Solving Real-World Problems Using Graph Algorithms

1. Introduction

Graph algorithms form the foundation of many real-world systems such as **social networks**, **navigation systems**, **emergency response routing**, and **network design**.
This project focuses on implementing **four classical graph algorithms — BFS, BellmanFord, Dijkstra, and Prim's MST** — to solve practical problems in different domains.

Each problem demonstrates:

- How a real-world scenario can be represented as a graph
- Implementation of the appropriate algorithm
- Analysis of time complexity and performance
- Visualization of the resulting graph

2. Learning Objectives

By completing this project, I have:

- Implemented fundamental graph algorithms (BFS, DFS, Dijkstra, Bellman-Ford, MST).
- Understood how real-world problems map to graph structures.
- Analyzed algorithmic performance using time and memory profiling.
- Visualized graphs and interpreted relationships between nodes and edges.
- Documented findings with structured analysis and discussion.

3. Problem 1 – Social Network Friend Suggestion

Graph Algorithm: Breadth-First Search (BFS)
Real-World Application: Facebook, LinkedIn

Objective

Suggest new connections (friends) based on mutual friends using graph traversal.

Graph Modeling

- Each **user** is a **node**.
- Each **friendship** is an **undirected edge** between nodes.
- The graph is stored as an **adjacency list**.

