity. BFS guarantees the shortest path in terms of the number of edges traversed.

DFS (Depth-First Search)

Depth-First Search is another traversal algorithm that explores as far as possible along each branch before backtracking. It can be implemented recursively or using a stack. DFS is useful for tasks like detecting cycles, topological sorting, and finding connected components in a graph. Unlike BFS, DFS may not find the shortest path in unweighted graphs but is more memory-efficient for deep graphs.

Topological Sort

Topological sorting is the linear ordering of vertices in a Directed Acyclic Graph (DAG) such that for every directed edge uv, vertex u comes before v in the ordering. It is used in scenarios like scheduling tasks, resolving dependencies, and course prerequisite planning. Topological sort can be performed using DFS or Kahn's algorithm (BFS-based).

Cycle Detection

Cycle detection determines whether a given graph contains a cycle. In undirected graphs, cycles can be detected using DFS by checking for back edges. In directed

graphs, cycles are detected by maintaining a recursion stack during DFS; if a node is revisited while still in the stack, a cycle exists. Cycle detection is essential in deadlock detection, dependency resolution, and graph validation.

Shortest Paths (Dijkstra, Bellman-Ford)

- **Dijkstra's Algorithm:** Finds the shortest path from a single source to all other nodes in a graph with non-negative edge weights. It uses a priority queue to always expand the closest unvisited node. It is efficient and widely used in network routing and mapping applications.
- **Bellman-Ford Algorithm:** Handles graphs with negative edge weights and can detect negative cycles. It works by repeatedly relaxing all edges and updating distances. It is slower than Dijkstra's but more versatile due to its ability to handle negative weights.

MST (Minimum Spanning Tree)

A Minimum Spanning Tree is a subset of the edges of a connected, undirected, weighted graph that connects all the vertices together, without any cycles, and with the minimum possible total edge weight. MSTs are used in network design, clustering, and other optimization problems. The two most common algorithms for finding MSTs are Kruskal's and Prim's algorithms.

Union Find (Disjoint Set Union)

Union Find is a data structure that keeps track of elements partitioned into a number of disjoint (non-overlapping) sets. It supports two operations efficiently:

- **Find:** Determine which set a particular element belongs to.
- **Union:** Merge two sets into a single set.
  It is used in Kruskal's algorithm for MST, cycle detection in graphs, and network connectivity problems. Path compression and union by rank optimize its performance.

Tarjan's Algorithm

Tarjan's Algorithm is used to find strongly connected components (SCCs) in a directed graph. An SCC is a maximal subset of vertices such that every vertex is reachable from every other vertex in the same subset. The algorithm uses DFS and assigns each node a unique index and a low-link value, efficiently identifying all SCCs in linear time relative to the number of nodes and edges.

Kosaraju's Algorithm

Kosaraju's Algorithm is another method to find strongly connected components in a directed graph. It works in two passes:

1. Perform a DFS to compute the finishing times of nodes.
2. Reverse the direction of all edges (transpose the graph) and perform DFS in the order of decreasing finishing times.
   Each DFS in the second pass identifies an SCC. The algorithm is simple and effective, running in linear time.

1. Introduction to Graphs

A **graph** is a collection of nodes (or vertices) connected by edges. It can be:

- **Directed** or **Undirected**
- **Weighted** or **Unweighted**

Applications: Social networks, maps, networks, recommendation systems.

---

2. Graph Representation

a. Adjacency Matrix

A 2D matrix where `matrix[i][j]` indicates an edge from vertex `i` to `j`.

Code (Python):

```
V = 4
adj_matrix = [[0] * V for _ in range(V)]
adj_matrix[0][1] = 1
adj_matrix[1][2] = 1
adj_matrix[2][3] = 1
```

b. Adjacency List

A list of lists or a dictionary, each index or key maps to a list of neighbors.

Code (Python):

```
graph = {
    0: [1],
    1: [2],
    2: [3],
    3: []
}
```

---

3. Breadth-First Search (BFS)

BFS explores nodes level by level using a queue.

Code (Python):

```
from collections import deque
```

```python
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            queue.extend(graph[node])
```

---

4. Depth-First Search (DFS)

DFS explores as far as possible along each branch before backtracking.

Code (Python):

```python
def dfs(graph, node, visited=set()):
    if node not in visited:
        print(node, end=' ')
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)
```

---

5. Topological Sort

Applicable for Directed Acyclic Graphs (DAGs). It sorts vertices such that for every edge u->v, u comes before v.

Code (DFS-based):

```python
def topo_sort(graph):
    visited = set()
    stack = []

    def dfs(node):
        if node not in visited:
            visited.add(node)
            for neighbor in graph[node]:
                dfs(neighbor)
            stack.append(node)

    for node in graph:
        dfs(node)

    return stack[::-1]
```

---

6. Cycle Detection

a. In Directed Graph (using DFS)

```
def has_cycle(graph):
    visited = set()
    rec_stack = set()

    def dfs(v):
        visited.add(v)
        rec_stack.add(v)
        for neighbor in graph[v]:
            if neighbor not in visited and dfs(neighbor):
                return True
            elif neighbor in rec_stack:
                return True
        rec_stack.remove(v)
        return False

    for node in graph:
        if node not in visited:
            if dfs(node):
                return True
    return False
```

---

7. Shortest Paths

a. Dijkstra's Algorithm

For weighted graphs with non-negative weights.

Code (Python):

```
import heapq

def dijkstra(graph, start):
    dist = {v: float('inf') for v in graph}
    dist[start] = 0
    pq = [(0, start)]

    while pq:
        current_dist, u = heapq.heappop(pq)
        for v, weight in graph[u]:
            if dist[v] > current_dist + weight:
                dist[v] = current_dist + weight
                heapq.heappush(pq, (dist[v], v))

    return dist
```

b. Bellman-Ford Algorithm

Handles negative weights; detects negative cycles.

