Assignment 1: Graph and Tree Algorithms in Python

Course: Design and analysis of an algorithms

Date: 12, June, 2025

---

1. Introduction to Graph and Tree-Based Algorithms

Graphs and trees are foundational structures in computer science used to model relationships, networks, and hierarchies. Algorithms that operate on these structures help solve real-world problems like routing, networking, data compression, and search.

- **Graph:** A set of vertices (nodes) and edges (connections).
- **Tree:** A special kind of graph with no cycles and exactly one path between any two nodes.

We will implement and understand the following algorithms:

- Warshall's Algorithm
- Floyd's Algorithm
- Prim's Algorithm
- Kruskal's Algorithm
- Dijkstra's Algorithm
- Huffman Tree Algorithm

Each algorithm will be explained in simple language, with visual examples and Python code.

What is an Algorithm?

An **algorithm** is a clearly defined, step-by-step process or set of rules to solve a specific problem or perform a task. Algorithms take an input, follow a sequence of steps, and produce an output.

Why Are Algorithms Important?

Algorithms are the backbone of computer science. They are essential for:

- **Efficiency**: Ensuring tasks are done in the fastest or most resource-effective way.
- **Automation**: Powering tasks behind the scenes in apps, websites, and devices.
- **Problem-Solving**: Providing structured approaches to complex problems.

How Are They Used in Computing Technologies?

Algorithms power nearly everything in modern computing:

- **Search Engines** (e.g., Google Search ranking)
- **Recommendation Systems** (e.g., Netflix, YouTube suggestions)
- **Data Compression** (e.g., ZIP files, JPEG images)
- **Routing and GPS** (e.g., finding the fastest path)
- **Cybersecurity** (e.g., encryption algorithms)
- **Artificial Intelligence** (e.g., decision trees, neural networks)

In essence, algorithms are silently at work in nearly every digital interaction we have today.

---

2. Warshall's Algorithm (Transitive Closure)

Definition:

Warshall's Algorithm is a method used in graph theory to compute the **transitive closure** of a directed graph. It tells us whether one vertex is **reachable** from another, considering all possible intermediate paths. In simple terms, it helps determine if there's a way to get from one node to another, directly or indirectly.

Goal:

Find out if there's a path between every pair of vertices in a graph.

Step-by-Step:

1. Represent the graph as an adjacency matrix (1 if there's a direct edge, 0 otherwise).
2. Use 3 nested loops:
    - Loop `k` through all vertices.
    - For every pair `(i, j)`, set `path[i][j] = path[i][j] or (path[i][k] and path[k][j])`
3. This checks if you can go from `i` to `j` through `k`.

Real-World Applications:

- **Social Network Analysis**: Determine whether a person is connected to another through mutual friends.
- **Web Crawlers**: Check which websites are reachable from a given site.
- **Database Optimization**: Evaluate indirect relationships or dependencies between data entities.
- **Access Control Systems**: Check permission inheritance in organizational hierarchies.

Example:

Matrix:

```
  0 1 2
0 0 1 0
1 0 0 1
2 0 0 0
```

After Warshall:

```
  0 1 2
0 0 1 1
1 0 0 1
2 0 0 0
```

Python Code:

```
import numpy as np

def warshall(matrix):
    n = len(matrix)
    reach = np.array(matrix)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                reach[i][j] = reach[i][j] or (reach[i][k] and
reach[k][j])
    return reach

adj_matrix = [
    [0, 1, 0],
    [0, 0, 1],
    [0, 0, 0]
]

print(warshall(adj_matrix))
```

---

3. Floyd's Algorithm (All-Pairs Shortest Path)

Definition:

Floyd's Algorithm, also known as the Floyd-Warshall algorithm, is a method used to find the **shortest paths between all pairs of vertices** in a weighted graph. It works even when the graph has negative edge weights (but no negative cycles). It systematically checks if going through an intermediate node gives a shorter path between two nodes.

Goal:

Find the shortest paths between every pair of nodes.

