Graph Algorithms: Explanation, Pseudocode, and Use Cases

1 Connected Components Algorithm

1.1 Definition

The Connected Components algorithm identifies all the connected subgraphs in an undirected graph. A connected component is a set of vertices such that there is a path between any two vertices in this set, and which is connected to no additional vertices in the supergraph.

1.2 Use Cases

- Analyzing the structure of undirected networks (e.g., social networks).
- Image segmentation in computer vision.
- Network reliability and fault tolerance analysis.

1.3 Runtimes

- Time complexity: $O(V + E)$, where V is the number of vertices and E is the number of edges.
- Space complexity: $O(V)$ for the visited set and storage of components.

1.4 Pseudocode

```
function ConnectedComponents(Graph):
    visited = set()
    components = []

    for each vertex in Graph:
        if vertex not in visited:
            component = []
            DFS(Graph, vertex, visited, component)
            components.append(component)

    return components

function DFS(Graph, vertex, visited, component):
    visited.add(vertex)
    component.append(vertex)

    for each neighbor in Graph[vertex]:
        if neighbor not in visited:
            DFS(Graph, neighbor, visited, component)
```

Breadth-First Search (BFS)

Definition:
Breadth-First Search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts from a given source node and explores all neighboring nodes at the present depth before moving on to nodes at the next depth level.

Use Cases:

- Shortest path in unweighted graphs.
- Finding connected components in graphs.
- Web crawlers, social network analysis.

Pseudocode:

```
BFS(Graph, start):
    create queue Q
    mark start as visited and enqueue it into Q

    while Q is not empty:
        node = Q.dequeue()
        for each neighbor of node:
            if neighbor is not visited:
                mark neighbor as visited
                enqueue neighbor into Q
```

Depth-First Search (DFS)

Definition:
Depth-First Search (DFS) is an algorithm used to traverse or search through graphs or trees. It starts at a source node and explores as far as possible along each branch before backtracking.

Use Cases:

- Topological sorting in directed graphs.
- Detecting cycles in a graph.
- Solving maze-like puzzles.

Recursive DFS

Pseudocode (Recursive):

```
DFS(Graph, node, visited):
    if node not in visited:
        mark node as visited
        for each neighbor of node:
            if neighbor not visited:
                DFS(Graph, neighbor, visited)
```

Iterative DFS

Pseudocode (Iterative):

```
DFS_Iterative(Graph, start):
    create a stack S
    mark start as visited
    push start onto S

    while S is not empty:
        node = S.pop()
        for each neighbor of node:
            if neighbor is not visited:
                mark neighbor as visited
                push neighbor onto S
```

Explanation:

- In *Recursive DFS*, the algorithm uses the system's call stack to keep track of unexplored nodes.
- In *Iterative DFS*, an explicit stack (LIFO data structure) is used to simulate recursion. It is useful when recursion depth might exceed system limits.

A* Search Algorithm

Definition:
A* Search is an informed search algorithm that finds the shortest path from a start node to a target node by using heuristics to prioritize which paths to explore.

Use Cases:

- Pathfinding in AI (e.g., for games).
- Navigation systems (e.g., GPS).

Pseudocode:

```
A*(start, goal):
    openSet = priority queue containing start
    gScore[start] = 0
    fScore[start] = heuristic(start, goal)

    while openSet is not empty:
        current = node in openSet with lowest fScore

        if current == goal:
            return reconstruct_path(current)

        remove current from openSet

        for each neighbor of current:
            tentative_gScore = gScore[current] + dist(current, neighbor)

            if tentative_gScore < gScore[neighbor]:
                gScore[neighbor] = tentative_gScore
                fScore[neighbor] = gScore[neighbor] + heuristic(neighbor, goal)
                if neighbor not in openSet:
                    add neighbor to openSet
```

Dijkstra's Algorithm

Definition:
Dijkstra's algorithm finds the shortest path from a source node to all other nodes in a graph with non-negative weights. It uses a priority queue to explore nodes with the smallest known distances first.

Use Cases:

- Shortest path in road networks.
- Routing protocols.

Pseudocode:

```
Dijkstra(Graph, source):
    dist = {node: infinity for node in Graph}
    dist[source] = 0
    priority_queue = [(0, source)]

    while priority_queue is not empty:
        current_dist, current_node = priority_queue.pop()

        for neighbor, weight in Graph[current_node]:
            distance = current_dist + weight

            if distance < dist[neighbor]:
                dist[neighbor] = distance
                priority_queue.push((distance, neighbor))

    return dist
```

Bellman-Ford Algorithm

Definition:
The Bellman-Ford algorithm calculates the shortest paths from a single source node to all other nodes in a graph, even with negative weights. It iterates through all edges to relax them.

Use Cases:

- Handling negative weights in graphs.
- Detecting negative weight cycles.

