BSc (Hons) in Software Engineering

IT3126: Graph theory

Week Number 08

Trees and distance

Learning Outcomes

- After completing this course and the essential reading and activities, student should be able to,
  - LO1:Apply basic notations of graph theory and fundamental theorems in graph theory
  - LO2:formulate and prove central theorems about trees, matching, connectivity, coloring and planar graphs;
  - LO3:describe and apply some basic algorithms for graphs;
  - LO4:use graph theory as a modelling tool.

Aims of the Lesson

- To introduce the fundamental graph theory topics and results.
- To be familiarize with the techniques for proofs and analysis.

Tree

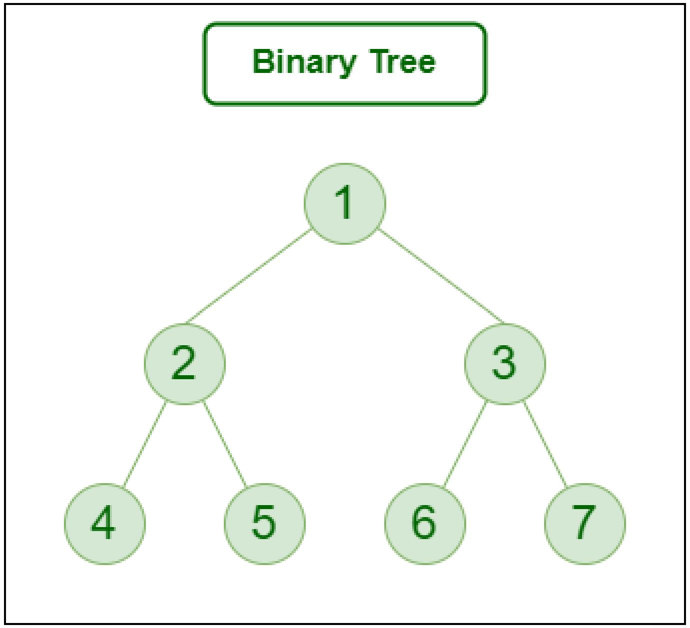
- In graph theory, trees are a specific type of graph that doesn't contain any cycles.
- A tree is a connected graph where there is exactly one path between any two nodes (also called vertices), and there are no repeating paths or loops.

Distance

- Distance in the context of trees in graph theory typically refers to the length of the shortest path between two nodes within the tree.
- The distance between two nodes in a tree is defined as the number of edges along the shortest path that connects those nodes.

Binary trees

- A *binary tree* is either empty, or it consists of a node called the *root* together with two binary trees called the *left subtree* and the *right subtree* of the root.



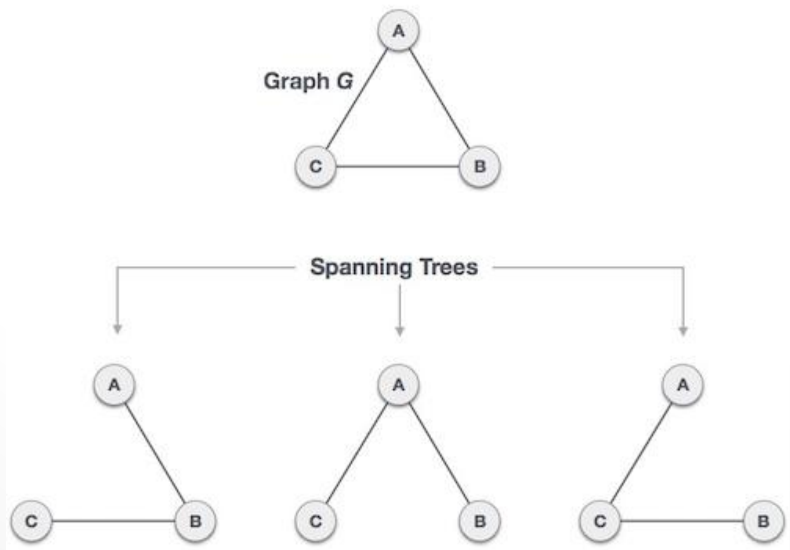
Traversal of binary trees

- One of the most important operations on a binary tree is **traversal**, moving through all the nodes of the binary tree, visiting each one in turn.