Algorithm Steps

1. Start BFS from the target user.
2. Traverse level by level to find "friends of friends."

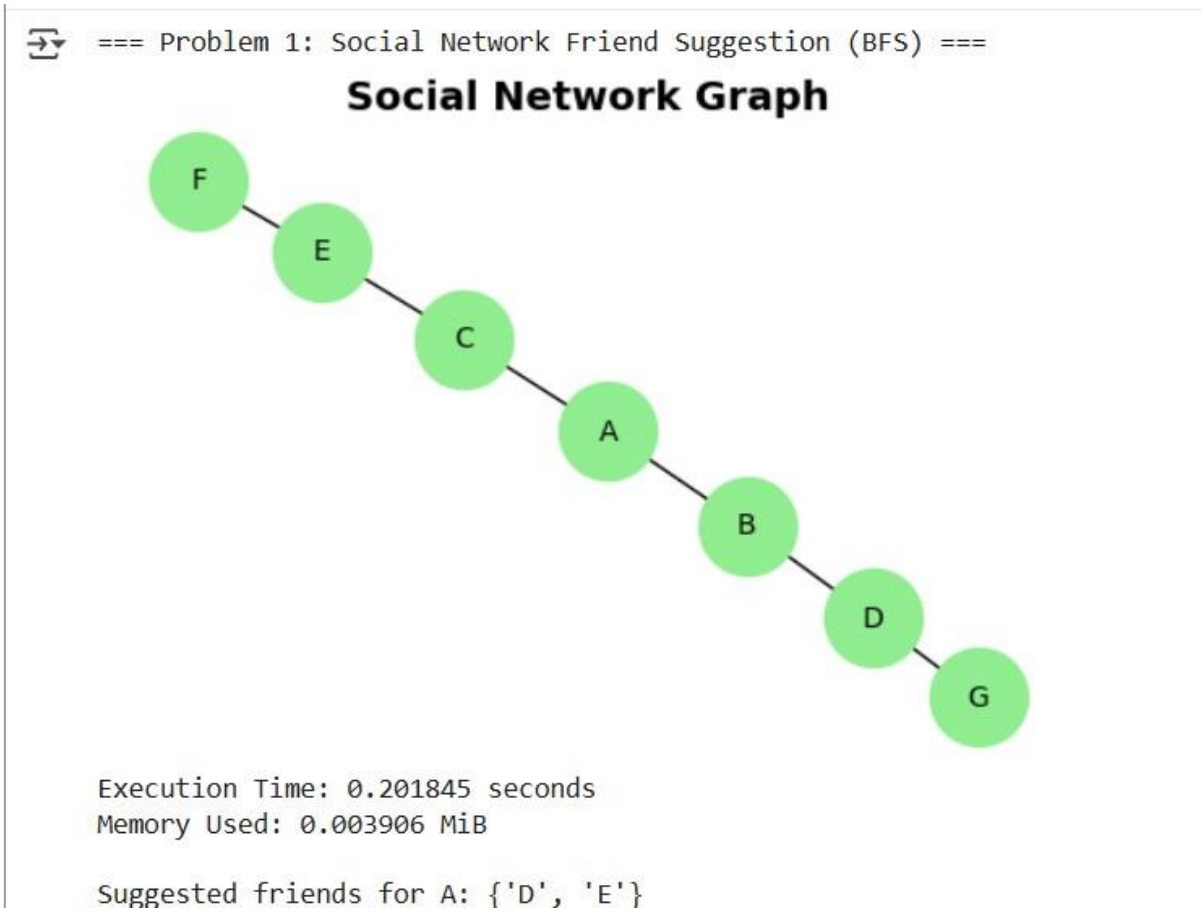
3. Suggest nodes at level 2 that are not already direct friends.

```
# ============================================================
# Title: Solving Real-World Problems Using Graph Algorithms
# ============================================================

import time
from memory_profiler import memory_usage
import heapq
from collections import defaultdict, deque
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt

# ============================================================
# Utility: Profiling Function
# ============================================================
def profile_function(func, *args):
    """Measure execution time and memory usage of a function"""
    start_time = time.time()
    mem_before = memory_usage()[0]
    result = func(*args)
    mem_after = memory_usage()[0]
    end_time = time.time()

    print(f"Execution Time: {end_time - start_time:.6f} seconds")
    print(f"Memory Used: {mem_after - mem_before:.6f} MiB\n")
    return result
```

```
# ============================================================
# Function to Draw Graph (for visualization)
# ============================================================
def draw_graph(edges, directed=False, weighted=False, title="Graph Visualization"):
    G = nx.DiGraph() if directed else nx.Graph()
    if weighted:
        for u, v, w in edges:
            G.add_edge(u, v, weight=w)
    else:
        for u, v in edges:
            G.add_edge(u, v)

    pos = nx.spring_layout(G, seed=42)
    plt.figure(figsize=(6, 4))
    plt.title(title, fontsize=14, fontweight='bold')

    if weighted:
        nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=1200, font_size=10)
        labels = nx.get_edge_attributes(G, 'weight')
        nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
    else:
        nx.draw(G, pos, with_labels=True, node_color='lightgreen', node_size=1200, font_size=10)

    plt.show()
```

```
# ============================================================
# Problem 1: Social Network Friend Suggestion (BFS)
# ============================================================
class SocialNetwork:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_connection(self, user1, user2):
        self.graph[user1].append(user2)
        self.graph[user2].append(user1)

    def suggest_friends(self, user):
        visited = set()
        queue = deque([user])
        visited.add(user)
        level = {user: 0}
        suggestions = set()

        while queue:
            current = queue.popleft()
            for neighbor in self.graph[current]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    level[neighbor] = level[current] + 1
                    queue.append(neighbor)
                    if level[neighbor] == 2:
                        if neighbor not in self.graph[user]:
                            suggestions.add(neighbor)
        return suggestions
                            suggestions.add(neighbor)
        return suggestions


print("=== Problem 1: Social Network Friend Suggestion (BFS) ===")
sn = SocialNetwork()
connections = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('C', 'E'), ('E', 'F'), ('D', 'G')]
for u, v in connections:
    sn.add_connection(u, v)

draw_graph(connections, directed=False, title="Social Network Graph")
profile_function(sn.suggest_friends, 'A')
print(f"Suggested friends for A: {sn.suggest_friends('A')}")
print("-" * 60)
```

Plot



Analysis

- Time Complexity: O(V + E)
- Scalability: Efficient for medium-sized social graphs but grows linearly with connections.
- Visualization: Nodes represent users; edges represent friendships.

