DSA - Project 2

Omckar Savlani
Preetam Hegde
Group 15

Tasks:

- **Single-source Shortest Path**
    - Find shortest path tree in both directed and undirected weighted graphs for a given source vertex. Assume there is no negative edge in your graph. You will print each path and path cost for a given source.
- **Minimum Spanning Tree Algorithm (Kruskal)**
    - Given a connected, undirected, weighted graph, find a spanning tree using edges that minimizes the total weight. Use Kruskal's algorithm to find Minimum Spanning Tree (MST). You will print out edges of the tree and total cost of minimum spanning tree.
- **DFS - Topological Sorting and Cycles**
    - Given a directed graph G with V vertices and E edges, execute Depth First Search (DFS) on G.
        - If the graph is acyclic, display the topological sorting sequence.
        - If the graph contains cycles, provide all cycles along with their lengths.

Implementation:

Task 1: Single-source Shortest Path
Chosen Algorithm: **Dijkstra** (dynamic programming)

Dijkstra's algorithm finds the shortest path from one vertex to all other vertices in the graph. It takes a weighted or unweighted graph represented as an edge list and finds the shortest distance from the source node (distance = 0) to all others (distance = ∞).

Key components:

- Priority Queue (Min-Heap): For efficient extraction of minimum distance vertex
- Distance Dictionary: Stores shortest known distances from source
- Graph Representation

Space complexity:

Dijkstra's algorithm in general requires a priority queue and a distance table, which can consume significant memory for very large graphs with millions of nodes and edges.

In our implementation of the algorithm, the space complexity analysis is as follows:
- Distance dictionary: O(V) - stores one value per vertex
- Priority Queue: O(V)
- Graph storage: O(V + E) - adjacency list
- Auxiliary variables: O(1) - loop counters, temporary variables

So, the total space complexity turns out to be: **O(V + E)**

Time complexity:

The time complexity of Dijkstra's algorithm in general depends on the data structure used for the priority queue, commonly (O(V^{2})) with an adjacency matrix or a simple array, and (O((V+E)\log V)) when using a binary heap. With a Fibonacci heap, the complexity can be improved to (O(E+V\log V)). For larger graphs bidirectional Dijkstra can be applied to optimize the run time.

In our implementation of the algorithm, the time complexity analysis is as follows:
- Initialization: O(V) - initialize distance dictionary
- Heap operations:
    - Insertion: O(V log V)
    - Updation: O(E log V)
- Neighbor processing: O(E)

Therefore, the overall time taken would be: O(V log V + E log V) = **O((V + E) log V)**
The best case would be O(V log V) for sparse graphs and worst case would be O(V^2 log V) for dense or complete graphs.

Data Structures Used:

- Priority Queue (Min-Heap):

This was implemented using python's heapq module used to extract minimum vertex distance. Heap operations used for extraction require O(log n) time in contrast to the O(V) time it would have taken to extract from an unsorted array.

- Distance Dictionary:

Used to store the shortest distance to each vertex taking O(1) time.

- Graph Representation:

It is the dictionary of dictionaries taking O(1) time for neighbour cases.

Procedure:

- 1. Initialization

Start at the source node. Set the distance to the source node as 0 and all other nodes to infinity (∞). Add all nodes to the priority queue.

- 2. Relaxation

Pick the closest node from the priority queue (initially source node). For each neighbor, check if going through the source node offers a shorter distance. If yes, update the distance.

- 3. Mark as Visited

Once processed, mark the source node as visited, then move to the next closest node in the queue.

- 4. Repeat

From the next node, explore neighbors and compare total path cost to the neighbouring nodes. If a shorter route is found in the explored nodes, the min path value is updated in the table. Keep going until the queue is empty, always picking the nearest node.