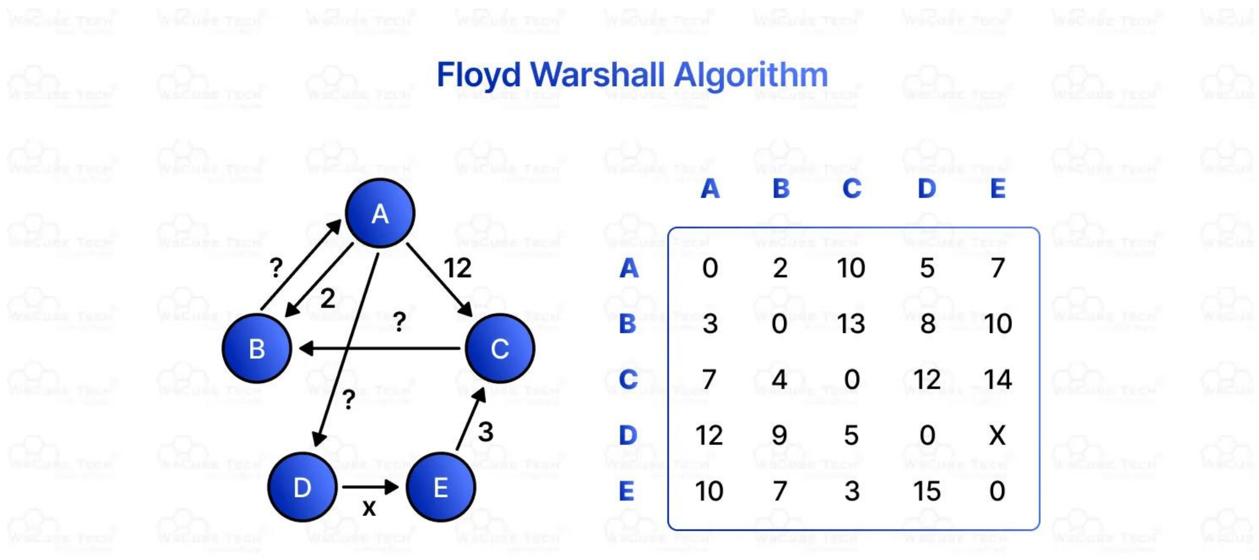
Step-by-Step:

1. Use a distance matrix: `dist[i][j]` is the distance from `i` to `j`.
2. For every vertex `k`, check if the path `i -> k -> j` is shorter than `i -> j`.
3.

Real-World Applications:

- **Transportation Networks**: Calculate shortest travel routes between all cities.
- **Routing Protocols**: Used in dynamic routing algorithms like OSPF in computer networks.
- **Telecommunication Systems**: Optimize signal path between stations.
- **Logistics and Delivery**: Plan efficient multi-drop delivery paths for couriers.

Example:



	A	B	C	D	E
A	0	2	10	5	7
B	3	0	13	8	10
C	7	4	0	12	14
D	12	9	5	0	X
E	10	7	3	15	0

Python Code:

```python
INF = float('inf')

def floyd(graph):
    n = len(graph)
    dist = [row[:] for row in graph]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist

graph = [
    [0, 3, INF],
    [INF, 0, 1],
    [INF, INF, 0]
]

print(floyd(graph))
```

---

4. Prim's Algorithm (Minimum Spanning Tree)

Definition:

Prim's Algorithm is a greedy algorithm used to find the **Minimum Spanning Tree (MST)** of a connected, undirected graph. It starts with a single node and repeatedly adds the smallest edge that connects a visited node to an unvisited node. This ensures all nodes are connected with the minimal possible total edge weight.

Goal:

Find the cheapest way to connect all nodes in an undirected graph.

Step-by-Step:

1. Start with one node.
2. Always pick the edge with the smallest weight that connects a new node.
3. Repeat until all nodes are included.

Real-World Applications:

- **Electric Grid Design**: Efficiently connect power stations and transformers with minimal wiring cost.
- **Computer Networking**: Laying out cables in a network without cycles while minimizing length and cost.

- **Telecommunication Towers**: Connecting cellular towers to form a cost-effective infrastructure.
- **Pipeline Distribution**: Building water or gas pipelines in rural areas with minimal cost and redundancy.

Python Code:

```
import heapq

def prim(graph):
    n = len(graph)
    visited = [False] * n
    min_heap = [(0, 0)]  # (cost, node)
    total_cost = 0

    while min_heap:
        cost, u = heapq.heappop(min_heap)
        if visited[u]:
            continue
        visited[u] = True
        total_cost += cost
        for v, weight in enumerate(graph[u]):
            if not visited[v] and weight != 0:
                heapq.heappush(min_heap, (weight, v))
    return total_cost

graph = [
    [0, 2, 0, 6, 0],
    [2, 0, 3, 8, 5],
    [0, 3, 0, 0, 7],
    [6, 8, 0, 0, 9],
    [0, 5, 7, 9, 0]
]

print(prim(graph))
```

---

5. Kruskal's Algorithm (Minimum Spanning Tree)

Definition:

Kruskal's Algorithm is a greedy method to find the **Minimum Spanning Tree (MST)** of a graph by sorting all edges by their weight and adding the smallest edge that doesn't form a cycle. It uses the Union-Find data structure to efficiently check and merge disjoint sets of nodes.

Goal:

Also finds the cheapest way to connect all nodes but by sorting edges.

Step-by-Step:

1. Sort all edges by weight.
2. Use Union-Find to avoid cycles.
3. Add edge if it connects two different sets.

Real-World Applications:

- **Network Design**: Laying out a minimal-cost infrastructure such as broadband or fiber optic networks without loops.
- **Road Construction**: Connecting rural villages with minimum road length and construction cost.
- **Clustering Algorithms**: Used in machine learning to build clusters based on distance or similarity.
- **Electrical Grid Setup**: Laying power lines while ensuring minimal total wiring cost and avoiding loops.

