Lab 15

Introduction

- Define graphs and their key components (vertices, edges, directed/undirected)

Here's a detailed definition of graphs and their key components:

Graphs:

- A graph is a non-linear data structure that represents relationships between entities.
- It consists of a set of vertices (also called nodes or points) and a set of edges (also called links or lines) that connect pairs of vertices.
- Graphs are used to model various real-world scenarios, such as social networks, maps, transportation systems, computer networks, and many more.

Key Components:

1. Vertices (Nodes):

    - Represent entities or objects in the graph.
    - Can be labeled with numbers, letters, or any other meaningful identifiers.
    - Examples: people in a social network, cities on a map, web pages on the internet.
2. Edges (Links):

    - Represent relationships or connections between vertices.

    - Can be either directed or undirected.
    - May have weights associated with them to represent additional information (e.g., distance, cost, capacity).

Types of Graphs:

1. Directed Graphs:

    - Edges have a direction, indicating a one-way relationship from one vertex to another.
    - Used to model situations where relationships have a specific flow or hierarchy.
    - Example: A flowchart, a citation network.
2. Undirected Graphs:

    - Edges do not have a direction, representing bidirectional relationships between vertices.
    - Used to model situations where connections are mutual or symmetric.
    - Example: A friendship network, a road map.

Additional Concepts:

- Adjacency: Two vertices are adjacent if they are connected by an edge.
- Degree of a Vertex: The number of edges incident to a vertex (either incoming or outgoing in directed graphs).
- Path: A sequence of vertices connected by edges.
- Cycle: A path that starts and ends at the same vertex.

- Explain real-world applications of graphs (social networks, maps, web pages)

Here are some real-world applications of graphs:

1. Social Networks:

- Graphs are the underlying structure of social networks like Facebook, Twitter, and LinkedIn.
- Vertices represent people, and edges represent friendships or connections between them.
- Graph algorithms are used for:
  - Friend recommendations
  - Identifying influential users
  - Analyzing community structure
  - Detecting fake accounts or bots
-

2. Maps and Navigation:

- Maps can be modeled as graphs, with vertices representing locations (cities, intersections) and edges representing roads or paths.
- Graph algorithms are used for:
  - Finding shortest paths between locations
  - Route planning for vehicles or pedestrians
  - Traffic analysis and optimization
  - Location-based services (e.g., finding nearby restaurants or points of interest)
-

3. Web Pages and the Internet:

- The internet can be viewed as a massive graph, with web pages as vertices and hyperlinks as edges.
- Graph algorithms are used for:
  - Web search engines (e.g., Google's PageRank algorithm)
  - Recommending related content or products

    - Analyzing website structure and link popularity
-

4. Transportation Networks:

- Graphs are used to model transportation systems like airlines, railways, and roadways.
- Vertices represent stations or airports, and edges represent routes or connections.
- Graph algorithms are used for:
    - Scheduling flights or trains
    - Optimizing routes for delivery services
    - Planning infrastructure expansion
-

5. Biological Networks:

- Graphs are used to represent relationships between biological entities, such as protein-protein interactions, gene regulatory networks, and metabolic pathways.
- Graph algorithms are used for:
    - Identifying key genes or proteins involved in diseases
    - Drug discovery and development
    - Understanding biological processes
-

6. Recommendation Systems:

- Graphs are used to model user preferences and item relationships in recommendation systems.
- Vertices represent users and items, and edges represent interactions (e.g., ratings, purchases).

- Graph algorithms are used for:
    - Recommending movies, music, books, or products
    - Personalizing user experiences
-

7. Task Scheduling and Resource Allocation:

- Graphs are used to model tasks and dependencies in project management and scheduling.
- Vertices represent tasks, and edges represent dependencies between them.
- Graph algorithms are used for:
    - Scheduling tasks efficiently
    - Allocating resources to tasks
    - Optimizing project timelines
-

- Discuss graph representation methods (adjacency list, adjacency matrix)

Here's a detailed discussion of graph representation methods:

Graphs can be represented in computer memory using two primary methods:

1. Adjacency List:

- Structure:

    - An array of lists, where each index in the array represents a vertex.
    - The list at each index stores the vertices that are directly connected to that vertex (its neighbors).
-

- Pros:
    - Efficient for sparse graphs (graphs with relatively few edges).
    - Space-efficient, as it only stores edges that actually exist.
    - Fast for adding and removing edges.
-
- Cons:
    - Less efficient for dense graphs (graphs with many edges).
    - Slower for checking if an edge exists between two vertices.
-
- Example:
- Python

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

-
-

2. Adjacency Matrix:

- Structure:
    - A 2D matrix where rows and columns represent vertices.