- There are three standard traversal orders:
  - With ***preorder*** *traversal* we first visit a node, then traverse its left subtree, and then traverse its right subtree. (VLR)
  - With ***inorder*** *traversal* we first traverse the left subtree, then visit the node, and then traverse its right subtree. (LVR)
  - With ***postorder*** *traversal* we first traverse the left subtree, then traverse the right subtree, and finally visit the node. (LRV)

Spanning tree

- A spanning tree of a graph is a subgraph that includes all the vertices of the original graph while being a tree itself.
- In other words, it's a connected acyclic subgraph that spans all the vertices of the original graph.
- A graph may have multiple spanning trees.



Spanning tree

- Key Properties of Spanning Trees:
- **Connectedness**: A spanning tree must connect all the vertices of the original graph.
- **Acyclic**: A spanning tree must not contain any cycles. Adding any edge to the spanning tree would create a cycle.
- **V-1 Edges**: If the original graph has 'V' vertices, a spanning tree will have exactly 'V-1' edges.
- **No Redundant Edges**: A spanning tree includes the minimum number of edges required to connect all the vertices. There are no redundant edges.
- **Subgraph**: A spanning tree is a subgraph of the original graph.

Minimum spanning tree

- A minimum spanning tree (MST) is a subset of edges of an undirected graph that connects all the vertices together without forming any cycles and has the minimum possible total edge weight.
- In other words, an MST is a tree that spans all the vertices of the graph while minimizing the sum of edge weights.

Prim's algorithm

- Prim's algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a connected, undirected graph.
- The goal of the algorithm is to find the subset of edges that connects all the vertices in the graph while minimizing the total edge weight.

- Initialization:
  - Start with an empty set **MSTSet** which initially contains no vertices.
  - Create an array **key[]** to store the minimum key values for each vertex. Initialize all key values to a large number (infinity).
  - Create an array **parent[]** to store the parent of each vertex in the MST. Initialize all parent values to -1 (no parent).

Prim’s algorithm

- Initialization:
  - Start with an empty set **MSTSet** which initially contains no vertices.
  - Create an array **key[]** to store the minimum key values for each vertex. Initialize all key values to a large number (infinity).
  - Create an array **parent[]** to store the parent of each vertex in the MST. Initialize all parent values to -1 (no parent).

- Select a Starting Vertex:
  - Choose any vertex as the starting point of the MST. Assign its key value to 0 to ensure it's selected first.

Prim's algorithm

- Grow the MST:
  - Repeat the following steps until all vertices are included in the MST:
  - a. Find the vertex u with the minimum key value among the vertices not yet in the MSTSet.
  - b. Add vertex u to the MSTSet.
  - c. For each adjacent vertex v of u that is not in the MSTSet, update its key value if the weight of the edge between u and v is less than its current key value. Set parent[v] to u.
- Print the MST:
  - The edges of the MST are given by the pairs (parent[v], v) for all vertices except the starting vertex. The edge weight is given by the corresponding key value.

Prim's algorithm

```
import java.util.LinkedList;

class Node {
    int vertex;
    int weight;
    Node next;

    public Node(int vertex, int weight) {
        this.vertex = vertex;
        this.weight = weight;
        this.next = null;
    }
}

class Graph {
    int numVertices;
    LinkedList<Node>[] adjList;

    public Graph(int numVertices) {
        this.numVertices = numVertices;
        adjList = new LinkedList[numVertices];
        for (int i = 0; i < numVertices; i++) {
            adjList[i] = new LinkedList<>();
        }
    }
```

