Graph Data Structure

Created by	j jubink
Created time	@February 26, 2025 10:44 PM
Tags	

🌟 Beginner Level - Graph Data Structure Practice Questions

Let's start with **beginner-level** Graph problems with **detailed algorithms and Python solutions.** 🚀

1️⃣ Graph Representation (Adjacency List & Matrix)

Problem Statement

Implement a graph using both **Adjacency List** and **Adjacency Matrix** representations.

Algorithm

1. **Adjacency List**
    - Use a dictionary where keys are nodes, and values are lists of adjacent nodes.
    - Space-efficient for sparse graphs.
2. **Adjacency Matrix**
    - Use a 2D list (`N x N` matrix) where `graph[i][j] = 1` if there is an edge between `i` and `j`, else `0`.
    - Efficient for dense graphs.

Python Code

```python
python
CopyEdit
class GraphAdjList:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)  # For undirected graph

    def display(self):
        for node in self.graph:
            print(f"{node}: {self.graph[node]}")

class GraphAdjMatrix:
    def __init__(self, size):
        self.size = size
        self.matrix = [[0] * size for _ in range(size)]

    def add_edge(self, u, v):
        self.matrix[u][v] = 1
        self.matrix[v][u] = 1  # For undirected graph

    def display(self):
        for row in self.matrix:
            print(row)

# Usage
g_list = GraphAdjList()
g_list.add_edge(0, 1)
g_list.add_edge(0, 2)
```

```
g_list.add_edge(1, 3)
print("Adjacency List Representation:")
g_list.display()

g_matrix = GraphAdjMatrix(4)
g_matrix.add_edge(0, 1)
g_matrix.add_edge(0, 2)
g_matrix.add_edge(1, 3)
print("\nAdjacency Matrix Representation:")
g_matrix.display()
```

2️⃣ Graph Traversal (BFS & DFS)

Problem Statement

Implement **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** for graph traversal.

Algorithm

1. **Breadth-First Search (BFS)**
    - Use a **queue** (`FIFO`).
    - Start from a node, visit all neighbors before moving deeper.
2. **Depth-First Search (DFS)**
    - Use **recursion** or a **stack (`LIFO`)**.
    - Start from a node, go as deep as possible before backtracking.

Python Code

```
python
CopyEdit
from collections import deque
```

```python
class GraphTraversal:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

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

    def dfs(self, start, visited=None):
        if visited is None:
            visited = set()
        if start not in visited:
            print(start, end=" ")
            visited.add(start)
            for neighbor in self.graph[start]:
                self.dfs(neighbor, visited)

# Usage
g = GraphTraversal()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
```

```
g.add_edge(2, 4)

print("\nBFS Traversal:")
g.bfs(0)  # Expected Output: 0 1 2 3 4

print("\nDFS Traversal:")
g.dfs(0)  # Expected Output: 0 1 3 2 4
```

Implementing BFS & DFS using Adjacency Matrix

Here's how we can modify your code to use an **Adjacency Matrix** instead of an adjacency list.

Python Code

```
python
CopyEdit
from collections import deque

class GraphMatrix:
    def __init__(self, size):
        self.size = size  # Number of nodes
        self.matrix = [[0] * size for _ in range(size)]  # Create size x size matrix

    def add_edge(self, u, v):
        self.matrix[u][v] = 1
        self.matrix[v][u] = 1  # Remove this line for directed graphs

    def bfs(self, start):
        visited = set()
        queue = deque([start])

        while queue:
            node = queue.popleft()
```

```
            if node not in visited:
                print(node, end=" ")
                visited.add(node)
                for neighbor in range(self.size):
                    if self.matrix[node][neighbor] == 1 and neighbor not in visited:
                        queue.append(neighbor)

    def dfs(self, start, visited=None):
        if visited is None:
            visited = set()
        if start not in visited:
            print(start, end=" ")
            visited.add(start)
            for neighbor in range(self.size):
                if self.matrix[start][neighbor] == 1 and neighbor not in visited:
                    self.dfs(neighbor, visited)

# Usage
g = GraphMatrix(5)
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(2, 4)

print("\nBFS Traversal:")
g.bfs(0)  # Expected Output: 0 1 2 3 4

print("\nDFS Traversal:")
g.dfs(0)  # Expected Output: 0 1 3 2 4
```

3️⃣ Detect Cycle in an Undirected Graph

Problem Statement

Detect if a **cycle** exists in an **undirected graph** using DFS.