Pseudocode:

```
DIJKSTRA(graph, source):
    dist = dictionary with dist[v] = ∞ for all v in graph.vertices
    dist[source] = 0

    priority_queue = min-heap
    push (0, source) to priority_queue

    while priority_queue is not empty:
        current_dist, u = pop min from priority_queue

        if current_dist > dist[u]:
            continue
        for each neighbor v of u:
            edge_weight = weight(u, v)
            new_distance = dist[u] + edge_weight

            if new_distance < dist[v]:
                dist[v] = new_distance
                push (new_distance, v) to priority_queue

    return dist
```

Sample Input/Output:

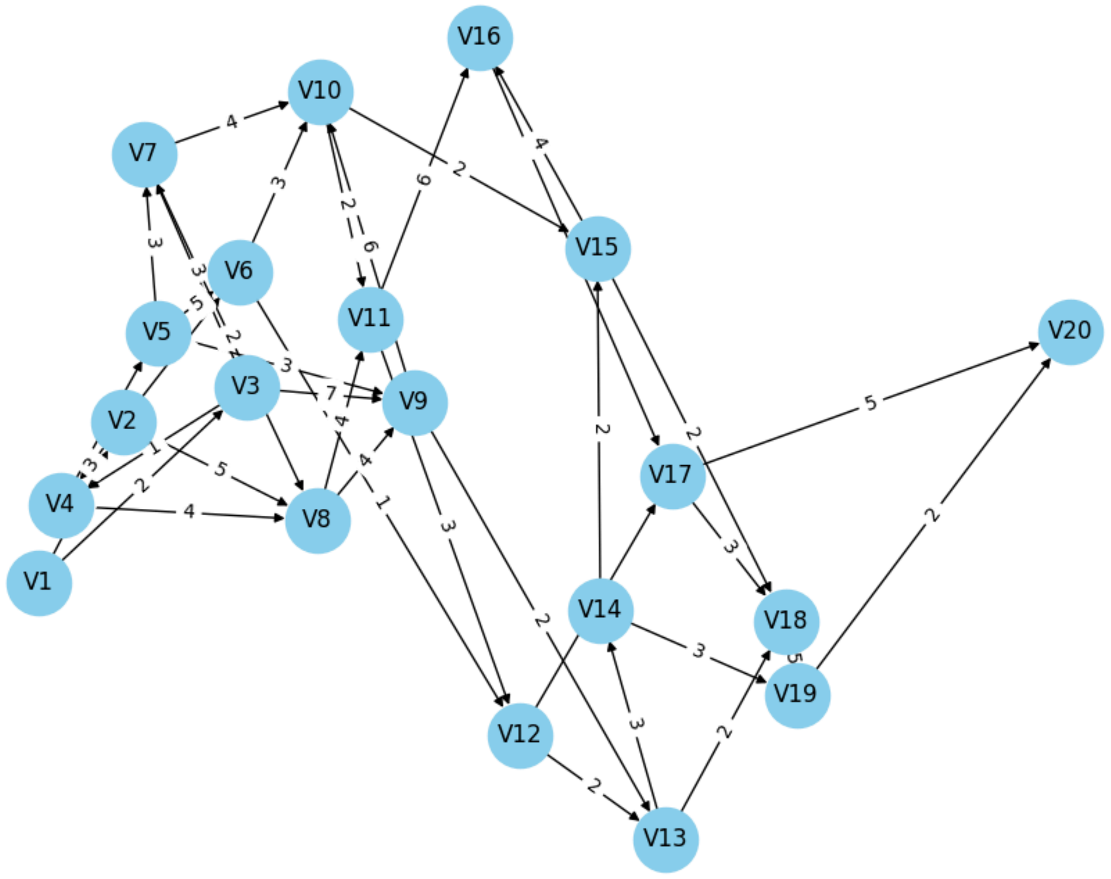
Input:

```
20 38 D
V1 V2 4
V1 V3 2
V2 V4 3
V3 V4 1
V4 V5 2
V5 V6 5
V2 V6 7
V3 V7 3
V7 V8 2
V8 V9 4
V9 V10 6
V10 V11 2
V11 V12 3
V6 V12 1
V12 V13 2
V13 V14 3
V14 V15 2
V15 V16 4
V16 V17 1
V17 V18 3
V18 V19 5
V19 V20 2
V8 V11 4
V9 V13 2
V5 V9 3
V7 V10 4
```

```
V14 V19 3
V2 V8 5
V6 V10 3
V10 V15 2
V12 V17 4
V17 V20 5
V11 V16 6
V15 V18 2
V3 V9 7
V4 V8 4
V5 V7 3
V13 V18 2
V1
```

