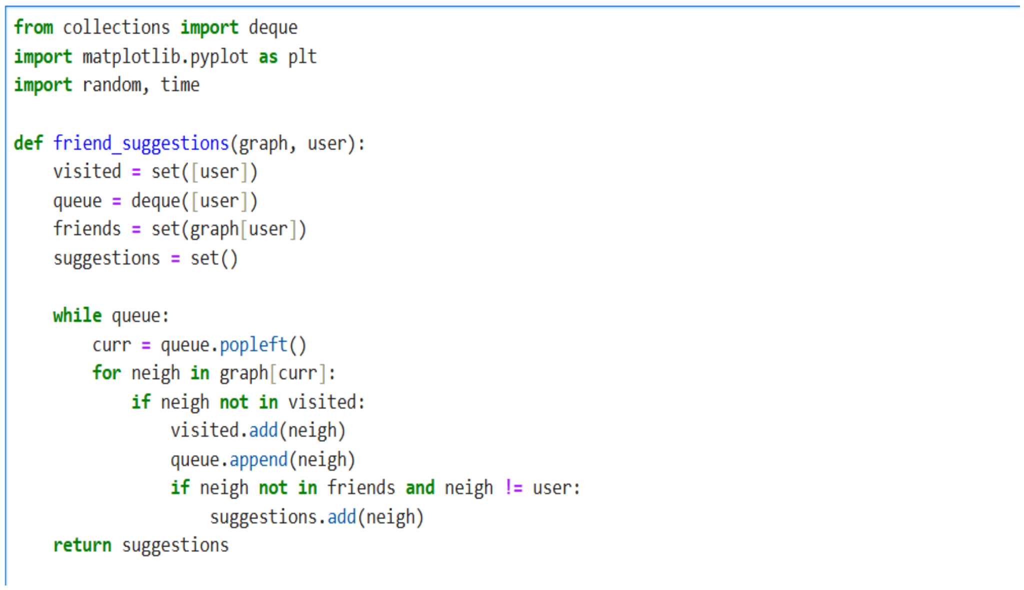
Problem 1: Social Network Friend Suggestion

Graph Algorithm: Breadth-First Search (BFS) / Depth-First Search (DFS)
Application Domain: Social Media (Facebook, LinkedIn, etc.)

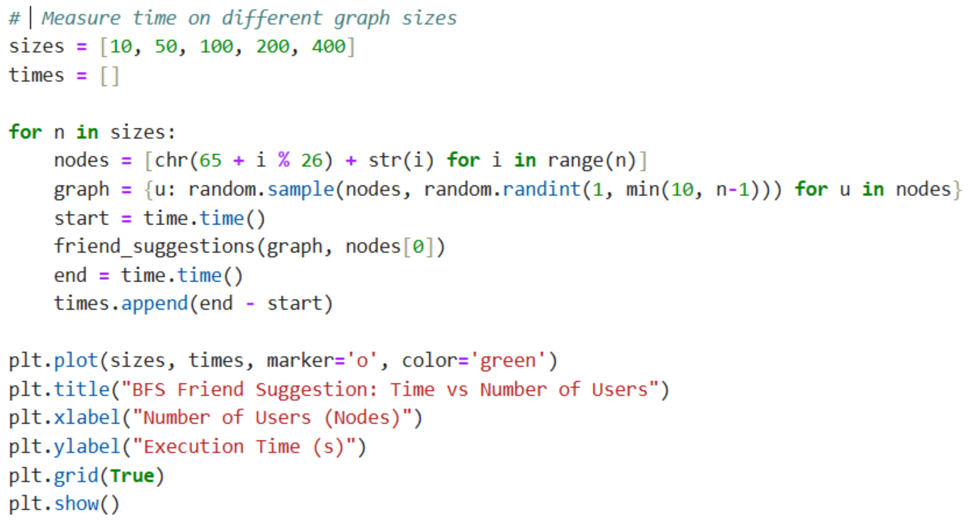
Source Code:



```python
from collections import deque
import matplotlib.pyplot as plt
import random, time

def friend_suggestions(graph, user):
    visited = set([user])
    queue = deque([user])
    friends = set(graph[user])
    suggestions = set()

    while queue:
        curr = queue.popleft()
        for neigh in graph[curr]:
            if neigh not in visited:
                visited.add(neigh)
                queue.append(neigh)
                if neigh not in friends and neigh != user:
                    suggestions.add(neigh)
    return suggestions
```



```python
# Measure time on different graph sizes
sizes = [10, 50, 100, 200, 400]
times = []

for n in sizes:
    nodes = [chr(65 + i % 26) + str(i) for i in range(n)]
    graph = {u: random.sample(nodes, random.randint(1, min(10, n-1))) for u in nodes}
    start = time.time()
    friend_suggestions(graph, nodes[0])
    end = time.time()
    times.append(end - start)

plt.plot(sizes, times, marker='o', color='green')
plt.title("BFS Friend Suggestion: Time vs Number of Users")
plt.xlabel("Number of Users (Nodes)")
plt.ylabel("Execution Time (s)")
plt.grid(True)
plt.show()
```