Algorithm

1. **Use DFS with a parent check.**
2. **If a visited node is reached again (not the parent), a cycle exists.**

Python Code

```
python
CopyEdit
class GraphCycleDetection:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

    def has_cycle_util(self, node, visited, parent):
        visited.add(node)
        for neighbor in self.graph[node]:
            if neighbor not in visited:
                if self.has_cycle_util(neighbor, visited, node):
                    return True
            elif neighbor != parent:
                return True
        return False

    def has_cycle(self):
        visited = set()
        for node in self.graph:
            if node not in visited:
```

```
            if self.has_cycle_util(node, visited, -1):
                return True
        return False

# Usage
g = GraphCycleDetection()
g.add_edge(0, 1)
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 0)  # This creates a cycle

print("\nCycle in Graph:", g.has_cycle())  # Output: True
```

4️⃣ Count Connected Components in an Undirected Graph

Problem Statement

Find the number of **connected components** in an undirected graph.

Algorithm

1. **Use DFS or BFS** to traverse the graph.
2. **Each DFS/BFS call** represents a new component.

Python Code

```
python
CopyEdit
class GraphConnectedComponents:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
```

```
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

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

    def count_components(self):
        visited = set()
        count = 0
        for node in self.graph:
            if node not in visited:
                self.dfs(node, visited)
                count += 1
        return count

# Usage
g = GraphConnectedComponents()
g.add_edge(0, 1)
g.add_edge(1, 2)
g.add_edge(3, 4)  # Separate component

print("\nNumber of Connected Components:", g.count_components())  # Outp
ut: 2
```

🌟 Intermediate Level - Graph Data Structure Practice Questions

Now, let's move on to **intermediate-level** Graph problems with **detailed algorithms and Python solutions**. 🚀

1️⃣ Find the Shortest Path in an Unweighted Graph (Using BFS)

Problem Statement

Given an **unweighted graph**, find the **shortest path** from a source node to all other nodes.

Algorithm (BFS-based)

1. Use **BFS** because it finds the shortest path in **O(V + E)** time.
2. Maintain a **distance array** initialized to infinity (`float('inf')`).
3. Start from the **source node**, set `distance[source] = 0`, and use a **queue**.
4. For each dequeued node, update its neighbors' distance (`dist[neighbor] = dist[node] + 1`).

Python Code

```
python
CopyEdit
from collections import deque

class GraphShortestPath:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
```

```
        self.graph[v].append(u)

    def shortest_path(self, source):
        distance = {node: float('inf') for node in self.graph}
        distance[source] = 0
        queue = deque([source])

        while queue:
            node = queue.popleft()
            for neighbor in self.graph[node]:
                if distance[neighbor] == float('inf'):  # Not visited
                    distance[neighbor] = distance[node] + 1
                    queue.append(neighbor)

        return distance

# Usage
g = GraphShortestPath()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(2, 4)

print("Shortest Paths from Node 0:", g.shortest_path(0))
# Expected Output: {0: 0, 1: 1, 2: 1, 3: 2, 4: 2}
```

2️⃣ Dijkstra’s Algorithm (Shortest Path in Weighted Graph)

Problem Statement

Given a **weighted graph**, find the shortest path from the source node to all other nodes.

Algorithm (Using Min-Heap / Priority Queue)

1. Initialize a **distance dictionary** (`float('inf')` for all nodes except source, which is `0`).
2. Use a **Min-Heap (Priority Queue)** to always expand the node with the smallest known distance.
3. For each **neighbor**, update its distance if a **shorter path** is found.
4. Continue until all nodes are processed.

Python Code

```
python
CopyEdit
import heapq

class GraphDijkstra:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v, weight):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append((v, weight))
        self.graph[v].append((u, weight))  # Undirected graph

    def dijkstra(self, source):
        distance = {node: float('inf') for node in self.graph}
        distance[source] = 0
        min_heap = [(0, source)]  # (cost, node)

        while min_heap:
            curr_dist, node = heapq.heappop(min_heap)
```

```
            if curr_dist > distance[node]:
                continue

            for neighbor, weight in self.graph[node]:
                new_dist = curr_dist + weight
                if new_dist < distance[neighbor]:
                    distance[neighbor] = new_dist
                    heapq.heappush(min_heap, (new_dist, neighbor))

        return distance

# Usage
g = GraphDijkstra()
g.add_edge(0, 1, 4)
g.add_edge(0, 2, 1)
g.add_edge(2, 1, 2)
g.add_edge(1, 3, 1)
g.add_edge(2, 3, 5)

print("Shortest Paths from Node 0:", g.dijkstra(0))
# Expected Output: {0: 0, 2: 1, 1: 3, 3: 4}
```

3️⃣ Detect a Cycle in a Directed Graph (Using DFS)

Problem Statement

Detect if a **cycle** exists in a **directed graph**.

Algorithm

1. Use **DFS with recursion stack** to detect back edges.
2. Maintain a `visited` set and a `rec_stack` (recursive stack).
3. If a node is encountered that is already in `rec_stack`, a **cycle exists**.