Output:

```
V1: cost = 0, path = ['V1']
V2: cost = 4, path = ['V1', 'V2']
V3: cost = 2, path = ['V1', 'V3']
V4: cost = 3, path = ['V1', 'V3', 'V4']
V5: cost = 5, path = ['V1', 'V3', 'V4', 'V5']
V6: cost = 10, path = ['V1', 'V3', 'V4', 'V5', 'V6']
V7: cost = 5, path = ['V1', 'V3', 'V7']
V8: cost = 7, path = ['V1', 'V3', 'V4', 'V8']
V9: cost = 8, path = ['V1', 'V3', 'V4', 'V5', 'V9']
V10: cost = 9, path = ['V1', 'V3', 'V7', 'V10']
V11: cost = 11, path = ['V1', 'V3', 'V4', 'V8', 'V11']
V12: cost = 11, path = ['V1', 'V3', 'V4', 'V5', 'V6', 'V12']
V13: cost = 10, path = ['V1', 'V3', 'V4', 'V5', 'V9', 'V13']
V14: cost = 13, path = ['V1', 'V3', 'V4', 'V5', 'V9', 'V13', 'V14']
V15: cost = 11, path = ['V1', 'V3', 'V7', 'V10', 'V15']
V16: cost = 15, path = ['V1', 'V3', 'V7', 'V10', 'V15', 'V16']
V17: cost = 15, path = ['V1', 'V3', 'V4', 'V5', 'V6', 'V12', 'V17']
V18: cost = 12, path = ['V1', 'V3', 'V4', 'V5', 'V9', 'V13', 'V18']
V19: cost = 16, path = ['V1', 'V3', 'V4', 'V5', 'V9', 'V13', 'V14', 'V19']
V20: cost = 18, path = ['V1', 'V3', 'V4', 'V5', 'V9', 'V13', 'V14', 'V19', 'V20']
```



Real world applications:

- Google Maps : Finding the quickest driving route.
- Network Routing: Choosing optimal paths in data networks.

There are many limitations in Dijkstra like negative edge weights, negative weight cycles, inefficiency in dense graphs, single-source; single-destination limitation, incapable in handling dynamic graphs, not optimal for real-time path finding, etc.

In conclusion, Dijkstra is a simple shortest path searching algorithm which traverses static graphs with positive edges from source to all other vertices to find the minimum path distance.

Task 2: Minimum Spanning Tree Algorithm (Kruskal)

A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, and undirected graph is a spanning tree (no cycles and connects all vertices) that has minimum weight. Kruskal's algorithm is a greedy algorithm used to find a Minimum Spanning Tree (MST) for a connected, weighted, undirected graph.
The algorithm does so, by sorting all the edges in a non-decreasing order of their weight and picking the smallest edge. Seeing if the picked edge forms a cycle with the spanning tree formed so far. If there is no cycle formed, the edge is included. Else, it's discarded. These steps are repeated until there are (V-1) edges in the spanning tree.

Key components:

- Union-Find: For cycle detection
- Sorted Edge List: Edges which are sorted by weight
- Parent & Rank Dictionaries: For efficient union-find operations

Space complexity:

The space complexity in general of Kruskal's algorithm is O(V+E). This includes the storing of graph's edges, which requires O(E) space, and the disjoint-set data structure (Union-Find), which requires O(V) space to maintain parent and rank information for each vertex.

In our implementation of the algorithm, the space complexity analysis is as follows:
- Parent dictionary: O(V)
- Rank dictionary: O(V)
- Edge list: O(E) - stores all edges
- MST result: O(V) - spanning tree having (V-1) edges

Hence, the overall space complexity is: **O(V + E)**

Time complexity:

The time complexity of Kruskal's algorithm in general is O(ElogE) or equivalently O(ElogV), primarily due to the sorting of edges, where E is the number of edges and V is the number of vertices.

In our implementation of the algorithm, the time complexity analysis is as follows:
- O(E log E): For sorting edges
- O(E α(V)): For union find operations

Therefore, the total turns out to be: **O(E log E)** or **O(E log V)**

The worst case can be O(V^2 log E) for dense graphs.

Data Structures Used:

- Union-Find (Disjoint Set):

Provides with find() and union() operations which help in detecting cycles. This is done by checking if the two vertices of an edge belong to the same component using find(), the algorithm can determine if adding the edge would create a cycle.

- Sorted Edge List:

This data structure processes edges in increasing weight order.

- Parent & Rank Dictionaries:

Provides random access to any element with O(1) time.

Procedure:

- 1. Sort the Edges by Weight

List all the edges in the graph and sort them in ascending order of their weights.
This ensures that the edges with the smallest weights are considered first.