Prim's algorithm

```
void addEdge(int src, int dest, int weight) {
    Node newNode = new Node(dest, weight);
    adjList[src].add(newNode);

    newNode = new Node(src, weight); // Add reverse edge for undirected
        graph
    adjList[dest].add(newNode);
}

int minKey(int[] key, boolean[] mstSet) {
    int min = Integer.MAX_VALUE, minIndex = -1;
    for (int v = 0; v < numVertices; v++) {
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }
    return minIndex;
}
```

Prim's algorithm

```
void printMST(int[] parent) {
    System.out.println("Edge \tWeight");
    for (int i = 1; i < numVertices; i++) {
        int u = parent[i];
        for (Node node : adjList[u]) {
            if (node.vertex == i) {
                System.out.println(u + " - " + i + "\t" + node.weight);
                break;
            }
        }
    }
}

void primMST() {
    int[] parent = new int[numVertices]; // Array to store the constructed
        MST
    int[] key = new int[numVertices];    // Key values used to pick
        minimum weight edge
    boolean[] mstSet = new boolean[numVertices]; // To represent the set
        of vertices included in MST

    for (int i = 0; i < numVertices; i++) {
        key[i] = Integer.MAX_VALUE;
        mstSet[i] = false;
    }
```

Prim's algorithm

```
        key[0] = 0;       // Start from the first vertex
        parent[0] = -1;   // First vertex is the root of MST

        for (int count = 0; count < numVertices - 1; count++) {
            int u = minKey(key, mstSet);
            mstSet[u] = true;

            for (Node current : adjList[u]) {
                int v = current.vertex;
                int weight = current.weight;

                if (!mstSet[v] && weight < key[v]) {
                    parent[v] = u;
                    key[v] = weight;
                }
            }
        }
        printMST(parent);
    }
}
```

Prim's algorithm

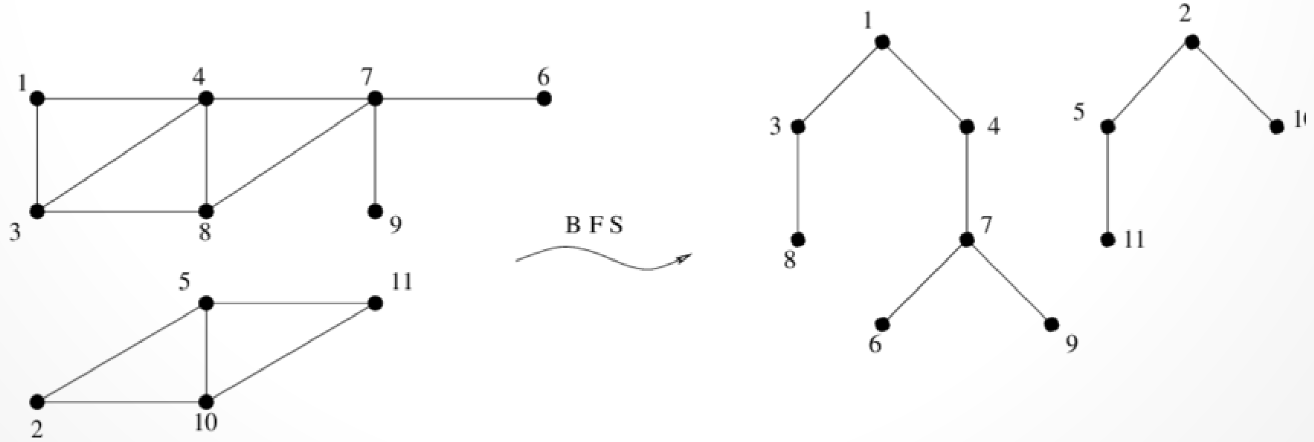
```java
public class Main {
    public static void main(String[] args) {
        int V = 5; // Number of vertices
        Graph graph = new Graph(V);

        graph.addEdge(0, 1, 2);
        graph.addEdge(0, 3, 6);
        graph.addEdge(1, 2, 3);
        graph.addEdge(1, 3, 8);
        graph.addEdge(1, 4, 5);
        graph.addEdge(2, 4, 7);
        graph.addEdge(3, 4, 9);

        graph.primMST();
    }
}
```

Breadth-first search forest

- The BFS forest consists of multiple BFS trees, each rooted at a different vertex.
- All vertices in the graph are part of exactly one BFS tree in the forest.
- The BFS trees are disjoint, meaning they do not share any vertices.
- Each BFS tree spans all the vertices that are reachable from its root vertex.
- The depth of each vertex in its corresponding BFS tree indicates the shortest path distance from the root vertex.