    - The value at matrix[i][j] represents the presence or weight of an edge between vertices i and j.
-
- Pros:
    - Fast for checking if an edge exists between two vertices (constant time).
    - Can easily represent weighted graphs.
-
- Cons:
    - Space-inefficient for sparse graphs, as it stores all possible edges even if they don't exist.
    - Less efficient for adding and removing edges (requires updating multiple matrix elements).
-
- Example:
- Python

```
graph = [
    [0, 1, 1, 0],
    [0, 0, 1, 1],
    [1, 1, 0, 1],
    [0, 1, 1, 0]
]
```

-
-

Choosing the Right Representation:

- Adjacency list is generally preferred for sparse graphs due to its space efficiency and faster edge operations.
- Adjacency matrix is preferred for dense graphs or when quick edge existence checks are crucial.
- The choice also depends on the specific algorithms you plan to use and the nature of the graph operations you'll be performing most frequently.

Lab 1: Creating Graphs in Python

- Use Python's built-in `dict` or `defaultdict` to create adjacency lists

**Here's how to create adjacency lists using Python's built-in `dict` and `defaultdict`:

1. Using `dict`:

```python
graph = {}



# Add vertices

graph[0] = [1, 2]  # Vertex 0 is connected to vertices 1 and 2

graph[1] = [2, 3]

graph[2] = [0, 3]

graph[3] = [1]
```

2. Using `defaultdict` (more convenient for adding edges):

```python
```

```
from collections import defaultdict

graph = defaultdict(list)

# Add edges directly without checking for vertex existence
graph[0].append(1)
graph[0].append(2)
graph[1].append(2)
graph[1].append(3)
graph[2].append(0)
graph[2].append(3)
graph[3].append(1)
```

Explanation:

- `dict`:
    - Stores key-value pairs, where keys represent vertices and values represent lists of adjacent vertices.
    - You need to create the list for a vertex before adding edges to it.
-
- `defaultdict`:
    - A subclass of `dict` that automatically creates a default value for a key when it's accessed for the first time.

    - Here, the default value is an empty list, so you can directly append edges without checking if the vertex exists in the dictionary.
-

Example Usage:

```python
# Print neighbors of vertex 2

print(graph[2])  # Output: [0, 3]



# Check if there's an edge between 1 and 3

if 3 in graph[1]:

    print("Edge exists between 1 and 3")
```

- Implement a Graph class using adjacency lists

Here's a Python implementation of a Graph class using adjacency lists:

```python
class
Graph:



def
__init__(self):

        self.adjacency_list = {}



    def
```

```
add_vertex(self, vertex):


if vertex not
in self.adjacency_list:
            self.adjacency_list[vertex] = []


    def
add_edge(self, vertex1, vertex2):
        self.adjacency_list[vertex1].append(vertex2)
        # For undirected graphs, add the edge in both directions
        if not self.is_directed():
            self.adjacency_list[vertex2].append(vertex1)


    def print_graph(self):
        for vertex, neighbors in self.adjacency_list.items():
            print(f"{vertex} -> {neighbors}")


    def is_directed(self):
        return False  # Override for directed graphs
```

Explanation:

- `__init__()`: Initializes an empty adjacency list to store the graph's vertices and edges.

- `add_vertex()`: Adds a new vertex to the graph.
- `add_edge()`: Adds an edge between two vertices. For undirected graphs, it adds the edge in both directions.
- `print_graph()`: Prints the graph's adjacency list for visualization.
- `is_directed()`: A placeholder method to indicate whether the graph is directed or undirected. Override this method in subclasses for directed graphs.

Example Usage:

Python
```python
graph = Graph()

graph.add_vertex(0)

graph.add_vertex(1)

graph.add_vertex(2)

graph.add_edge(0, 1)

graph.add_edge(0, 2)

graph.add_edge(1, 2)

graph.print_graph()
```

Output:

```
0 -> [1, 2]

1 -> [0, 2]

2 -> [0, 1]
```

- Practice creating simple graphs with vertices and edges

Here are some examples to practice creating simple graphs with vertices and edges:

1. Simple Undirected Graph:

```python
graph = Graph()  # Create an undirected graph instance



# Add vertices

graph.add_vertex('A')

graph.add_vertex('B')

graph.add_vertex('C')

graph.add_vertex('D')



# Add edges

graph.add_edge('A', 'B')

graph.add_edge('A', 'C')

graph.add_edge('B', 'D')

graph.add_edge('C', 'D')



graph.print_graph()
```