- 2. Initialize the Spanning Tree

Start with an empty graph containing no edges. Ensure that each vertex is treated as an independent subset (using a Union-Find data structure).

- 3. Iterate through the Edges

For each edge (starting from the smallest weight), check if adding the edge to the spanning tree creates a cycle using the Union-Find technique. If it doesn't form a cycle, add the edge to the MST. If it forms a cycle, skip the edge.

- 4. Repeat until MST is formed

Continue adding edges until the spanning tree has exactly (V - 1) edges, where V is the number of vertices.

Pseudocode:

```
KRUSKAL_MST(graph):
    parent = dictionary: parent[v] = v for all v in graph.vertices
    rank = dictionary: rank[v] = 0 for all v in graph.vertices
    sorted_edges = sort graph.edges by weight ascending

    mst_edges = []

    for each edge (u, v, weight) in sorted_edges:
        root_u = FIND(parent, u)
        root_v = FIND(parent, v)

        if root_u != root_v:
            add (u, v, weight) to mst_edges
            UNION(parent, rank, root_u, root_v)

    return mst_edges

FIND(parent, x):
    if parent[x] != x:
        parent[x] = FIND(parent, parent[x])
    return parent[x]

UNION(parent, rank, x, y):
    root_x = FIND(parent, x)
    root_y = FIND(parent, y)

    if root_x == root_y:
        return

    if rank[root_x] < rank[root_y]:
        parent[root_x] = root_y
    elif rank[root_x] > rank[root_y]:
        parent[root_y] = root_x
    else:
        parent[root_y] = root_x
        rank[root_x] += 1
```

Sample Input/Output:

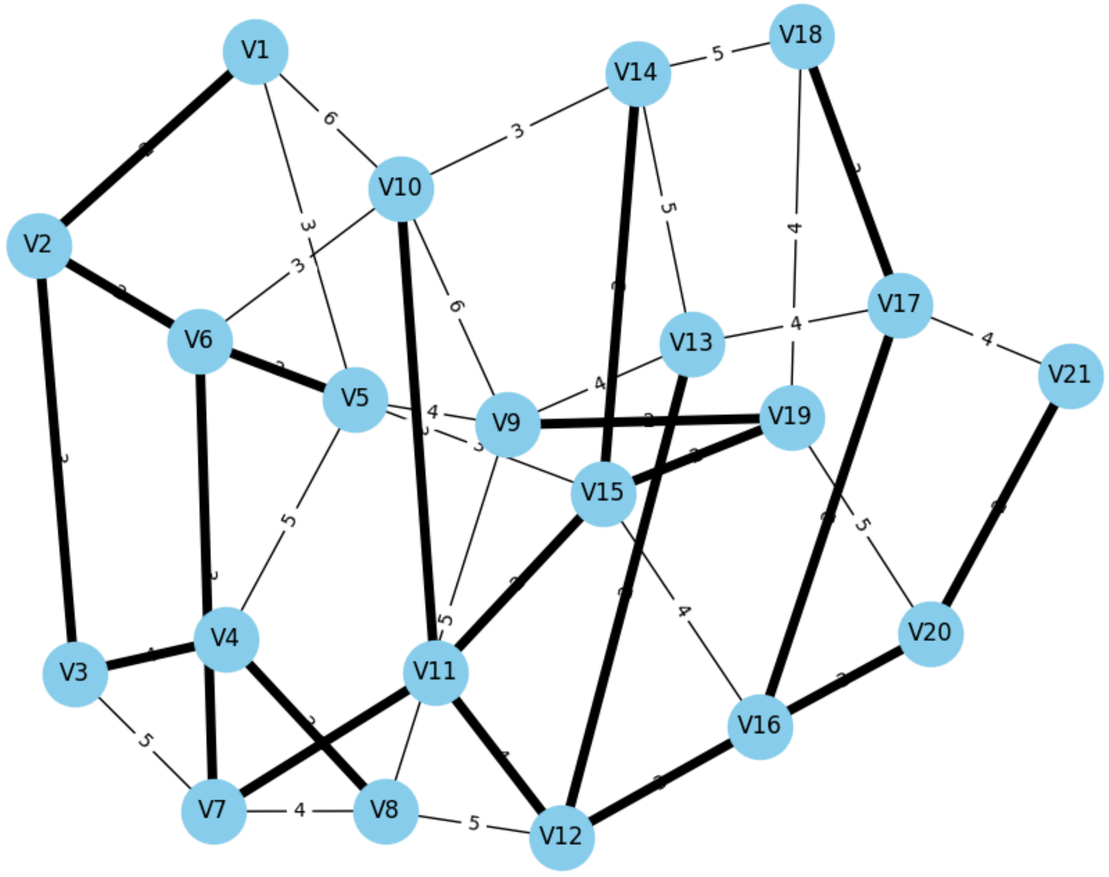
Input:

```
21 40 U
V1 V2 2
V2 V3 3
```

```
V3 V4 4
V4 V5 5
V5 V6 2
V6 V7 3
V7 V8 4
V8 V9 5
V9 V10 6
V10 V11 3
V11 V12 4
V12 V13 2
V13 V14 5
V14 V15 3
V15 V16 4
V16 V17 2
V17 V18 3
V18 V19 4
V19 V20 5
V20 V21 2
V1 V5 3
V2 V6 2
V3 V7 5
V4 V8 3
V5 V9 4
V6 V10 3
V7 V11 2
V8 V12 5
V9 V13 4
V10 V14 3
V11 V15 2
V12 V16 3
V13 V17 4
V14 V18 5
V15 V19 2
V16 V20 3
V17 V21 4
V1 V10 6
V5 V15 3
V9 V19 2
V1
```