Breadth-first search forest

- BFS forests are commonly used to explore and analyze the connectivity and distance relationships within a graph.
- They are especially useful in finding shortest paths and understanding the structure of a graph's components.



Breadth-first search forest

```
import java.util.LinkedList;
import java.util.Queue;

class Node {
    int vertex;
    Node next;

    public Node(int vertex) {
        this.vertex = vertex;
        this.next = null;
    }
}

class Graph {
    int numVertices;
    LinkedList<Node>[] adjacencyList;

    public Graph(int numVertices) {
        this.numVertices = numVertices;
        adjacencyList = new LinkedList[numVertices];
        for (int i = 0; i < numVertices; i++) {
            adjacencyList[i] = new LinkedList<>();
        }
    }
```

Breadth-first search forest

```
void addEdge(int source, int destination) {
    Node newNode = new Node(destination);
    adjacencyList[source].add(newNode);

    // Since it's an undirected graph, add an edge in the other direction
       as well
    newNode = new Node(source);
    adjacencyList[destination].add(newNode);
}

void bfs(int startVertex, boolean[] visited) {
    // Create a queue for BFS traversal
    Queue<Integer> queue = new LinkedList<>();

    // Enqueue the starting vertex
    queue.add(startVertex);
    visited[startVertex] = true;

    System.out.println("BFS Tree rooted at vertex " + startVertex + ":");

    while (!queue.isEmpty()) {
        int currentVertex = queue.poll();
        System.out.print(currentVertex + " ");
```

Breadth-first search forest

```
            for (Node neighbor : adjacencyList[currentVertex]) {
                int neighborVertex = neighbor.vertex;
                if (!visited[neighborVertex]) {
                    queue.add(neighborVertex);
                    visited[neighborVertex] = true;
                }
            }
        }
        System.out.println();
    }

    void bfsForest() {
        boolean[] visited = new boolean[numVertices];

        for (int i = 0; i < numVertices; i++) {
            if (!visited[i]) {
                bfs(i, visited);
            }
        }
    }
}
```

Breadth-first search forest

```
public class Main {
    public static void main(String[] args) {
        int numVertices = 5;
        Graph graph = new Graph(numVertices);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 2);
        graph.addEdge(3, 4);

        graph.bfsForest();
    }
}
```

Eccentricity of a vertex

- In graph theory, the eccentricity of a vertex is a measure of how far that vertex is from the vertex farthest away from it.
- It's often used to analyze the structure of a graph and its vertices.
- The eccentricity of a vertex v, denoted as ε(v), is the maximum shortest path distance between vertex v and any other vertex in the graph.

Eccentricity of a vertex

```java
class Node {
    int vertex;
    Node next;

    Node(int vertex) {
        this.vertex = vertex;
        this.next = null;
    }
}
```

```java
class LinkedList {
    Node head;

    LinkedList() {
        this.head = null;
    }

    void add(Node newNode) {
        if (head == null) {
            head = newNode;
        } else {
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
        }
    }
}
```

Eccentricity of a vertex

```
class Graph {
    int numVertices;
    LinkedList[] adjacencyList;

    Graph(int numVertices) {
        this.numVertices = numVertices;
        adjacencyList = new LinkedList[numVertices];
        for (int i = 0; i < numVertices; i++) {
            adjacencyList[i] = new LinkedList();
        }
    }

    void addEdge(int source, int destination) {
        Node newNode = new Node(destination);
        adjacencyList[source].add(newNode);

        // Assuming it's an undirected graph
        newNode = new Node(source);
        adjacencyList[destination].add(newNode);
    }
}
```