Python Code

```
python
CopyEdit
class GraphCycleDirected:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append(v)

    def has_cycle_util(self, node, visited, rec_stack):
        visited.add(node)
        rec_stack.add(node)

        for neighbor in self.graph.get(node, []):
            if neighbor not in visited:
                if self.has_cycle_util(neighbor, visited, rec_stack):
                    return True
            elif neighbor in rec_stack:
                return True

        rec_stack.remove(node)
        return False

    def has_cycle(self):
        visited = set()
        rec_stack = set()
        for node in self.graph:
            if node not in visited:
                if self.has_cycle_util(node, visited, rec_stack):
                    return True
        return False
```

```
# Usage
g = GraphCycleDirected()
g.add_edge(0, 1)
g.add_edge(1, 2)
g.add_edge(2, 0)  # This creates a cycle

print("\nCycle in Directed Graph:", g.has_cycle())  # Output: True
```

4️⃣ Topological Sorting (Kahn's Algorithm - BFS)

Problem Statement

Given a **DAG (Directed Acyclic Graph)**, perform **Topological Sorting**.

Algorithm (Using Kahn's Algorithm - BFS)

1. **Calculate in-degree** of all nodes.
2. **Start with nodes** that have `in-degree = 0`.
3. **Process nodes** in a queue, reduce in-degree of neighbors.
4. **If all nodes are processed,** return topological order.

Python Code

```
python
CopyEdit
from collections import deque

class GraphTopologicalSort:
    def __init__(self):
        self.graph = {}
        self.in_degree = {}

    def add_edge(self, u, v):
```

```
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)

        self.in_degree[v] = self.in_degree.get(v, 0) + 1
        self.in_degree.setdefault(u, 0)

    def topological_sort(self):
        queue = deque([node for node in self.in_degree if self.in_degree[node] =
= 0])
        result = []

        while queue:
            node = queue.popleft()
            result.append(node)
            for neighbor in self.graph[node]:
                self.in_degree[neighbor] -= 1
                if self.in_degree[neighbor] == 0:
                    queue.append(neighbor)

        return result if len(result) == len(self.graph) else "Cycle Detected (Not a
DAG)"

# Usage
g = GraphTopologicalSort()
g.add_edge(5, 2)
g.add_edge(5, 0)
g.add_edge(4, 0)
g.add_edge(4, 1)
g.add_edge(2, 3)
g.add_edge(3, 1)
```

```
print("\nTopological Sort Order:", g.topological_sort())
# Expected Output: [5, 4, 2, 3, 1, 0] (Order may vary)
```

🌟 Advanced Level - Graph Data Structure Practice Questions

Now, let's tackle **advanced-level** Graph problems with **detailed algorithms and Python solutions.** 🚀

1️⃣ Floyd-Warshall Algorithm (All-Pairs Shortest Path)

Problem Statement

Given a **weighted directed graph**, find the **shortest path** between all pairs of nodes.

Algorithm (Dynamic Programming)

1. Create a **distance matrix** where `dist[i][j]` stores the shortest path from node `i` to `j`.
2. Initialize the matrix:
    - `dist[i][i] = 0` (Distance to itself is 0).
    - `dist[i][j] = weight(i, j)` if there is a direct edge.
    - `dist[i][j] = ∞` (or large value) if no direct edge exists.
3. Use **three nested loops** to iteratively update distances:
    - `dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])`
    - This checks if an **intermediate node** `k` provides a shorter path.
4. Time Complexity: **O(V^3)**.

Python Code

```python
python
CopyEdit
class GraphFloydWarshall:
    def __init__(self, vertices):
        self.V = vertices
        self.INF = float('inf')
        self.graph = [[self.INF] * vertices for _ in range(vertices)]

    def add_edge(self, u, v, weight):
        self.graph[u][v] = weight

    def floyd_warshall(self):
        dist = [row[:] for row in self.graph]

        for k in range(self.V):
            for i in range(self.V):
                for j in range(self.V):
                    if dist[i][k] != self.INF and dist[k][j] != self.INF:
                        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

        return dist

# Usage
g = GraphFloydWarshall(4)
g.add_edge(0, 1, 3)
g.add_edge(0, 2, 10)
g.add_edge(1, 2, 2)
g.add_edge(2, 3, 1)

result = g.floyd_warshall()
print("\nAll-Pairs Shortest Paths:")
for row in result:
    print(row)
```

2️⃣ Bellman-Ford Algorithm (Single-Source Shortest Path with Negative Weights)

Problem Statement

Find the **shortest path from a source node to all other nodes**, even if negative-weight edges exist.

