

DESIGN AND ANALYSIS OF ALGORITHMS

BACHELOR OF COMPUTER APPLICATIONS

to

K.R Mangalam University

by

ADITYA RAJ SINHA (2301201189)

Lab Assignment 3

Graph Algorithms

Faculty: Dr. Aarti Sangwan



Department of Computer Science and Engineering

School of Engineering and Technology

K.R Mangalam University, Gurugram- 122001, India

April 2025

Solving Real-World Problems Using Graph Algorithms

Aditya Raj Sinha
2301201189

Contents

1	Introduction	2
2	Problem 1: Social Network Friend Suggestion (BFS)	3
2.1	Real-World Context	3
2.2	Graph Modeling	3
2.3	Algorithm Choice: BFS	3
2.4	Time Complexity	3
2.5	Python Implementation	4
2.6	Illustration	4
3	Problem 2: Route Finding with Negative Weights (Bellman-Ford)	5
3.1	Real-World Context	5
3.2	Graph Modeling	5
3.3	Algorithm Choice	5
3.4	Time Complexity	5
3.5	Python Implementation	5
3.6	Illustration	6
4	Problem 3: Emergency Response System (Dijkstra)	7
4.1	Real-World Context	7
4.2	Graph Modeling	7
4.3	Algorithm Choice	7
4.4	Time Complexity	7
4.5	Python Implementation	7
4.6	Illustration	8
5	Problem 4: Minimum Cable Cost (Prim's MST)	9
5.1	Real-World Context	9
5.2	Graph Modeling	9
5.3	Algorithm Choice	9
5.4	Time Complexity	9
5.5	Python Implementation	9
5.6	Illustration	10
6	Profiling and Performance Evaluation	11
7	Summary and Comparison	12
8	Conclusion	13

Chapter 1

Introduction

Graph algorithms are used everywhere: from social networking and navigation to emergency response and network cabling. This project demonstrates how four classical graph algorithms can be mapped to four real-world applications:

- BFS for Social Network Friend Suggestions
- Bellman-Ford for Navigation with Negative Weights
- Dijkstra for Emergency Response Route Planning
- Prim's MST for Network Cable Installation

The objective of this report is to explain:

- How each real-world problem can be modeled as a graph
- Why a particular algorithm fits that scenario
- Implementation details (Python, cell-wise)
- Profiling (time and memory)
- Final comparative analysis

Chapter 2

Problem 1: Social Network Friend Suggestion (BFS)

2.1 Real-World Context

Platforms like Facebook or LinkedIn recommend friends by identifying people who share common connections. If two users have many mutual friends, the platform considers them likely to know each other.

2.2 Graph Modeling

- Nodes represent users.
- Edges represent friendships.
- The graph is undirected.

2.3 Algorithm Choice: BFS

BFS explores nodes level by level:

- Level 0: The user
- Level 1: Direct friends
- Level 2: Friends of friends

Nodes at Level 2 but not at Level 1 become suggested friends.

2.4 Time Complexity

$$O(V + E)$$

Efficient even for large networks.

2.5 Python Implementation

```
def friend_suggestions_bfs( adj , user ):  
    from collections import deque  
    visited = set([ user ])  
    q = deque([( user ,0)])  
    direct = set( adj .get( user ,[]))  
    fof = {}  
    while q:  
        node , d = q .popleft ()  
        if d==2:  
            continue  
        for nb in adj .get( node ,[]):  
            if nb not in visited :  
                visited .add( nb)  
                q.append(( nb ,d+1))  
        if d==1:  
            for nb in adj .get( node ,[]):  
                if nb!=user and nb not in direct :  
                    fof [nb] = fof .get(nb,0)+1  
    s = sorted( fof .items () , key=lambda x:(-x[1] ,x[0]))  
    return [x[0] for x in s]
```

2.6 Illustration

Friend Suggestions for A (Blue = A, Green = Direct Friends, Orange = Suggestions)

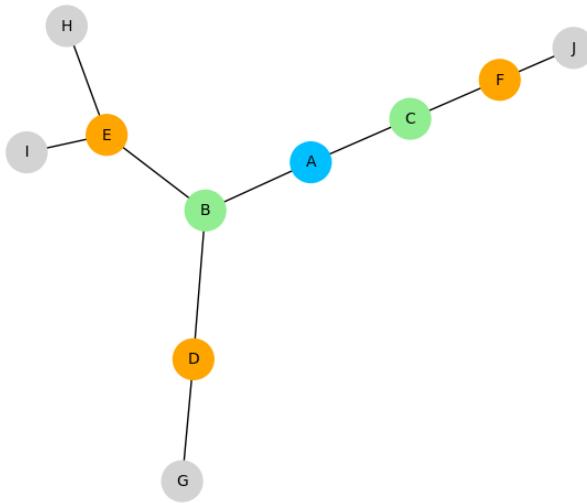


Figure 2.1: Friend suggestion graph: blue (user), green (direct friends), orange (suggested).