Output:

```
A -> ['B', 'C']

B -> ['A', 'D']

C -> ['A', 'D']

D -> ['B', 'C']
```

2. Directed Graph:

```python
class DirectedGraph(Graph):

    def is_directed(self):

        return True



# Create a directed graph instance

directed_graph = DirectedGraph()



# Add vertices and edges (direction matters)

directed_graph.add_vertex('X')

directed_graph.add_vertex('Y')

directed_graph.add_vertex('Z')

directed_graph.add_edge('X', 'Y')

directed_graph.add_edge('Y', 'Z')

directed_graph.add_edge('Z', 'X')
directed_graph.print_graph()
```

Output:

```
X -> ['Y']

Y -> ['Z']

Z -> ['X']
```

3. Graph with Weighted Edges:

```python
class WeightedGraph(Graph):

    def add_edge(self, vertex1, vertex2, weight):

        self.adjacency_list[vertex1].append((vertex2, weight))

        if not self.is_directed():

            self.adjacency_list[vertex2].append((vertex1, weight))



# Create a weighted graph instance

weighted_graph = WeightedGraph()



# Add vertices and weighted edges

weighted_graph.add_vertex('1')

weighted_graph.add_vertex('2')
weighted_graph.add_vertex('3')

weighted_graph.add_edge('1', '2', 5)

weighted_graph.add_edge('1', '3', 2)

weighted_graph.add_edge('2', '3', 1)


# Print the graph with weights

for vertex, neighbors in weighted_graph.adjacency_list.items():

    print(f"{vertex} -> {[(neighbor, weight) for neighbor, weight in 
neighbors]}")
```

Output:

```
1 -> [('2', 5), ('3', 2)]

2 -> [('1', 5), ('3', 1)]

3 -> [('1', 2), ('2', 1)]
```

Lab 2: Graph Traversal Algorithms

- Implement Depth-First Search (DFS) recursively and iteratively

Here's the Python implementation of Depth-First Search (DFS) both recursively and iteratively:

Recursive DFS:

Python

```
def dfs_recursive(graph, start, visited=None):

    if visited is None:

        visited = set()

    visited.add(start)

    print(start, end=" ")  # Print the vertex as it's visited



    for neighbor in graph[start]:

        if neighbor not in visited:

            dfs_recursive(graph, neighbor, visited)
```

Explanation:

1. Base Case: If a vertex is already visited, don't explore it again.
2. Recursive Step:
   - Mark the current vertex as visited.
   - Print it (or perform any desired action).
   - Recursively explore each unvisited neighbor.
3.

Iterative DFS:

```
Python
def
dfs_iterative(graph, start):

    visited = set()

    stack = [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            print(vertex, end=" ")
            stack.extend(graph[vertex])  # Add neighbors to the stack in reverse order
```

Explanation:

1. Initialization: Create a set to track visited vertices and a stack to manage the traversal.
2. Exploration:
    - Pop a vertex from the stack.
    - If it's not visited, mark it as visited and print it.
    - Add its neighbors to the stack to explore them later.

Key Points:

- Both implementations explore vertices as deeply as possible before backtracking.
- Recursive DFS uses the call stack for exploration, while iterative DFS uses an explicit stack.
- Choose the implementation based on your preference and the problem's requirements.

- Implement Breadth-First Search (BFS) using a queue

Here's the Python implementation of Breadth-First Search (BFS) using a queue:

Python
```python
def bfs(graph, start):

    visited = set()

    queue = [start]



    while queue:

        vertex = queue.pop(0)  # Dequeue from the front

        if vertex not in visited:

            visited.add(vertex)

            print(vertex, end=" ")

            queue.extend(graph[vertex])  # Add neighbors to the queue
```

Explanation:

1. Initialization: Create a set to track visited vertices and a queue to manage the traversal.
2. Exploration:
    - Dequeue a vertex from the front of the queue.
    - If it's not visited, mark it as visited and print it.
    - Enqueue its neighbors to explore them later, ensuring level-by-level traversal.
3.

Visual Representation of BFS:

Key Points:

- BFS explores vertices level by level, starting from the source vertex.
- It uses a queue to maintain the order of vertices to explore.
- It's often used to find the shortest paths between vertices in unweighted graphs.

- Visualize graph traversals using diagrams or Python libraries

Here are visual examples of graph traversals and Python libraries for visualization:

1. Visual Representations of DFS and BFS:

DFS:

BFS:

2. Python Libraries for Graph Visualization:

- NetworkX: A powerful library for creating, manipulating, and analyzing graphs.
- matplotlib: A general-purpose plotting library that can be used with NetworkX to create graph visualizations.