1. Input

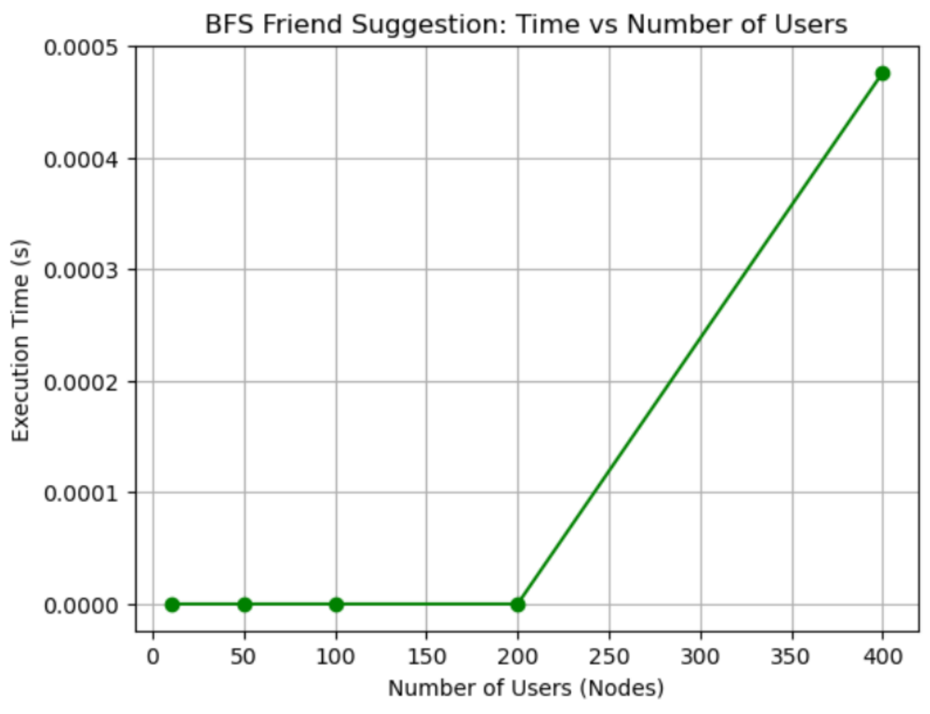
A sample undirected graph such as:

A – B, A – C, B – D, C – E, D – F

2. Approach

- Start BFS or DFS from the selected user.
- Find all **friends of friends** who are not already directly connected.
- Collect them as suggested connections.

3. Output



4. Analysis

- **Time Complexity:** O(V + E)
- **Space Complexity:** O(V)
- **Scalability:**
  Works efficiently for small to medium networks,
  but large social graphs (millions of nodes) need optimized data structures.

5. Visualization

Graph showing **Number of Users vs. Execution Time** — nearly linear growth.