Chapter 3

Problem 2: Route Finding with Negative Weights (Bellman-Ford)

3.1 Real-World Context

Navigation apps sometimes include negative weights (e.g., subsidies, energy gains, toll refunds). Bellman-Ford works safely even with negative weights.

3.2 Graph Modeling

- Nodes = cities
- Directed edges = roads
- Edge weight = travel cost (may be negative)

3.3 Algorithm Choice

Bellman-Ford is used because:

- Dijkstra cannot handle negative weights.
- Bellman-Ford detects negative cycles.

3.4 Time Complexity

$$O(VE)$$

3.5 Python Implementation

```
def bellman_ford(edges, V, src):  
    dist = [float('inf')] * V  
    dist[src] = 0  
    for _ in range(V-1):  
        changed = False
```

```

for u,v,w in edges:
    if dist[u] != float('inf') and dist[u] + w < dist[v]:
        dist[v] = dist[u] + w
        changed = True
    if not changed:
        break
neg = False
for u,v,w in edges:
    if dist[u] + w < dist[v]:
        neg = True
        break
return dist , neg

```

3.6 Illustration

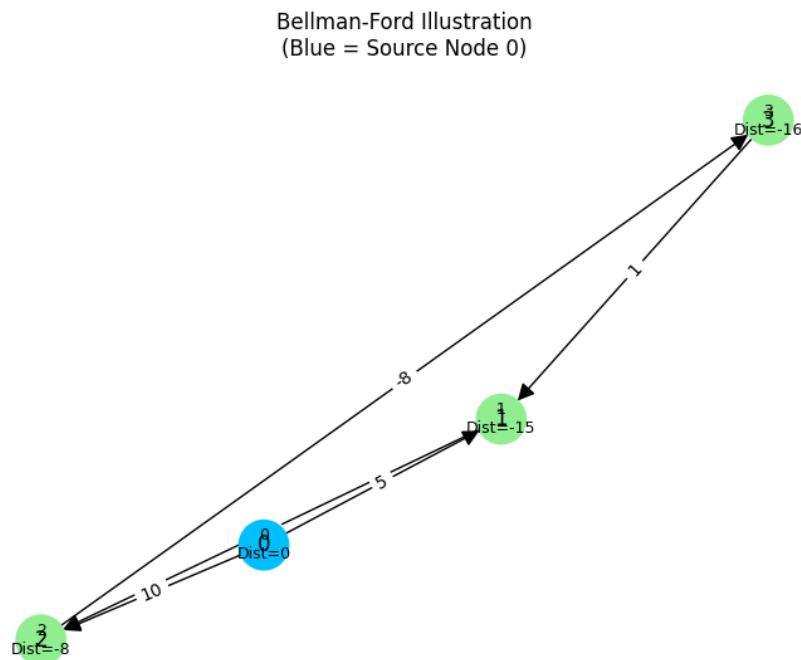


Figure 3.1: Bellman-Ford directed graph with weights.

Chapter 4

Problem 3: Emergency Response System (Dijkstra)

4.1 Real-World Context

Emergency vehicles require the fastest possible route where all travel times are positive.

4.2 Graph Modeling

- Nodes = intersections
- Weighted edges = travel time on roads

4.3 Algorithm Choice

Dijkstra efficiently finds shortest paths when all weights are positive.

4.4 Time Complexity

$$O(E \log V)$$

4.5 Python Implementation

```
def dijkstra(adj, src):  
    dist = {u: float('inf') for u in adj}  
    dist[src] = 0  
    pq = [(0, src)]  
    while pq:  
        d, u = heapq.heappop(pq)  
        if d > dist[u]:  
            continue  
        for v, w in adj[u]:  
            nd = d + w  
            if nd < dist[v]:
```

```

        dist [ v ] = nd
        heapq.heappush(pq, (nd, v))
return dist

```

4.6 Illustration

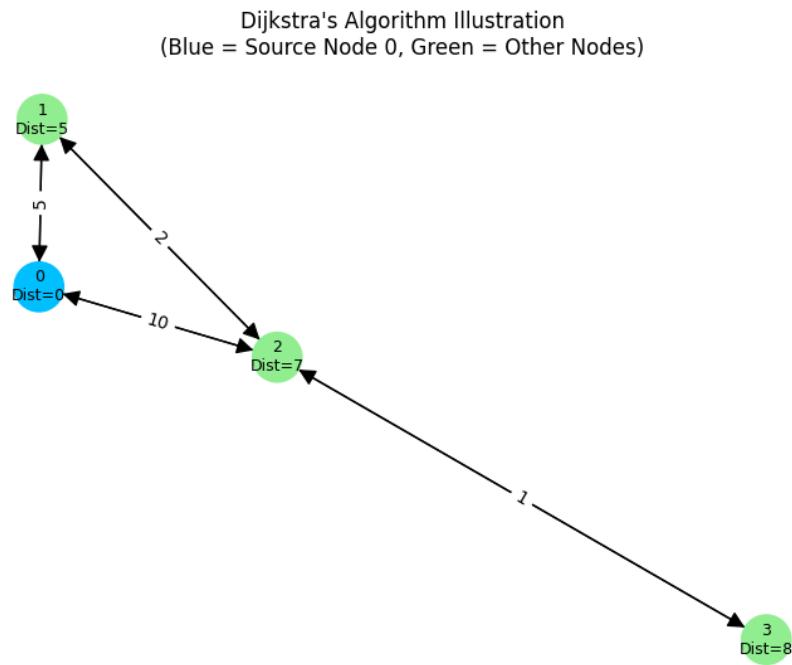


Figure 4.1: Weighted graph used in Dijkstra's algorithm.