Example using NetworkX and matplotlib:

Python
```
import networkx as nx

import matplotlib.pyplot as plt



# Create a graph
```

```
G = nx.Graph()

G.add_edges_from([('A', 'B'), ('A', 'C'), ('B', 'D'), ('C', 'D'), ('B', 'E'), ('D', 'F')])


# Perform DFS or BFS traversal (e.g., using the implementations provided earlier)

# ...


# Visualize the graph with highlighted traversal paths

nx.draw(G, with_labels=True, node_color='lightblue', edge_color='gray')

# Highlight edges in the traversal path (replace with actual edges from your traversal)

nx.draw_networkx_edges(G, edgelist=[('A', 'B'), ('B', 'D'), ('D', 'F')], edge_color='red', width=2)

plt.show()
```

This code will produce a visualization of the graph with the traversal path highlighted in red, enabling you to visually trace the traversal algorithm's progress.

Lab 3: Graph Paths and Shortest Paths

- Implement algorithms to find paths between vertices (e.g., iterative DFS)

Here's a Python implementation of iterative DFS with modifications to find paths between vertices:

Python
```python
def find_path_dfs_iterative(graph, start, goal):

    stack = [(start, [start])]  # Tuples of (vertex, path)

    visited = set()



    while stack:

        (vertex, path) = stack.pop()

        if vertex not in visited:

            visited.add(vertex)

            if vertex == goal:

                return path

            stack.extend((neighbor, path + [neighbor]) for neighbor in
graph[vertex])



    return None  # No path found
```

Explanation:

1. Initialization:

    - Create a stack to manage the traversal, storing tuples of (vertex, path).
    - Create a set to track visited vertices.
2.
3. Exploration:

    - Pop a vertex and its path from the stack.

        - If the vertex is not visited:
            - Mark it as visited.
            - If it's the goal vertex, return the path.
            - Push its neighbors and their extended paths onto the stack.
        -
4.

Key Points:

- The `path` list in each tuple tracks the vertices visited during the traversal.
- When the goal vertex is found, the complete path from the start vertex is returned.
- If no path exists, the function returns `None`.

Example Usage:

Python
```
graph = {

    'A': ['B', 'C'],

    'B': ['D', 'E'],

    'C': ['F'],

    'D': [],

    'E': ['F'],

    'F': []

}



path = find_path_dfs_iterative(graph, 'A', 'F')
```

```
print(path)  # Output: ['A', 'B', 'D'] or ['A', 'C', 'F'] or ['A', 'B',
'E', 'F'] (depending on traversal order)
```

- Introduce Dijkstra's algorithm for finding shortest paths in weighted graphs

Here's an introduction to Dijkstra's algorithm for finding shortest paths in weighted graphs:

Purpose:

- Finds the shortest paths from a source vertex to all other vertices in a weighted graph.
- Weights can represent distances, travel times, costs, or other measures of edge lengths.

Key Concepts:

- Priority Queue: Used to efficiently select the next vertex to explore based on its current tentative distance.
- Tentative Distance: The shortest known distance from the source vertex to a given vertex at a particular point in the algorithm's execution.
- Visited Set: Keeps track of vertices whose shortest distances have already been finalized.

Algorithm Steps:

1. Initialization:

    - Mark all vertices as unvisited.
    - Assign a tentative distance of zero to the source vertex and infinity to all other vertices.

    - Create a priority queue and insert the source vertex with its tentative distance.
2.
3. Exploration:

    - While the priority queue is not empty:
        - Remove the vertex with the smallest tentative distance from the queue.
        - Mark it as visited.
        - For each of its unvisited neighbors:
            - Calculate the tentative distance to the neighbor through the current vertex.
            - If this tentative distance is less than the neighbor's current tentative distance, update the neighbor's distance and priority in the queue.
        -
    -
4.
5. Finalization:

    - The final shortest distances from the source vertex to all other vertices are stored in their respective tentative distance values.
6.

Key Points:

- Dijkstra's algorithm only works for non-negative edge weights.
- It's a single-source shortest path algorithm, meaning it finds shortest paths from a single source to all other vertices.
- It has a time complexity of O(V^2) for dense graphs and O(E log V) for sparse graphs, where V is the number of vertices and E is the number of edges.