Algorithm

1. Initialize `distance` array, where `distance[source] = 0` and all others are `∞`.
2. **Relax all edges** `V-1` times (as the longest shortest path has at most `V-1` edges).
3. After `V-1` iterations, if any edge can still be relaxed, a **negative-weight cycle** exists.
4. Time Complexity: **O(VE)**.

Python Code

```
python
CopyEdit
class GraphBellmanFord:
    def __init__(self, vertices):
        self.V = vertices
        self.edges = []

    def add_edge(self, u, v, weight):
        self.edges.append((u, v, weight))

    def bellman_ford(self, source):
        distance = {i: float('inf') for i in range(self.V)}
        distance[source] = 0

        for _ in range(self.V - 1):
            for u, v, w in self.edges:
```

```
            if distance[u] != float('inf') and distance[u] + w < distance[v]:
                distance[v] = distance[u] + w

        for u, v, w in self.edges:
            if distance[u] != float('inf') and distance[u] + w < distance[v]:
                return "Negative Weight Cycle Detected"

        return distance

# Usage
g = GraphBellmanFord(5)
g.add_edge(0, 1, -1)
g.add_edge(1, 2, 4)
g.add_edge(2, 3, -3)
g.add_edge(3, 4, 2)
g.add_edge(4, 1, 1)

print("\nShortest Paths from Node 0:", g.bellman_ford(0))
```

3️⃣ Find Bridges in a Graph (Tarjan's Algorithm)

Problem Statement

Find all **bridges** in a graph. A **bridge** is an edge whose removal increases the number of connected components.

Algorithm (DFS-Based Tarjan's Algorithm)

1. Perform **DFS traversal**.
2. Maintain:
    - `tin[]`: The **discovery time** of each node.
    - `low[]`: The **earliest reachable ancestor** of the node.
3. If `low[neighbor] > tin[node]`, the edge `(node, neighbor)` is a **bridge**.

4. Time Complexity: **O(V + E)**.

Python Code

```python
python
CopyEdit
class GraphBridges:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = {i: [] for i in range(vertices)}
        self.timer = 0
        self.bridges = []

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs(self, node, parent, visited, tin, low):
        visited[node] = True
        tin[node] = low[node] = self.timer
        self.timer += 1

        for neighbor in self.graph[node]:
            if neighbor == parent:
                continue
            if visited[neighbor]:
                low[node] = min(low[node], tin[neighbor])
            else:
                self.dfs(neighbor, node, visited, tin, low)
                low[node] = min(low[node], low[neighbor])
                if low[neighbor] > tin[node]:
                    self.bridges.append((node, neighbor))

    def find_bridges(self):
        visited = [False] * self.V
```

```
        tin = [-1] * self.V
        low = [-1] * self.V

        for i in range(self.V):
            if not visited[i]:
                self.dfs(i, -1, visited, tin, low)

        return self.bridges

# Usage
g = GraphBridges(5)
g.add_edge(0, 1)
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 4)

print("\nBridges in the Graph:", g.find_bridges())
```

4️⃣ Find Strongly Connected Components (Kosaraju's Algorithm)

Problem Statement

Find all **strongly connected components (SCCs)** in a directed graph.

Algorithm (Kosaraju's Algorithm)

1. **First DFS**: Store nodes in **postorder (stack)**.
2. **Reverse Graph**: Reverse all edges.
3. **Second DFS**: Pop nodes from stack and perform DFS in **reversed graph** to identify SCCs.
4. Time Complexity: **O(V + E)**.

Python Code

```
python
CopyEdit
class GraphSCC:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = {i: [] for i in range(vertices)}

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs(self, node, visited, stack):
        visited[node] = True
        for neighbor in self.graph[node]:
            if not visited[neighbor]:
                self.dfs(neighbor, visited, stack)
        stack.append(node)

    def reverse_graph(self):
        reversed_graph = {i: [] for i in range(self.V)}
        for node in self.graph:
            for neighbor in self.graph[node]:
                reversed_graph[neighbor].append(node)
        return reversed_graph

    def find_sccs(self):
        stack, visited = [], [False] * self.V
        for i in range(self.V):
            if not visited[i]:
                self.dfs(i, visited, stack)

        reversed_graph = self.reverse_graph()
        visited = [False] * self.V
        sccs = []
```

```
        while stack:
            node = stack.pop()
            if not visited[node]:
                scc, dfs_stack = [], [node]
                while dfs_stack:
                    v = dfs_stack.pop()
                    if not visited[v]:
                        visited[v] = True
                        scc.append(v)
                        dfs_stack.extend(reversed_graph[v])
                sccs.append(scc)

        return sccs

# Usage
g = GraphSCC(5)
g.add_edge(0, 2)
g.add_edge(2, 1)
g.add_edge(1, 0)
g.add_edge(1, 3)
g.add_edge(3, 4)

print("\nStrongly Connected Components:", g.find_sccs())
```