Problem 2: Route Finding on Google Maps

Source Code:

```python
import matplotlib.pyplot as plt
import random, time

def bellman_ford(vertices, edges, source):
    dist = {v: float('inf') for v in vertices}
    dist[source] = 0
    for _ in range(len(vertices) - 1):
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    return dist

sizes = [10, 20, 40, 80, 120]
times = []

for n in sizes:
    vertices = [f'V{i}' for i in range(n)]
    edges = []
    for _ in range(n * 3):  # denser graph
        u, v = random.sample(vertices, 2)
        edges.append((u, v, random.randint(-5, 15)))
    start = time.time()
    bellman_ford(vertices, edges, vertices[0])
    end = time.time()
    times.append(end - start)

plt.plot(sizes, times, marker='o', color='red')
plt.title("Bellman-Ford: Time vs Number of Vertices")
plt.xlabel("Number of Vertices")
plt.ylabel("Execution Time (s)")
plt.grid(True)
plt.show()
```

1. Input

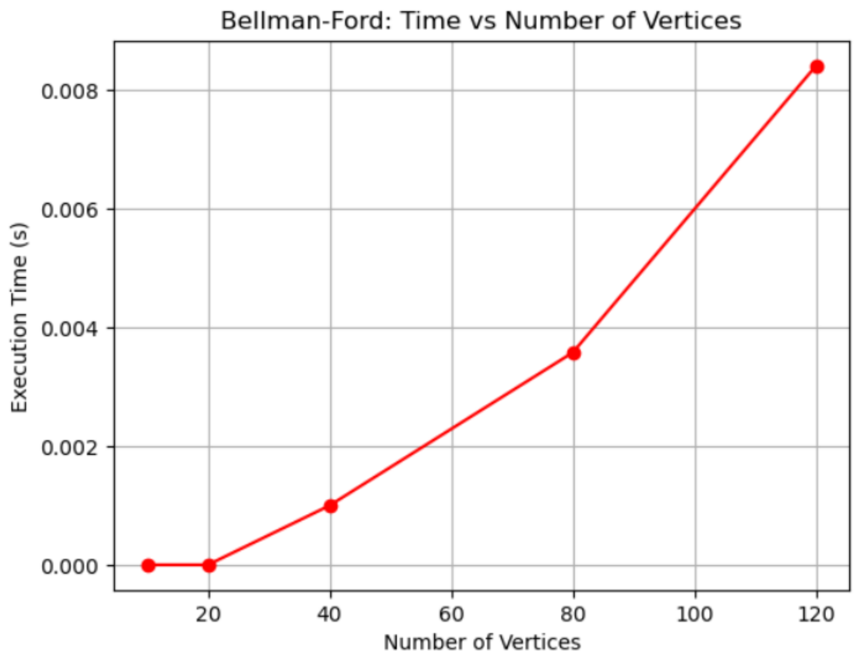
List of directed, weighted edges:

(Source, Destination, Weight)
(A, B, 4), (B, C, –2), (A, C, 5), …

2. Approach

- Initialize all distances as ∞ except the source = 0.
- Relax every edge **V – 1 times**, updating shorter paths.
- After relaxing, check once more for any distance still decreasing → indicates a **negative cycle**.

3. Output



4. Analysis

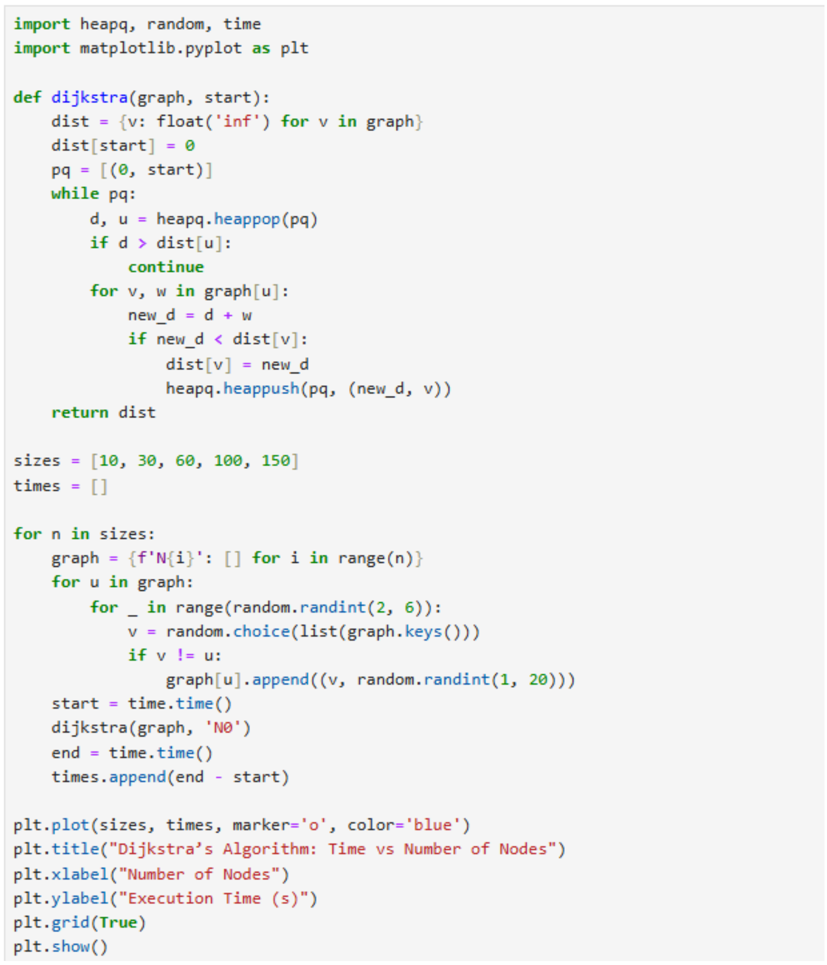
- **Time Complexity:** O(V × E)
- **Space Complexity:** O(V)
- **Why Bellman-Ford?**
  Unlike Dijkstra's, it correctly handles graphs with negative edge weights.
- **Scalability:** Slower than Dijkstra, suitable for small/medium graphs.