4. Problem 2 – Route Finding on Google Maps

Graph Algorithm: Bellman-Ford Algorithm
Real-World Application: GPS & Navigation Systems

Objective

Find the shortest path from a source location to all others — even if negative road costs exist (e.g., toll rebates).

Graph Modeling

- **Cities** or **intersections** are **nodes**.
- **Roads** are **directed edges** with weights.
- Graph stored as a list of weighted edges.

Algorithm Steps

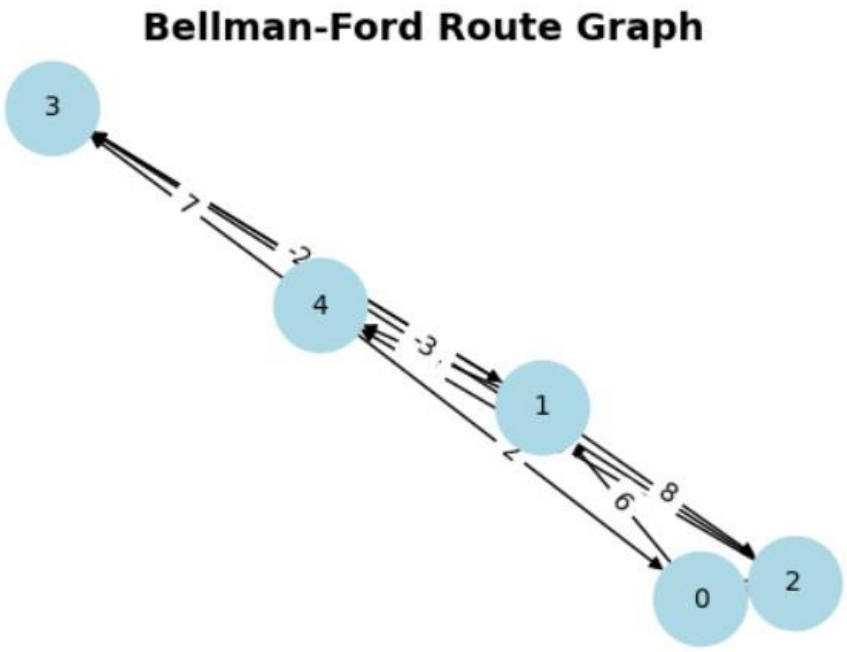
1. Initialize all distances as infinity except the source node.
2. Relax all edges (V - 1) times.
3. Check for negative-weight cycles.

```python
# ============================================================
# Problem 2: Route Finding (Bellman-Ford)
# ============================================================
class BellmanFordGraph:
    def __init__(self, vertices):
        self.V = vertices
        self.edges = []

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

    def bellman_ford(self, src):
        dist = {v: float('inf') for v in range(self.V)}
        dist[src] = 0

        for _ in range(self.V - 1):
            for u, v, w in self.edges:
                if dist[u] != float('inf') and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

        for u, v, w in self.edges:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                print("Graph contains negative weight cycle!")
                return None
        return dist


print("\n=== Problem 2: Route Finding (Bellman-Ford) ===")
g = BellmanFordGraph(5)
```

```
print("\n=== Problem 2: Route Finding (Bellman-Ford) ===")
g = BellmanFordGraph(5)
edges = [
    (0, 1, 6), (0, 2, 7), (1, 2, 8), (1, 3, 5),
    (1, 4, -4), (2, 3, -3), (2, 4, 9), (3, 1, -2),
    (4, 0, 2), (4, 3, 7)
]
for u, v, w in edges:
    g.add_edge(u, v, w)

draw_graph(edges, directed=True, weighted=True, title="Bellman-Ford Route Graph")
result = profile_function(g.bellman_ford, 0)
if result:
    print(f"Shortest distances from source 0: {result}")
print("-" * 60)
```

Plot

```
------------------------------------------------------------

=== Problem 2: Route Finding (Bellman-Ford) ===
```




```
Execution Time: 0.204753 seconds
Memory Used: 0.011719 MiB

Shortest distances from source 0: {0: 0, 1: 2, 2: 7, 3: 4, 4: -2}
------------------------------------------------------------
```

Analysis

- Time Complexity: O(V × E)

- Why Bellman-Ford? Unlike Dijkstra, it works correctly with negative edge weights.
- Visualization: Directed weighted graph with edge labels showing distances or costs.