- Apply shortest path algorithms to real-world scenarios (e.g., navigation)

Here are examples of how shortest path algorithms are applied in real-world scenarios:

1. Navigation Systems:

- GPS routing: Dijkstra's algorithm is often used to find the shortest or fastest route between two points on a road network, considering factors like distance, traffic conditions, and speed limits.
- In-car navigation systems: They use shortest path algorithms to guide drivers to their destinations, dynamically adjusting routes based on real-time traffic updates.
- Public transportation routing: Algorithms find the quickest or most efficient routes across multiple modes of transport, taking into account schedules, delays, and transfer times.

2. Logistics and Delivery:

- Vehicle routing: Algorithms optimize delivery routes for trucks, ensuring efficient delivery of goods to multiple locations while minimizing travel time and fuel costs.
- Package delivery: Companies use shortest path algorithms to plan efficient routes for couriers, considering factors like package weight, delivery deadlines, and traffic patterns.
- Supply chain optimization: Algorithms help optimize the flow of goods through complex supply chains, minimizing transportation costs and ensuring timely delivery.

3. Network Routing:

- Internet routing: Protocols like OSPF (Open Shortest Path First) use shortest path algorithms to determine the most efficient paths for data packets to travel through the internet.
- Telephone networks: Algorithms route calls through the most cost-effective paths, considering factors like call duration, time of day, and network congestion.

4. Social Networks:

- Friend recommendation: Algorithms suggest potential friends based on the shortest paths between users in the social network, considering factors like mutual friends and interests.
- Information diffusion: Algorithms analyze how information spreads through social networks by identifying the shortest paths between users who share content.

5. Gaming:

- Pathfinding in games: Algorithms guide characters through game environments, finding the shortest or safest routes to objectives while avoiding obstacles and enemies.
- Strategy games: Algorithms calculate optimal troop movements and resource allocation based on the shortest paths between locations and resource nodes.

Exercises:

Here are 10 lab exercises for practicing graph data structures in Python:

1. Graph Creation and Manipulation:

- Create graphs: Implement functions to create graphs using both adjacency list and adjacency matrix representations.
- Add and remove elements: Implement functions to add and remove vertices and edges from a graph.
- Visualize graphs: Use NetworkX or matplotlib to visualize graphs and explore their structure.

2. Graph Traversals:

- DFS and BFS implementations: Implement both recursive and iterative versions of Depth-First Search (DFS) and Breadth-First Search (BFS).
- Traversal applications:
  - Find connected components in a graph.
  - Identify cycles in a graph.
  - Determine if a graph is bipartite.
-

3. Pathfinding Algorithms:

- Iterative DFS: Implement iterative DFS to find paths between vertices in a general graph.
- Dijkstra's algorithm: Implement Dijkstra's algorithm to find the shortest paths from a source vertex to all other vertices in a weighted graph.
- Pathfinding applications:
  - Find the shortest route between cities on a map.
  - Determine the most efficient sequence of tasks in a project.
-

4. Graph Properties and Analysis:

- Calculate graph properties:
  - Degree of vertices.

    - Density of a graph.
    - Diameter of a graph.
-
- Analyze network characteristics: Use these properties to analyze the structure and patterns of different networks.

5. Topological Sorting:

- Implement topological sorting: Implement an algorithm to topologically sort a directed acyclic graph (DAG), ordering vertices based on their dependencies.
- Applications:
    - Schedule tasks in a project with dependencies.
    - Resolve dependency conflicts in software packages.
-

6. Minimum Spanning Trees:

- Prim's or Kruskal's algorithm: Implement Prim's or Kruskal's algorithm to find the minimum spanning tree of a weighted graph.
- Applications:
    - Design cost-efficient networks (e.g., telecommunications, transportation).
    - Identify clusters in data.
-

7. Graph Coloring:

- Implement graph coloring: Implement an algorithm to color the vertices of a graph so that no adjacent vertices share the same color.
- Applications:
    - Schedule tasks or exams to avoid conflicts.
    - Assign frequencies to radio stations to avoid interference.

-

*8. A Search:**

- *Implement A search:** Implement A* search, a pathfinding algorithm that uses heuristics to guide its exploration.
- Applications:
  - Pathfinding in games with complex terrains.
  - Route planning in navigation systems.
-

9. Graph Isomorphism:

- Explore graph isomorphism: Investigate techniques for determining if two graphs are structurally identical.
- Applications:
  - Pattern recognition in molecules.
  - Identifying similar social networks.
-

10. Graph Optimization:

- Solve optimization problems:
  - Traveling Salesman Problem: Find the shortest possible route that visits every vertex exactly once and returns to the starting vertex.
  - Graph partitioning: Divide a graph into subgraphs with specific properties.
  - Graph clustering: Identify groups of vertices that are closely connected.