Code (Python):

```python
def bellman_ford(edges, V, start):
    dist = [float('inf')] * V
    dist[start] = 0

    for _ in range(V - 1):
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # Negative cycle check
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            print("Negative cycle detected")
            return None

    return dist
```

---

8. Minimum Spanning Tree (MST)

a. Kruskal's Algorithm

Sort all edges, pick smallest, avoid cycles using Union Find.

Code (Python):

```python
def find(parent, i):
    if parent[i] != i:
        parent[i] = find(parent, parent[i])
    return parent[i]

def union(parent, rank, x, y):
    xroot = find(parent, x)
    yroot = find(parent, y)
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    else:
        parent[yroot] = xroot
        rank[xroot] += 1

def kruskal(V, edges):
    result = []
```

```
    parent = list(range(V))
    rank = [0] * V
    edges.sort(key=lambda x: x[2])

    for u, v, w in edges:
        if find(parent, u) != find(parent, v):
            result.append((u, v, w))
            union(parent, rank, u, v)

    return result
```

---

9. Union Find (Disjoint Set Union - DSU)

Efficient way to check whether two nodes are in the same connected component.

Code (Python):

```
def find(parent, x):
    if parent[x] != x:
        parent[x] = find(parent, parent[x])
    return parent[x]

def union(parent, rank, x, y):
    xroot = find(parent, x)
    yroot = find(parent, y)
    if xroot != yroot:
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        else:
            parent[yroot] = xroot
            if rank[xroot] == rank[yroot]:
                rank[xroot] += 1
```

---

10. Tarjan's Algorithm (Strongly Connected Components)

Code (Python):

```
index = 0
stack = []
indices = {}
lowlink = {}
on_stack = set()
sccs = []

def tarjan(graph):
    def strongconnect(v):
```

```
        global index
        indices[v] = lowlink[v] = index
        index += 1
        stack.append(v)
        on_stack.add(v)

        for w in graph[v]:
            if w not in indices:
                strongconnect(w)
                lowlink[v] = min(lowlink[v], lowlink[w])
            elif w in on_stack:
                lowlink[v] = min(lowlink[v], indices[w])

        if lowlink[v] == indices[v]:
            scc = []
            while True:
                w = stack.pop()
                on_stack.remove(w)
                scc.append(w)
                if w == v:
                    break
            sccs.append(scc)

    for v in graph:
        if v not in indices:
            strongconnect(v)
    return sccs
```

---

11. Kosaraju's Algorithm (SCC)

Performs 2 DFS traversals, one on the original graph and one on the transpose.

Code (Python):

```
def kosaraju(graph):
    visited = set()
    order = []

    def dfs(v):
        visited.add(v)
        for u in graph[v]:
            if u not in visited:
                dfs(u)
        order.append(v)

    for v in graph:
        if v not in visited:
            dfs(v)
```

```
    transpose = {v: [] for v in graph}
    for v in graph:
        for u in graph[v]:
            transpose[u].append(v)

    visited.clear()
    sccs = []

    def dfs_rev(v, comp):
        visited.add(v)
        comp.append(v)
        for u in transpose[v]:
            if u not in visited:
                dfs_rev(u, comp)

    for v in reversed(order):
        if v not in visited:
            comp = []
            dfs_rev(v, comp)
            sccs.append(comp)

    return sccs
```

This tutorial provides a concise yet comprehensive guide to the most common graph algorithms. Use these as foundational blocks in competitive programming, software engineering interviews, and real-world problem solving.

Graphs, Representation (Adjacency Matrix/List)
https://www.youtube.com/watch?v=EQsZJCsNOSc
https://www.youtube.com/watch?v=8BhqEeW4a9w
https://www.youtube.com/watch?v=SwUNIiV6ePg
BFS (Breadth First Search)
https://www.youtube.com/watch?v=-2CI8oUbjO8
https://www.youtube.com/watch?v=pcKY4hjDrxk
DFS (Depth First Search)
https://www.youtube.com/watch?v=Qzf1a--rhp8
https://www.youtube.com/watch?v=AfSk24UTFS8
https://www.youtube.com/watch?v=8BhqEeW4a9w
https://www.youtube.com/watch?v=Qzf1a--rhp8
https://www.classcentral.com/course/youtube-depth-first-search-dfs-explained-algorithm-examples-and-code-129151

Topological Sort
https://www.youtube.com/watch?v=alF66XqpwJA
https://www.youtube.com/watch?v=ddTC4Zovtbc
Cycle Detection
https://www.youtube.com/watch?v=Y8bOGiLkBqg
https://www.youtube.com/watch?v=9twcmtQj4DU
Shortest Paths (Dijkstra, Bellman-Ford)
Dijkstra:
https://www.youtube.com/watch?v=pVfj6mxhdMw
https://www.youtube.com/watch?v=XB4MIexjvY0
Bellman-Ford:
https://www.youtube.com/watch?v=FtN3BYH2Zes
MST (Minimum Spanning Tree)
https://www.youtube.com/watch?v=ZtZaR7EcI5Y
https://www.youtube.com/watch?v=cplfcGZmX7I
Union Find (Disjoint Set Union)
https://www.youtube.com/watch?v=ibjEGG7ylHk
https://www.youtube.com/watch?v=VHRhJWacxis
Tarjan's Algorithm
https://www.youtube.com/watch?v=ZeDNSeilf-Y
https://www.youtube.com/watch?v=TyWtx7q2D7Y
Kosaraju's Algorithm
https://www.youtube.com/watch?v=RpgcYiky7uw
https://www.youtube.com/watch?v=V8qIqJxCioo