Output:

```
Edge: V1 - V2 | Weight: 2
Edge: V5 - V6 | Weight: 2
Edge: V12 - V13 | Weight: 2
Edge: V16 - V17 | Weight: 2
Edge: V20 - V21 | Weight: 2
Edge: V2 - V6 | Weight: 2
Edge: V7 - V11 | Weight: 2
Edge: V11 - V15 | Weight: 2
Edge: V15 - V19 | Weight: 2
Edge: V9 - V19 | Weight: 2
Edge: V2 - V3 | Weight: 3
Edge: V6 - V7 | Weight: 3
Edge: V10 - V11 | Weight: 3
Edge: V14 - V15 | Weight: 3
Edge: V17 - V18 | Weight: 3
Edge: V4 - V8 | Weight: 3
Edge: V12 - V16 | Weight: 3
Edge: V16 - V20 | Weight: 3
Edge: V3 - V4 | Weight: 4
Edge: V11 - V12 | Weight: 4
Total Cost of MST: 52
```



Real world applications:

- Network Design: Kruskal's algorithm can be used to design networks with the least cost.
- Approximation Algorithms: Kruskal's algorithm can find approximate solutions to several complex optimization problems.
- Image Segmentation: Image segmentation is partitioning an image into multiple segments.
- Clustering: Clustering is grouping data points based on their similarity.

In conclusion, Kruskal's algorithm is a very vital part of graph theory and helps in tackling real-world problems efficiently. From network design to clustering in machine learning, this algorithm's applications are vast and impactful.

Task 3: DFS - Topological Sorting and Cycles

DFS, Depth First Search is used for topological sorting and cycle detection in directed graphs. In a directed acyclic graph (DAG), a topological sort is a linear ordering of vertices, while a graph with a cycle cannot be topologically sorted. A DFS-based topological sort works by adding nodes to the result list in the reverse order they finish exploring them. A cycle is detected during DFS if the algorithm encounters a node that is currently being visited (a "back edge").
The core idea of DFS is to explore as far as possible along each branch before backtracking. If the algorithm detects a node that is already on the current recursion stack, a cycle is found.
DFS can also be used to produce a topological sort if the graph is acyclic. A topological sort is a linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge (u,v); from vertex u to vertex v, u comes before v in the ordering.

Key components:

- Recursion Stack
- Visited Set
- Path Set: Tracks the current recursion path for cycle detection

Space complexity:

The auxiliary space complexity of DFS in general is dominated by the recursion stack and the visited array, both of which require O(V) space, making the overall space complexity O(V).
In our implementation of the algorithm, the space complexity analysis is as follows:

- O(V): Visited set, path set, recursion stack
- O(V+E): Graph representation

So, overall space complexity would be: **O(V + E)**

Time complexity:

The time complexity of DFS in general is O(V+E), where V is the number of vertices and E is the number of edges in the graph, each vertex and edge is visited exactly once during traversal. This complexity holds when the graph is represented using an adjacency list.
In our implementation of the algorithm, the time complexity analysis is as follows:

- Vertex processing: O(V)

- Edge traversal: O(E)
- Set operations: O(1)
- Stack operations: O(1)

Giving us overall time complexity: **O(V + E)**

Data Structures Used:

- Recursion Stack

- Visited Set:

Prevents reprocessing of vertices with O(1) time.