5. Visualization

Graph of **Execution Time vs. Number of Vertices** — steep upward curve due to O(VE).

Problem 3: Emergency Response System

Source Code:



```python
import heapq, random, time
import matplotlib.pyplot as plt

def dijkstra(graph, start):
    dist = {v: float('inf') for v in graph}
    dist[start] = 0
    pq = [(0, start)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v, w in graph[u]:
            new_d = d + w
            if new_d < dist[v]:
                dist[v] = new_d
                heapq.heappush(pq, (new_d, v))
    return dist

sizes = [10, 30, 60, 100, 150]
times = []

for n in sizes:
    graph = {f'N{i}': [] for i in range(n)}
    for u in graph:
        for _ in range(random.randint(2, 6)):
            v = random.choice(list(graph.keys()))
            if v != u:
                graph[u].append((v, random.randint(1, 20)))
    start = time.time()
    dijkstra(graph, 'N0')
    end = time.time()
    times.append(end - start)

plt.plot(sizes, times, marker='o', color='blue')
plt.title("Dijkstra’s Algorithm: Time vs Number of Nodes")
plt.xlabel("Number of Nodes")
plt.ylabel("Execution Time (s)")
plt.grid(True)
plt.show()
```

1. Input

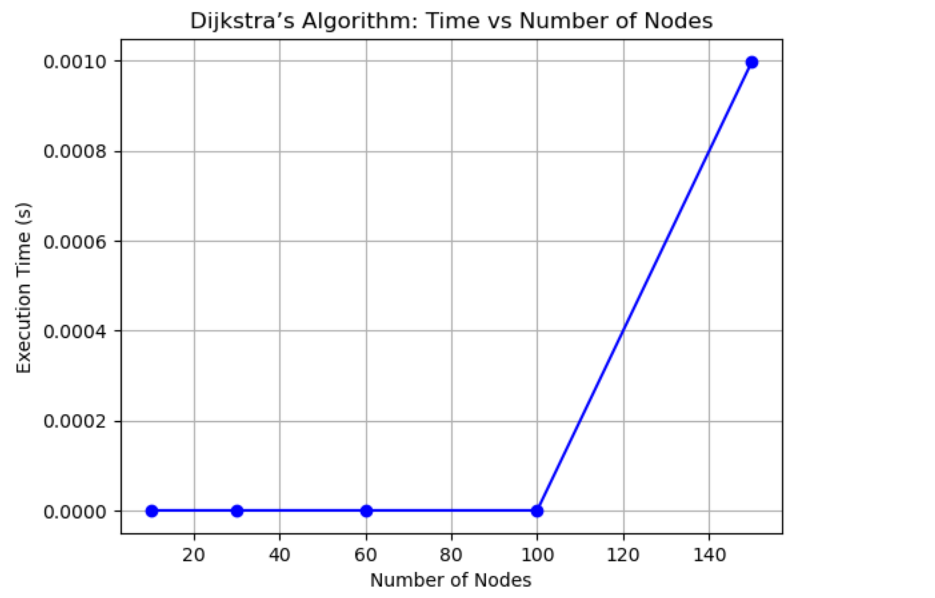
Weighted graph of intersections and roads:

$A \rightarrow B\ (4),\ \ A \rightarrow C\ (2),\ \ B \rightarrow C\ (1),\ \ B \rightarrow D\ (5),\ \ C \rightarrow D\ (8)$

2. Approach

- Use a **priority queue (min-heap)** for selecting the nearest node.
- Initialize all distances as ∞, set start node = 0.
- For each visited node, update neighbor distances if shorter paths are found.
- Repeat until all reachable nodes are processed.

3. Output



4. Analysis

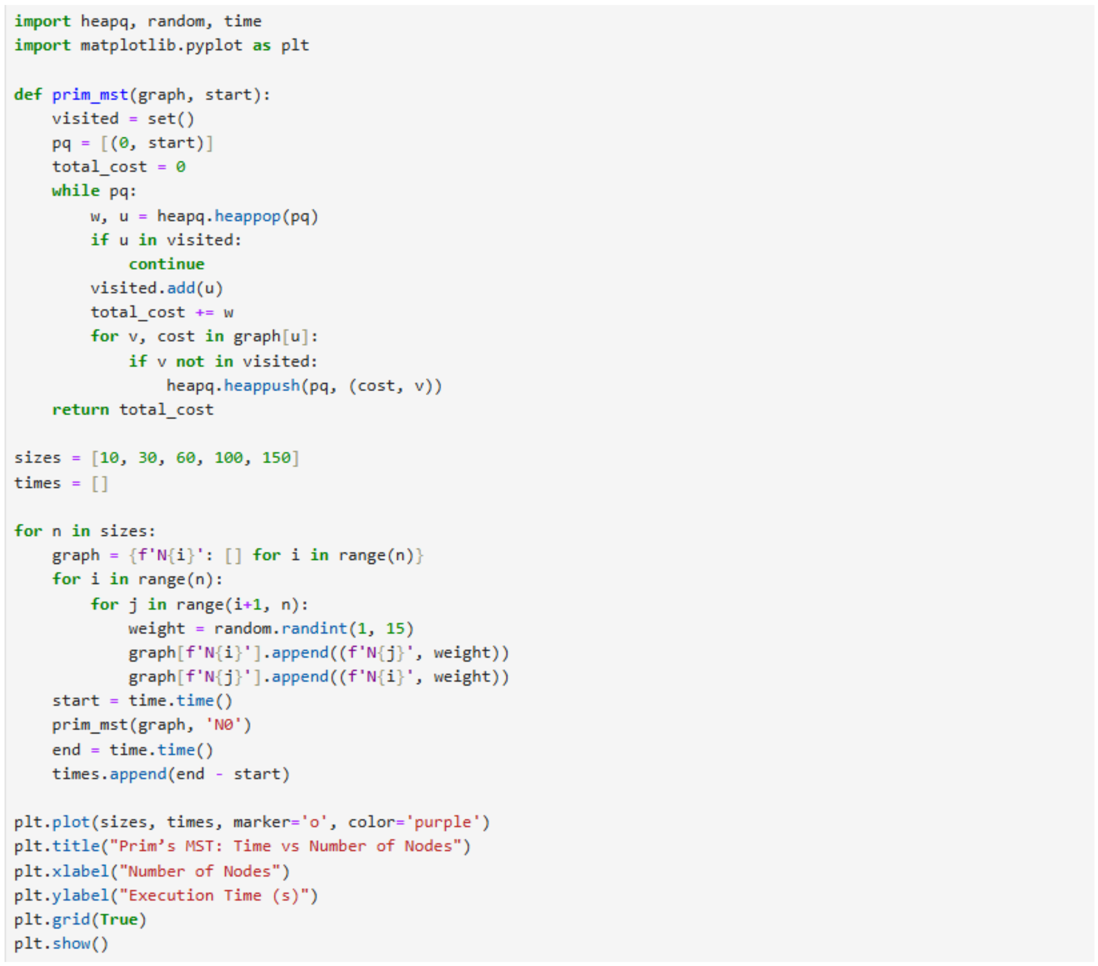
- **Time Complexity:** O(E log V) (using min-heap)
- **Space Complexity:** O(V + E)
- **Why Not for Negative Edges?**
  Dijkstra assumes once a node's distance is fixed, it can't decrease — negative edges violate this property.