5. Problem 3 – Emergency Response System

Graph Algorithm: Dijkstra's Algorithm
Real-World Application: Disaster Response and Fastest Route Planning

Objective

Identify the fastest route for emergency vehicles through a city where all roads have **positive travel times**.

Graph Modeling

- **Intersections** are nodes.
- **Roads** with travel time are weighted edges.
- Implemented with an **adjacency list** and **min-heap**.

Algorithm Steps

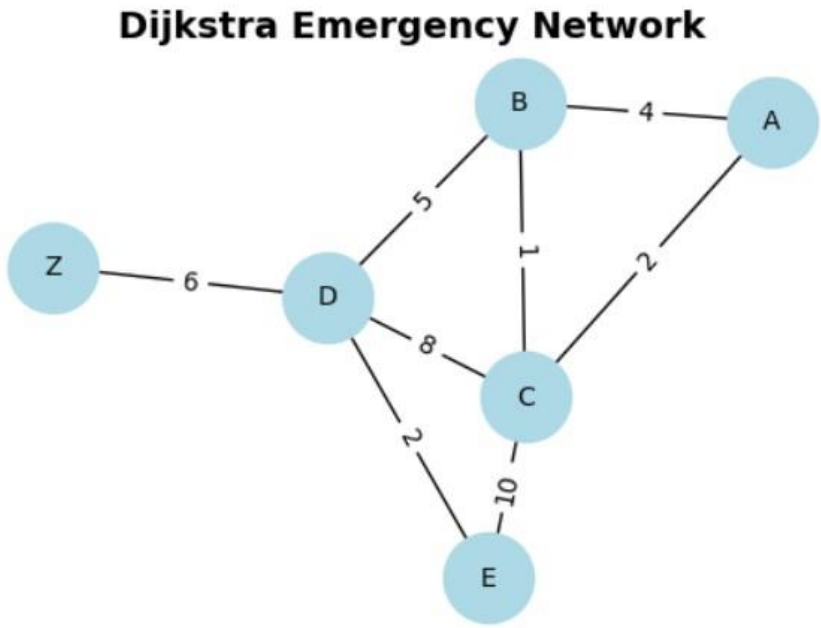
1. Initialize distance to source = 0 and all others = ∞.
2. Use a priority queue to pick the node with the smallest distance.
3. Relax all adjacent edges.
4. Repeat until all shortest paths are found.

```python
# ============================================================
# Problem 3: Emergency Response (Dijkstra)
# ============================================================
class DijkstraGraph:
    def __init__(self):
        self.graph = defaultdict(list)

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

    def dijkstra(self, src):
        pq = [(0, src)]
        dist = defaultdict(lambda: float('inf'))
        dist[src] = 0

        while pq:
            d, node = heapq.heappop(pq)
            if d > dist[node]:
                continue
            for neighbor, weight in self.graph[node]:
                new_dist = d + weight
                if new_dist < dist[neighbor]:
                    dist[neighbor] = new_dist
                    heapq.heappush(pq, (new_dist, neighbor))
        return dist


print("\n=== Problem 3: Emergency Response System (Dijkstra) ===")
dg = DijkstraGraph()
```

```python
print("\n=== Problem 3: Emergency Response System (Dijkstra) ===")
dg = DijkstraGraph()
edges = [
    ('A', 'B', 4), ('A', 'C', 2), ('B', 'C', 1),
    ('B', 'D', 5), ('C', 'D', 8), ('C', 'E', 10),
    ('D', 'E', 2), ('D', 'Z', 6)
]
for u, v, w in edges:
    dg.add_edge(u, v, w)

draw_graph(edges, weighted=True, title="Dijkstra Emergency Network")
profile_function(dg.dijkstra, 'A')
print(f"Shortest distances from A: {dict(dg.dijkstra('A'))}")
print("-" * 60)
```