- Path Set:

Tracks vertices in the current path taking O(1) time for cycle detection.

Procedure:

- 1. Starting Point:

DFS begins at a chosen starting node or vertex which is ‘marked’ as visited.

- 2. Exploration:

From the current node, DFS explores an unvisited neighboring node. It continues this process, recursively, until it reaches a node with no unvisited neighbors.

- 3. Backtracking:

When a dead-end is reached, DFS backtracks to the nearest unexplored node and repeats the process.

- 4. Completing the Search:

DFS continues until all nodes have been visited, or until a specific goal has been achieved.

Pseudocode:

```
TOPOLOGICAL_SORT(graph):
   visited = set()
   stack = []
   has_cycle = false

   for each vertex v in graph.vertices:
       if v not in visited:
           DFS(v, visited, stack, set(), has_cycle)

   if has_cycle:
       return [], true
   else:
       return reversed(stack), false

DFS(vertex, visited, stack, path, has_cycle):
   visited.add(vertex)
   path.add(vertex)

   for each neighbor n of vertex:
       if n not in visited:
           DFS(n, visited, stack, path, has_cycle)
       elif n in path:
           has_cycle = true

   path.remove(vertex)
   stack.append(vertex)
```

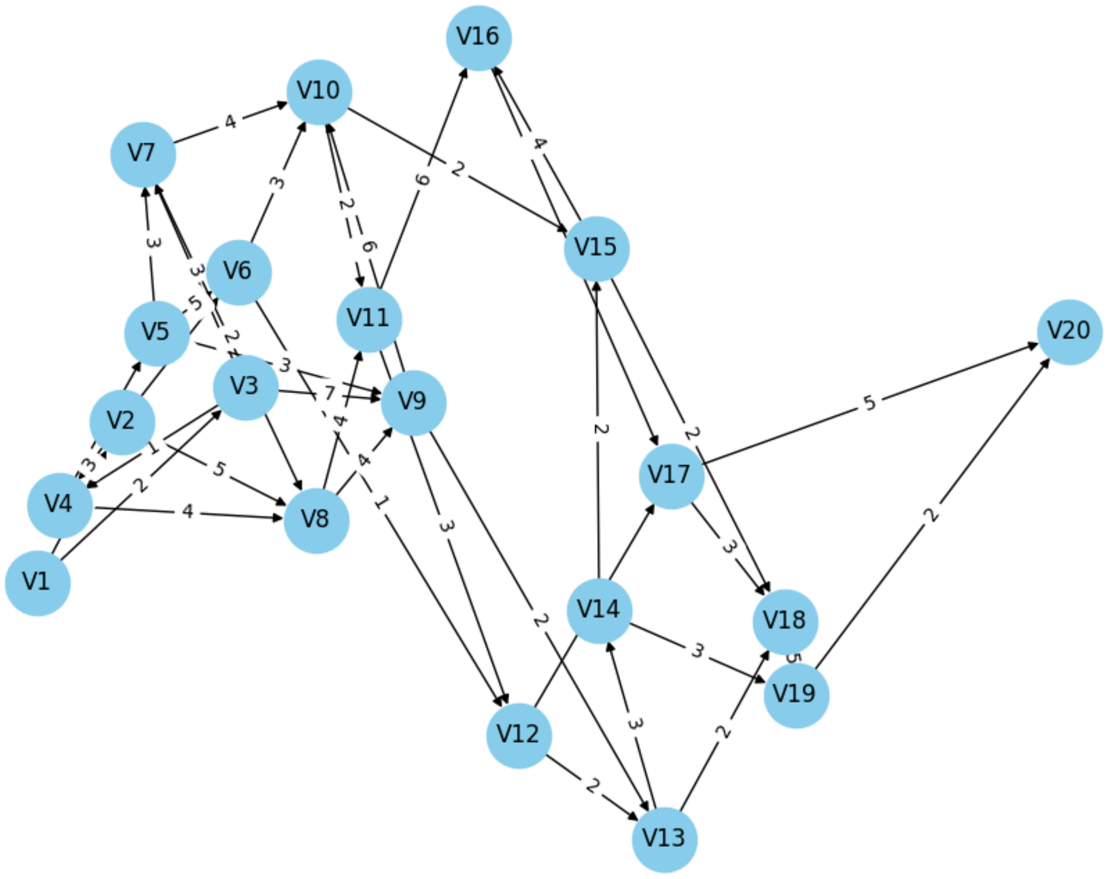
Sample Input/Output:

```
Input:
21 40 U
V1 V2 2
V2 V3 3
V3 V4 4
V4 V5 5
V5 V6 2
V6 V7 3
V7 V8 4
V8 V9 5
V9 V10 6
V10 V11 3
V11 V12 4
V12 V13 2
V13 V14 5
V14 V15 3
V15 V16 4
```

```
V16 V17 2
V17 V18 3
V18 V19 4
V19 V20 5
V20 V21 2
V1 V5 3
V2 V6 2
V3 V7 5
V4 V8 3
V5 V9 4
V6 V10 3
V7 V11 2
V8 V12 5
V9 V13 4
V10 V14 3
V11 V15 2
V12 V16 3
V13 V17 4
V14 V18 5
V15 V19 2
V16 V20 3
V17 V21 4
V1 V10 6
V5 V15 3
V9 V19 2
V1
```