Pseudocode:

```
BellmanFord(Graph, source):
    dist = {node: infinity for node in Graph}
    dist[source] = 0

    for i in range(len(Graph) - 1):
        for node in Graph:
            for neighbor, weight in Graph[node]:
                if dist[node] + weight < dist[neighbor]:
                    dist[neighbor] = dist[node] + weight

    for node in Graph:
        for neighbor, weight in Graph[node]:
            if dist[node] + weight < dist[neighbor]:
                raise NegativeCycleError()

    return dist
```

2 Kruskal's Algorithm

2.1 Definition

Kruskal's Algorithm is a greedy algorithm used for finding the Minimum Spanning Tree (MST) of a connected, undirected graph. It works by sorting the edges of the graph by weight and adding the smallest edge to the MST, provided it does not form a cycle.

2.2 Use Cases

- Network design (e.g., computer networks, electrical grids).
- Clustering data.
- Finding the least expensive way to connect points.

2.3 Runtimes

- Time complexity: $O(E \log E)$, where E is the number of edges.
- Space complexity: $O(V)$, where V is the number of vertices.

2.4 Pseudocode

Algorithm 1 Kruskal's Algorithm

```
function Kruskal(vertices, edges)
  sort(edges)  by weight
  MST = []
  Union-Find structure
for each edge in edges do
  if no cycle in MST with this edge then
    add edge to MST
  end if
end for
return  MST
```

3 Floyd-Warshall Algorithm

3.1 Definition

The Floyd-Warshall Algorithm is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. It works by repeatedly updating the shortest paths based on intermediate vertices.

3.2 Use Cases

- Finding shortest paths in dense graphs.
- Transitive closure of graphs.
- Network routing algorithms.

3.3 Runtimes

- Time complexity: $O(V^3)$, where V is the number of vertices.
- Space complexity: $O(V^2)$ for the distance matrix.

3.4 Pseudocode

Algorithm 2 Floyd-Warshall Algorithm

```
function FloydWarshall(graph)
for each vertex k do
  for each vertex i do
    for each vertex j do
      if dist[i][j] ¿ dist[i][k] + dist[k][j] then
        dist[i][j] = dist[i][k] + dist[k][j]
      end if
    end for
  end for
end for
```

4 Prim's Algorithm

4.1 Definition

Prim's Algorithm is a greedy algorithm that finds a Minimum Spanning Tree (MST) for a weighted undirected graph. It grows the MST one edge at a time, starting from an arbitrary vertex and adding the cheapest edge from the tree to a vertex not in the tree.

4.2 Use Cases

- Designing minimum-cost networks.
- Cluster analysis.
- Approximation algorithms for NP-hard problems.

4.3 Runtimes

- Time complexity: $O(E \log V)$, where E is the number of edges and V is the number of vertices.
- Space complexity: $O(V)$ for the priority queue.

4.4 Pseudocode

Algorithm 3 Prim's Algorithm

```
function Prim(graph, start)
    MST = []
    visited = set()
    minHeap = [(0, start)]
while minHeap is not empty do
    weight, node = heappop(minHeap)
    if node not in visited then
        visited.add(node)
        MST.add(node)
        for each neighbor in graph[node] do
            heappush(minHeap, (weight, neighbor))
        end for
    end if
end while
return MST
```

5 Strongly Connected Components (SCC)

5.1 Definition

The Strongly Connected Components algorithm identifies the maximal strongly connected subgraphs in a directed graph, where each vertex is reachable from every other vertex in that component.

5.2 Use Cases

- Analyzing the structure of directed networks.
- Finding clusters in social network analysis.

5.3 Runtimes

- Time complexity: $O(V + E)$, where V is the number of vertices and E is the number of edges.
- Space complexity: $O(V)$ for the stack and visited set.

5.4 Pseudocode

Algorithm 4 SCC Algorithm (Tarjan's Algorithm)

```
function SCC(graph)
    index = 0
    stack = []
    indices = [-1] * len(graph)
    lowlinks = [-1] * len(graph)
    onStack = [False] * len(graph)
for each v in graph do
    if indices[v] == -1 then
        strongconnect(v)
    end if
end for
function strongconnect(v)
    indices[v] = index
    lowlinks[v] = index
    index += 1
    stack.append(v)
    onStack[v] = True
for each neighbor in graph[v] do
    if indices[neighbor] == -1 then
        strongconnect(neighbor)
        lowlinks[v] = min(lowlinks[v], lowlinks[neighbor])
    else if onStack[neighbor] then
        lowlinks[v] = min(lowlinks[v], indices[neighbor])
    end if
end for
if lowlinks[v] == indices[v] then
        component = []
    while True do
        w = stack.pop()
        onStack[w] = False
        component.append(w)
        if w == v then
        end if
    end while
    output component
end if
```