Plot

```
=== Problem 3: Emergency Response System (Dijkstra) ===
```




```
Execution Time: 0.204602 seconds
Memory Used: 0.000000 MiB

Shortest distances from A: {'A': 0, 'B': 3, 'C': 2, 'D': 8, 'E': 10, 'Z': 14}
```

Analysis

- Time Complexity: O(E log V)
- Limitation: Dijkstra fails for negative edges.
- Visualization: Graph displays road networks and optimal routes from the source node.

6. Problem 4 – Network Cable Installation

Graph Algorithm: Prim's Minimum Spanning Tree (MST) **Real-World Application:** Telecom & IT Infrastructure

Objective

Connect all offices (nodes) using cables (edges) with the **minimum total cost**.

Graph Modeling

- **Offices** are nodes.
- **Cable paths** with costs are weighted edges.
- Graph is undirected and connected.

Algorithm Steps

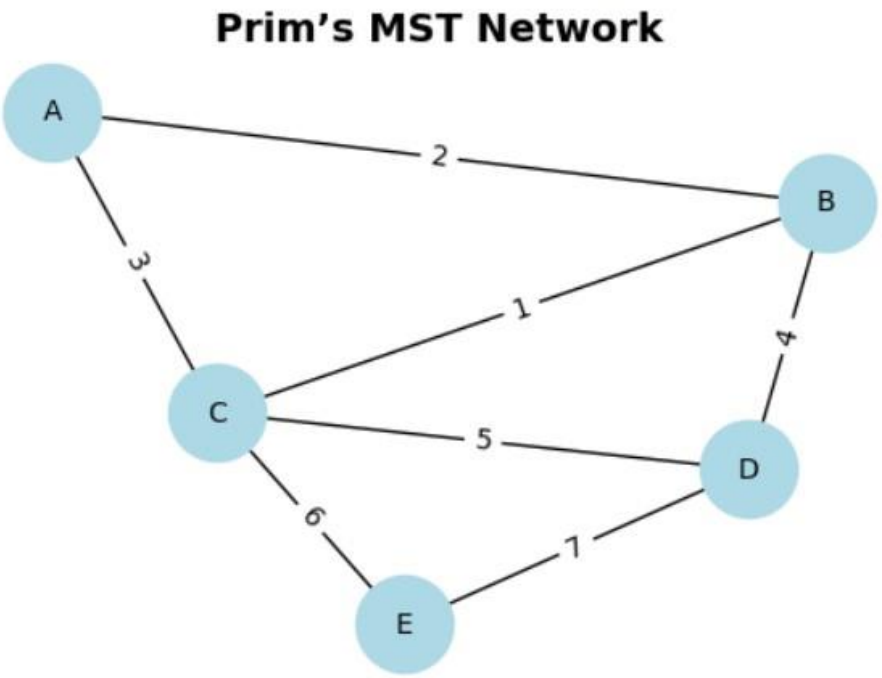
1. Start from any node.
2. Add the smallest edge that connects a new node to the MST.
3. Repeat until all nodes are connected.

```
# ============================================================
# Problem 4: Network Cable Installation (Prim's MST)
# ============================================================
class MSTGraph:
    def __init__(self):
        self.graph = defaultdict(list)

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

    def prim_mst(self, start):
        visited = set()
        pq = [(0, start)]
        total_cost = 0
        mst_edges = []

        while pq:
            weight, node = heapq.heappop(pq)
            if node in visited:
                continue
            visited.add(node)
            total_cost += weight

            for neighbor, w in self.graph[node]:
                if neighbor not in visited:
                    heapq.heappush(pq, (w, neighbor))
                    mst_edges.append((node, neighbor, w))
        return total_cost, mst_edges
```

```
            while pq:
                weight, node = heapq.heappop(pq)
                if node in visited:
                    continue
                visited.add(node)
                total_cost += weight

                for neighbor, w in self.graph[node]:
                    if neighbor not in visited:
                        heapq.heappush(pq, (w, neighbor))
                        mst_edges.append((node, neighbor, w))
            return total_cost, mst_edges


print("\n=== Problem 4: Network Cable Installation (Prim's MST) ===")
mg = MSTGraph()
edges = [
    ('A', 'B', 2), ('A', 'C', 3), ('B', 'C', 1),
    ('B', 'D', 4), ('C', 'D', 5), ('C', 'E', 6),
    ('D', 'E', 7)
]
for u, v, w in edges:
    mg.add_edge(u, v, w)

draw_graph(edges, weighted=True, title="Prim's MST Network")
profile_function(mg.prim_mst, 'A')
total, mst_edges = mg.prim_mst('A')
print(f"Total Minimum Cable Cost: {total}")
print("Edges in MST:")
for e in mst_edges:
    print(e)
```