Chapter 5

Problem 4: Minimum Cable Cost (Prim's MST)

5.1 Real-World Context

Telecom companies must connect offices using the minimum cable length.

5.2 Graph Modeling

- Nodes = offices
- Weighted edges = cable cost

5.3 Algorithm Choice

Prim's MST minimizes total connection cost.

5.4 Time Complexity

$$O(E \log V)$$

5.5 Python Implementation

```
def prim_mst(adj , start=0):
    visited = set ([ start ])
    pq = []
    for v,w in adj [ start ]:
        heapq.heappush(pq, (w, start , v))
    total = 0
    edges = []
    while pq and len(visited) < len(adj):
        w,u,v = heapq.heappop(pq)
        if v in visited:
            continue
        total += w
        edges.append((u,v))
        visited.add(v)
    return edges, total
```

```

visited . add( v )
total += w
edges . append( ( u , v , w ) )
for nxv , nw in adj [ v ] :
    if nxv not in visited :
        heapq . heappush( pq , ( nw , v , nxv ) )
return total , edges

```

5.6 Illustration

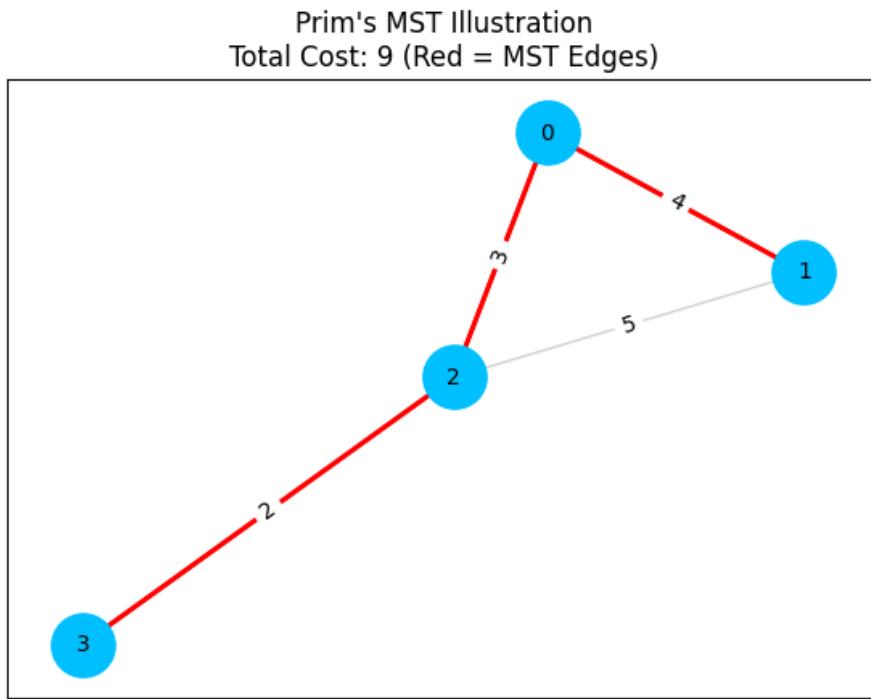


Figure 5.1: MST selected edges highlighted.

Chapter 6

Profiling and Performance Evaluation

Time and memory were recorded for various graph sizes (100, 300, 600 nodes). Plots included execution time vs number of nodes and memory usage vs number of nodes.

Summary:

- BFS: fastest, very low memory use.
- Dijkstra and Prim: scale logarithmically due to priority queue.
- Bellman-Ford: slowest because complexity is $O(VE)$.

Chapter 7

Summary and Comparison

Problem	Algorithm	Complexity	Notes
Social Network	BFS	$O(V + E)$	Mutual-friend suggestions
Navigation	Bellman-Ford	$O(VE)$	Handles negative weights
Emergency Response	Dijkstra	$O(E \log V)$	Only positive weights
Network Cabling	Prim MST	$O(E \log V)$	Minimum total cost

Table 7.1: Comparison of tasks and algorithms

Chapter 8

Conclusion

Real-world problems can be effectively modeled using graph theory. Choosing the correct algorithm depends entirely on graph structure and constraints.

This project demonstrated:

- How different problems map to different graph types
- Why algorithm choice matters
- Practical Python implementation
- Performance behavior across graph sizes

Graph algorithms remain some of the most useful tools in computer science and industry.