Python Code:

```
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            self.parent[root_v] = root_u
            return True
        return False

def kruskal(n, edges):
    uf = UnionFind(n)
    edges.sort(key=lambda x: x[2])
    mst_cost = 0
    for u, v, weight in edges:
        if uf.union(u, v):
            mst_cost += weight
    return mst_cost

edges = [
    (0, 1, 10),
    (0, 2, 6),
    (0, 3, 5),
    (1, 3, 15),
    (2, 3, 4)
] print(kruskal(4, edges))
```

6. Dijkstra's Algorithm (Single-Source Shortest Path)

Definition:

Dijkstra's Algorithm is a greedy algorithm used to compute the **shortest path from a single source node to all other nodes** in a graph with non-negative weights. It explores nodes in order of increasing distance from the source using a priority queue.

Goal:

Find the shortest path from one node to all others.

Step-by-Step:

1. Start with source node, set distance to 0.
2. Use priority queue to explore the nearest unvisited node.
3. Update distance to its neighbors if shorter path found.

Real-World Applications:

- **Navigation Systems**: GPS applications like Google Maps use Dijkstra’s Algorithm to find the shortest route.
- **Network Routing Protocols**: Used in algorithms such as OSPF to find optimal paths for data packets.
- **Emergency Services**: Determine the quickest route for ambulances or fire trucks.
- **Game AI**: Help non-player characters (NPCs) move optimally in gaming environments.

Python Code:

```
import heapq

def dijkstra(graph, start):
    n = len(graph)
    dist = [float('inf')] * n
    dist[start] = 0
    pq = [(0, start)]

    while pq:
        d, u = heapq.heappop(pq)
        for v, w in enumerate(graph[u]):
            if w != 0 and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))
    return dist

graph = [
    [0, 4, 0, 0, 0, 0, 0, 8, 0],
    [4, 0, 8, 0, 0, 0, 0, 11, 0],
    [0, 8, 0, 7, 0, 4, 0, 0, 2],
    [0, 0, 7, 0, 9, 14, 0, 0, 0],
    [0, 0, 0, 9, 0, 10, 0, 0, 0],
    [0, 0, 4, 14, 10, 0, 2, 0, 0],
    [0, 0, 0, 0, 0, 2, 0, 1, 6],
    [8, 11, 0, 0, 0, 0, 1, 0, 7],
    [0, 0, 2, 0, 0, 0, 6, 7, 0]
]

print(dijkstra(graph, 0))
```

---

7. Huffman Tree Algorithm (Data Compression)

Definition:

Huffman Tree Algorithm is a greedy algorithm used to create an **optimal prefix code** for lossless data compression. Characters with higher frequencies get shorter codes, and those with lower frequencies get longer codes, reducing the total number of bits needed.

Goal:

Build an optimal prefix code (binary) to compress data.

Step-by-Step:

1. Count frequency of each character.
2. Build a min-heap of trees.
3. Merge two smallest trees repeatedly.

Real-World Applications:

- **File Compression**: Used in ZIP, RAR, and other archival formats to compress text and binary files efficiently.
- **Multimedia Encoding**: JPEG image compression and MP3 audio compression rely on Huffman coding to reduce file sizes without losing quality.
- **Data Transmission**: Minimize the amount of data transferred over networks by encoding messages efficiently.
- **Compiler Design**: Used in syntax tree representations and encoding symbol tables for faster access.

Python Code:

```
import heapq

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(freq_map):
    heap = [Node(char, freq) for char, freq in freq_map.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]

def generate_codes(node, code="", code_map={}):
    if node:
        if node.char:
            code_map[node.char] = code
        generate_codes(node.left, code + "0", code_map)
        generate_codes(node.right, code + "1", code_map)
    return code_map

freq_map = {'a': 5, 'b': 9, 'c': 12, 'd': 13, 'e': 16, 'f': 45}
root = build_huffman_tree(freq_map)
codes = generate_codes(root)
print(codes)
```

8. Conclusion

- **Warshall:** Finds connectivity between all pairs.
- **Floyd:** Finds shortest distance between all pairs.
- **Prim & Kruskal:** Minimum spanning tree from different strategies.
- **Dijkstra:** Best path from one node to others.
- **Huffman:** Best compression strategy using tree structure.

These algorithms are not just textbook exercises—they are practical tools at the heart of modern computing. From designing communication networks to enabling efficient web search, from building road maps to compressing files and streaming videos, the knowledge and implementation of these algorithms directly affect how systems work around us.

Understanding them builds the foundation not only for solving abstract problems, but for crafting real-world technologies that scale, adapt, and serve millions. These tools give structure to complexity, find efficiency where chaos hides, and unlock smarter systems across domains. Mastery of these algorithms is therefore not only an academic milestone—but a professional superpower.

---

9. References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*.
- Python Documentation
- GeeksforGeeks, TutorialsPoint, Programiz