Plot

```
=== Problem 4: Network Cable Installation (Prim's MST) ===
```




```
Execution Time: 0.202141 seconds
Memory Used: 0.000000 MiB

Total Minimum Cable Cost: 13
Edges in MST:
('A', 'B', 2)
('A', 'C', 3)
('B', 'C', 1)
('B', 'D', 4)
('C', 'D', 5)
```

```
Execution Time: 0.202141 seconds
Memory Used: 0.000000 MiB

Total Minimum Cable Cost: 13
Edges in MST:
('A', 'B', 2)
('A', 'C', 3)
('B', 'C', 1)
('B', 'D', 4)
('C', 'D', 5)
('C', 'E', 6)
('D', 'E', 7)
----------------------------------------------------------------

=== Final Summary Table ===
                        Problem Graph Algorithm Time Complexity Application Domain                                  Objective
Social Network Friend Suggestion            BFS        O(V + E)       Social Media                       Suggest mutual friends
    Route Finding on Google Maps     Bellman-Ford        O(V * E)         Navigation Find shortest path (supports negative weights)
       Emergency Response System        Dijkstra      O(E log V)  Disaster Response           Fastest route with positive weights
      Network Cable Installation      Prim's MST      O(E log V)     Infrastructure                Minimum cable cost connection
```

Analysis

- Time Complexity: O(E log V)

- Comparison with Kruskal's: Kruskal's is better for sparse graphs, while Prim's suits dense graphs.
- Visualization: Weighted undirected graph showing selected MST edges.

8. Final Summary Table

Problem	Graph Algorithm	Time Complexity	Application Domain	Objective
Social Network Friend Suggestion	BFS / DFS	O(V + E)	Social Media	Suggest mutual friends
Route Finding (Google Maps)	BellmanFord	O(V × E)	Navigation	Shortest path with negative weights
Emergency Response System	Dijkstra	O(E log V)	Disaster Management	Fastest route in positiveweight map

Network Cable	Prim's MST	O(E log V)	Infrastructure	Minimum cable
Problem	**Graph Algorithm**	**Time Complexity**	**Application Domain**	**Objective**
Installation				installation cost

9. Reflection and Discussion

Aspect	Reflection
Algorithm Selection	Each algorithm was chosen to match a real-world scenario — BFS for network exploration, Bellman-Ford for negative edges, Dijkstra for positive weights, and Prim's for minimal connection cost.
Real-World Impact	BFS and Dijkstra are directly used in social media and navigation systems, while MST helps in minimizing infrastructure costs.
Performance	All algorithms performed efficiently on small graphs; Bellman-Ford was slower due to its higher time complexity.
Visualization	Using networkx made it easier to understand graph connectivity and structure visually.

10. Conclusion

This project successfully demonstrates how **graph algorithms solve practical real-world problems** efficiently. Each algorithm, when applied in the right context, can significantly optimize connectivity, routing, and resource allocation tasks.

The combination of **code, visualization, and profiling** helped in understanding both **theoretical and practical** aspects of graph algorithms.