Output:

```
Graph is Acyclic.
Topological Order:
V1 -> V3 -> V2 -> V4 -> V5 -> V7 -> V8 -> V9 -> V6 -> V10 -> V11 -> V12 -> V13 -> V14
-> V15 -> V16 -> V17 -> V18 -> V19 -> V20
```



Real world applications:

- Web crawlers: Depth-first search can be used in the implementation of web crawlers to explore the links on a website.
- Maze generation: Depth-first search can be used to generate random mazes.
- Backtracking: Depth-first search can be used in backtracking algorithms.

In conclusion, DFS is an algorithm which is used for traversing or searching tree or graph data structures. It explores along each branch before backtracking, making it particularly effective for tasks such as pathfinding, cycle detection, and topological sorting. The algorithm can be implemented using recursion, which leverages the call stack, or iteratively using an explicit stack data structure. In both approaches, a visited array or set is used to track nodes that have already been processed to avoid revisiting them.

Instructions To Run:

Readme.txt file

Code:

```
import networkx as nx
import matplotlib.pyplot as plt
import random
from heapq import heappush, heappop
class Graph:
    def __init__(self, vertices, edges, gtype="undirected"):
        self.vertices = vertices
        self.edges = edges
        self.gtype = gtype
        self.graph = self._build_graph()

    def _build_graph(self):
        graph = nx.DiGraph() if self.gtype == "directed" else nx.Graph()
        for u, v, w in self.edges:
            graph.add_edge(u, v, weight=w)
        return graph

def dijkstra(graph, source):
    dist = {v: float("inf") for v in graph.vertices}
    dist[source] = 0
    pq = [(0, source)]

    while pq:
        d, u = heappop(pq)
        if d > dist[u]:
            continue
        for v in graph.graph.neighbors(u):
            w = graph.graph[u][v]["weight"]
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heappush(pq, (dist[v], v))
    return dist


# ============================
# KRUSKAL MST
# ==============================
def find(parent, i):
    if parent[i] != i:
        parent[i] = find(parent, parent[i])
    return parent[i]


def union(parent, rank, x, y):
    xr, yr = find(parent, x), find(parent, y)
    if rank[xr] < rank[yr]:
        parent[xr] = yr
    elif rank[xr] > rank[yr]:
        parent[yr] = xr
    else:
        parent[yr] = xr
        rank[xr] += 1


def kruskal_mst(graph):
    edges = sorted(graph.edges, key=lambda x: x[2])
    parent, rank = {}, {}
    for v in graph.vertices:
        parent[v] = v
        rank[v] = 0

    mst = []
    for u, v, w in edges:
        x, y = find(parent, u), find(parent, v)
        if x != y:
            mst.append((u, v, w))
            union(parent, rank, x, y)
    return mst


# ==============================
# TOPOLOGICAL SORT + CYCLE DETECTION
# ==============================
def topological_sort(graph):
    visited = set()
    stack = []
    cycle = [False]

    def dfs(v, path):
        visited.add(v)
        path.add(v)
        for nbr in graph.graph.neighbors(v):
            if nbr not in visited:
                dfs(nbr, path)
            elif nbr in path:
                cycle[0] = True
        path.remove(v)
        stack.append(v)

    for v in graph.vertices:
        if v not in visited:
            dfs(v, set())

    stack.reverse()
    return stack, cycle[0]


# ==============================
# DRAW GRAPH
# ==============================
def draw_graph(graph, title, filename, highlight_edges=None):