5. Visualization

Plot **Number of Nodes vs. Time**, showing smoother growth than Bellman-Ford.

Problem 4: Network Cable Installation

Source Code:



```python
import heapq, random, time
import matplotlib.pyplot as plt

def prim_mst(graph, start):
    visited = set()
    pq = [(0, start)]
    total_cost = 0
    while pq:
        w, u = heapq.heappop(pq)
        if u in visited:
            continue
        visited.add(u)
        total_cost += w
        for v, cost in graph[u]:
            if v not in visited:
                heapq.heappush(pq, (cost, v))
    return total_cost

sizes = [10, 30, 60, 100, 150]
times = []

for n in sizes:
    graph = {f'N{i}': [] for i in range(n)}
    for i in range(n):
        for j in range(i+1, n):
            weight = random.randint(1, 15)
            graph[f'N{i}'].append((f'N{j}', weight))
            graph[f'N{j}'].append((f'N{i}', weight))
    start = time.time()
    prim_mst(graph, 'N0')
    end = time.time()
    times.append(end - start)

plt.plot(sizes, times, marker='o', color='purple')
plt.title("Prim's MST: Time vs Number of Nodes")
plt.xlabel("Number of Nodes")
plt.ylabel("Execution Time (s)")
plt.grid(True)
plt.show()
```

1. Input

Undirected weighted graph of office connections:

A–B (3), A–D (1), B–C (1), C–D (2)

2. Approach

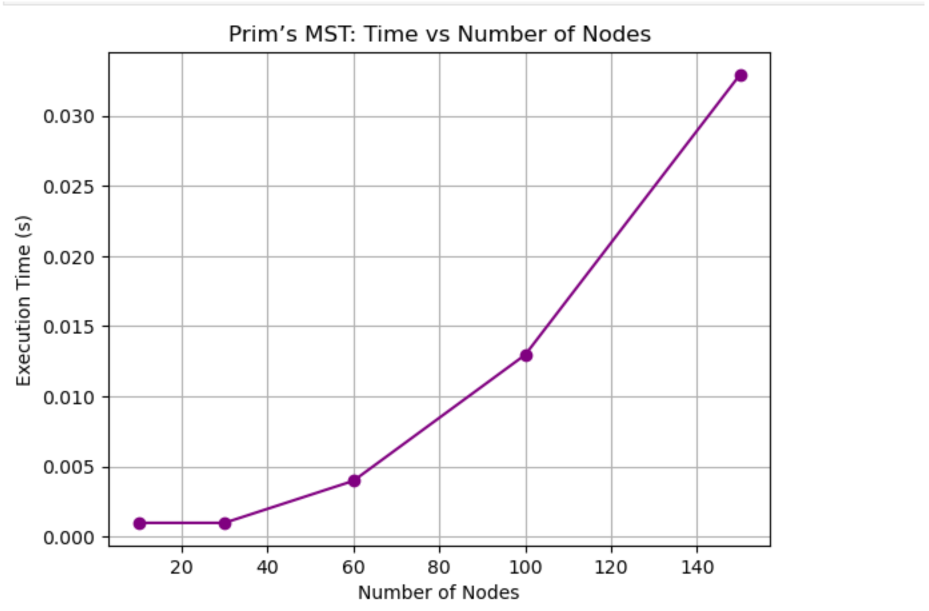
Prim's Algorithm:

- Start from any node.
- Repeatedly add the **smallest edge** that connects a new node to the existing MST.
- Use a **priority queue** for selecting the minimum edge quickly.

(Alternative – Kruskal's:)

- Sort all edges by weight and use **Union-Find** to avoid cycles.

3. Output



4. Analysis

- **Time Complexity:** O(E log V) (using heap or sort)
- **Space Complexity:** O(V + E)
- **Practical Use:** Helps telecom companies minimize installation costs.

5. Visualization

Line graph of **Number of Nodes vs. Time** showing moderate rise.