Data Structure:

GRAPH

A graph is a data structure that consists of a set of vertices (also called nodes) and a set of edges (also called arcs) that connect pairs of vertices. Graphs are used to represent various types of real-world and abstract relationships, such as social networks, computer networks, transportation systems, and more.

Key Components of a Graph

Vertices (Nodes): These are the fundamental units of the graph. Each vertex can represent any entity such as a person, a city, or a computer.

Edges (Arcs): These are the connections between the vertices. Each edge connects a pair of vertices and can represent relationships or paths between them. Edges can be:

Directed: The edge has a direction, indicating a one-way relationship from one vertex to another (e.g., a Twitter follow).

Undirected: The edge has no direction, indicating a two-way relationship (e.g., a Facebook friendship).

Types of Graphs

Undirected Graph: A graph where edges have no direction. The edge (u, v) is identical to the edge (v, u).

Directed Graph (Digraph): A graph where edges have directions. The edge (u, v) is not the same as the edge (v, u).

Weighted Graph: A graph where each edge has an associated weight or cost. This is useful in scenarios like finding the shortest path in a network.

Unweighted Graph: A graph where all edges are considered to have the same weight or no weight.

Connected Graph: A graph where there is a path between every pair of vertices.

Disconnected Graph: A graph where at least one pair of vertices does not have a path connecting them.

Cyclic Graph: A graph that contains at least one cycle, which is a path that starts and ends at the same vertex.

Acyclic Graph: A graph with no cycles. A special type of directed acyclic graph is called a DAG (Directed Acyclic Graph).

Representations of Graphs

Adjacency Matrix: A 2D array of size VxV (where V is the number of vertices), used to represent a graph. If there is an edge from vertex i to vertex j, the element at row i and column j is 1 (or the weight of the edge); otherwise, it is 0.

Pros: Easy to implement and understand.

Cons: Requires O(V^2) space, even for sparse graphs.

Adjacency List: An array of lists. The array index represents a vertex, and each element in the list at index i represents the vertices adjacent to vertex i.

Pros: More space-efficient for sparse graphs.

Cons: More complex to implement than the adjacency matrix.

Edge List: A list of all edges in the graph, where each edge is represented as a pair (or triplet, if weighted) of vertices.

Pros: Simple and space-efficient for sparse graphs.

Cons: Not as efficient for certain operations, such as checking if there is an edge between two vertices.

Common Graph Algorithms

Breadth-First Search (BFS):

Explores the graph layer by layer, starting from a given vertex and exploring all its neighbors before moving to the next level. Used to find the shortest path in unweighted graphs.

Purpose: To traverse or search through all the nodes of a graph level by level.

Applicable To: Both directed and undirected graphs (unweighted).

Algorithm:

Initialize a queue and mark the starting node as visited.

Enqueue the starting node.

While the queue is not empty:

- Dequeue a node.
- Process the node (e.g., print it).
- For each unvisited adjacent node, mark it as visited and enqueue it.

Program:

```
from collections import deque

def bfs(graph, start):
  visited = set()
  queue = deque([start])
  visited.add(start)
  while queue:
    vertex = queue.popleft()
    print(vertex)  # Process the node
    for neighbor in graph[vertex]:
      if neighbor not in visited:
        visited.add(neighbor)
        queue.append(neighbor)
```

Depth-First Search (DFS):

Explores the graph by going as deep as possible along each branch before backtracking. Useful for pathfinding and topological sorting.

1. **Purpose:** To traverse or search through all the nodes of a graph by going as deep as possible along each branch before backtracking.

Applicable To: Both directed and undirected graphs.

Algorithm (Iterative):

1. Initialize a stack and mark the starting node as visited.
2. Push the starting node onto the stack.
3. While the stack is not empty:

- Pop a node.
- Process the node (e.g., print it).
- For each unvisited adjacent node, mark it as visited and push it onto the stack.

```
def dfs_iterative(graph, start):
  visited = set()
  stack = [start]
  visited.add(start)

  while stack:
    vertex = stack.pop()
    print(vertex)  # Process the node

    for neighbor in graph[vertex]:
      if neighbor not in visited:
        visited.add(neighbor)
        stack.append(neighbor)
```

Dijkstra's Algorithm:

Finds the shortest path from a starting vertex to all other vertices in a weighted graph.