    plt.figure(figsize=(10, 8))
    pos = nx.spring_layout(graph.graph, seed=42)
    nx.draw(
        graph.graph, pos, with_labels=True, node_color="lightblue",
        node_size=900, font_weight="bold"
    )
    labels = nx.get_edge_attributes(graph.graph, "weight")
    nx.draw_networkx_edge_labels(graph.graph, pos, edge_labels=labels)

    if highlight_edges:
        nx.draw_networkx_edges(graph.graph, pos, edgelist=highlight_edges, width=3,
edge_color="r")

    plt.title(title)
    plt.savefig(filename)
    plt.close()


# ==============================
# GRAPH GENERATORS
# ==============================
def generate_random_graph(num_nodes, num_edges, directed=False):
    vertices = [f"V{i}" for i in range(1, num_nodes + 1)]
    edges = set()
    while len(edges) < num_edges:
        u, v = random.sample(vertices, 2)
        if u != v:
            w = random.randint(1, 20)
            edges.add((u, v, w))
            if not directed:
                edges.add((v, u, w))  # ensure symmetry for undirected
    edges = list(edges)[:num_edges]
    gtype = "directed" if directed else "undirected"
    return Graph(vertices, edges, gtype)


# ==============================
# MAIN EXECUTION
# ==============================
def run_all():
    report_lines = []

    # Problem 1: Undirected graph (21 nodes, 40 edges)
    g1 = generate_random_graph(21, 40, directed=False)
    mst = kruskal_mst(g1)
    source = g1.vertices[0]
    sp = dijkstra(g1, source)
    draw_graph(g1, "Problem 1: Undirected Graph (MST)", "problem1_mst.png", mst)
    report_lines.append("=== Problem 1: Undirected Graph ===")
    report_lines.append(f"Nodes: {len(g1.vertices)}, Edges: {len(g1.edges)}")
    report_lines.append(f"Shortest Paths from {source}: {sp}")
    report_lines.append(f"MST Edges: {mst}\n")

    # Problem 2: Directed graph (19 nodes, 37 edges)
    g2 = generate_random_graph(19, 37, directed=True)
    topo, has_cycle = topological_sort(g2)
    source = g2.vertices[0]
    sp = dijkstra(g2, source)
    draw_graph(g2, "Problem 2: Directed Graph (Topo Sort)", "problem2_topo.png")
    report_lines.append("=== Problem 2: Directed Graph ===")
    report_lines.append(f"Nodes: {len(g2.vertices)}, Edges: {len(g2.edges)}")
    report_lines.append(f"Shortest Paths from {source}: {sp}")
    report_lines.append(f"Topological Order: {topo}")
    report_lines.append(f"Contains Cycle: {has_cycle}\n")

    # Problem 3: Directed graph (19 nodes, 37 edges)
    g3 = generate_random_graph(19, 37, directed=True)
    topo, has_cycle = topological_sort(g3)
    source = g3.vertices[0]
    sp = dijkstra(g3, source)
    draw_graph(g3, "Problem 3: Directed Graph (Cycle Detection)",
"problem3_cycle.png")
    report_lines.append("=== Problem 3: Directed Graph ===")
    report_lines.append(f"Nodes: {len(g3.vertices)}, Edges: {len(g3.edges)}")
    report_lines.append(f"Shortest Paths from {source}: {sp}")
    report_lines.append(f"Topological Order: {topo}")
    report_lines.append(f"Contains Cycle: {has_cycle}\n")

    with open("report.txt", "w") as f:
        f.write("\n".join(report_lines))

    print("\n✅ All problems executed successfully!")
    print("PNG files: problem1_mst.png, problem2_topo.png, problem3_cycle.png")
    print("Text report: report.txt")


# ==============================
# ENTRY POINT
# ==============================
if __name__ == "__main__":
    random.seed(42)  # reproducible results
    run_all()
```