Purpose: To find the shortest path from a starting node to all other nodes in a weighted graph (non-negative weights).

Applicable To: Directed and undirected weighted graphs (non-negative weights).

Algorithm:

1. Initialize a priority queue and distances.
2. Set the distance to the starting node to 0 and all others to infinity.
3. While the queue is not empty:
- Extract the node with the smallest distance.
- For each adjacent node, update its distance if a shorter path is found.

```
import heapq


def dijkstra(graph, start):
  distances = {vertex: float('infinity') for vertex in graph}
  distances[start] = 0
  priority_queue = [(0, start)]

  while priority_queue:
    current_distance, current_vertex = heapq.heappop(priority_queue)

    if current_distance > distances[current_vertex]:
      continue

    for neighbor, weight in graph[current_vertex].items():
      distance = current_distance + weight

      if distance < distances[neighbor]:
        distances[neighbor] = distance
        heapq.heappush(priority_queue, (distance, neighbor))

  return distances
```

Bellman-Ford Algorithm:

Also finds the shortest paths from a single source vertex to all other vertices but can handle graphs with negative weights.

Purpose: To find the shortest path from a starting node to all other nodes in a graph, including graphs with negative weights.

Applicable To: Directed and undirected weighted graphs (can handle negative weights).

Algorithm:

1. Initialize distances.
2. Relax all edges up to |V|-1 times (where |V| is the number of vertices).
3. Check for negative-weight cycles.

```
def bellman_ford(graph, start):
  distances = {vertex: float('infinity') for vertex in graph}
  distances[start] = 0

  for _ in range(len(graph) - 1):
    for vertex in graph:
      for neighbor, weight in graph[vertex].items():
        if distances[vertex] + weight < distances[neighbor]:
          distances[neighbor] = distances[vertex] + weight

  # Check for negative-weight cycles
  for vertex in graph:
    for neighbor, weight in graph[vertex].items():
      if distances[vertex] + weight < distances[neighbor]:
        raise ValueError("Graph contains a negative-weight cycle")
```

```
    return distances
```

Prim's and Kruskal's Algorithms:

Used to find the Minimum Spanning Tree (MST) of a graph, which is a subset of the edges that connects all vertices with the minimum total edge weight.

Prim's Algorithm

Purpose: To find the Minimum Spanning Tree (MST) for a connected, undirected graph with weighted edges.

Applicable To: Connected, undirected weighted graphs.

Algorithm:

1. Initialize a priority queue.
2. Start with an arbitrary node and mark it as visited.
3. While the queue is not empty, add the smallest edge that connects a visited node to an unvisited node to the MST.

```
import heapq


def prim(graph, start):
    mst = []
    visited = set([start])
    edges = [(weight, start, to) for to, weight in graph[start].items()]
    heapq.heapify(edges)

    while edges:
        weight, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.append((frm, to, weight))
```

```
        for next_to, next_weight in graph[to].items():
            if next_to not in visited:
                heapq.heappush(edges, (next_weight, to, next_to))

    return mst
```

Kruskal's Algorithm

Purpose: To find the Minimum Spanning Tree (MST) for a connected, undirected graph with weighted edges.

Applicable To: Connected, undirected weighted graphs.

Algorithm:

1. Sort all edges by weight.
2. Initialize a disjoint-set data structure.
3. Add edges to the MST, ensuring no cycles are formed, until the MST contains |V|-1 edges.

```
class DisjointSet:
    def __init__(self, vertices):
        self.parent = {v: v for v in vertices}
        self.rank = {v: 0 for v in vertices}

    def find(self, vertex):
        if self.parent[vertex] != vertex:
            self.parent[vertex] = self.find(self.parent[vertex])
        return self.parent[vertex]

    def union(self, u, v):
        root_u = self.find(u)
```

```
        root_v = self.find(v)

        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            else:
                self.parent[root_u] = root_v
                if self.rank[root_u] == self.rank[root_v]:
                    self.rank[root_v] += 1

def kruskal(graph):
    mst = []
    edges = sorted((weight, u, v) for u in graph for v, weight in graph[u].items())
    ds = DisjointSet(graph.keys())

    for weight, u, v in edges:
        if ds.find(u) != ds.find(v):
            ds.union(u, v)
            mst.append((u, v, weight))

    return mst
```