Eccentricity of a vertex

```
public class Main {
    static final int MAX_VERTICES = 100;

    static void initializeGraph(Graph graph, int numVertices) {
        graph.numVertices = numVertices;
        for (int i = 0; i < numVertices; i++) {
            graph.adjacencyList[i] = new LinkedList();
        }
    }

    static int bfsEccentricity(Graph graph, int startVertex) {
        int[] dist = new int[MAX_VERTICES];
        boolean[] visited = new boolean[MAX_VERTICES];
        Arrays.fill(dist, Integer.MAX_VALUE);
        Arrays.fill(visited, false);
        dist[startVertex] = 0;
        visited[startVertex] = true;
        int[] queue = new int[MAX_VERTICES];
        int front = 0, rear = 0;
        queue[rear++] = startVertex;
```

Eccentricity of a vertex

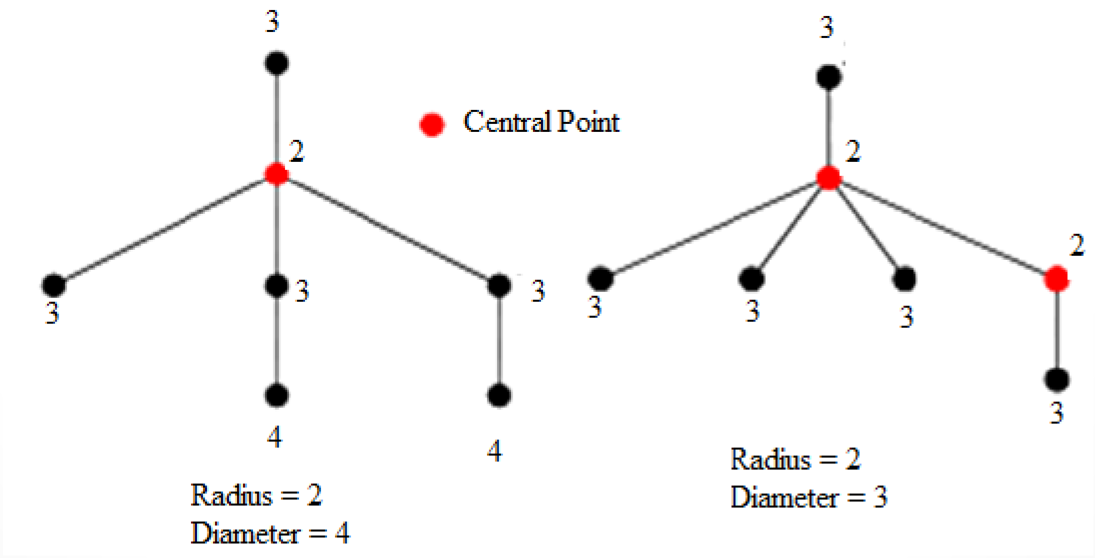
```java
static int bfsEccentricity(Graph graph, int startVertex) {
    int[] dist = new int[MAX_VERTICES];
    boolean[] visited = new boolean[MAX_VERTICES];
    Arrays.fill(dist, Integer.MAX_VALUE);
    Arrays.fill(visited, false);
    dist[startVertex] = 0;
    visited[startVertex] = true;
    int[] queue = new int[MAX_VERTICES];
    int front = 0, rear = 0;
    queue[rear++] = startVertex;

    while (front < rear) {
        int currentVertex = queue[front++];
        LinkedList current = graph.adjacencyList[currentVertex];
        Node temp = current.head;
        while (temp != null) {
            int neighbor = temp.vertex;
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                dist[neighbor] = dist[currentVertex] + 1;
                queue[rear++] = neighbor;
            }
            temp = temp.next;
        }
    }
```

Eccentricity of a vertex

```
    public static void main(String[] args) {
        int numVertices = 7;
        Graph graph = new Graph(numVertices);

        initializeGraph(graph, numVertices);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 2);
        graph.addEdge(2, 3);
        graph.addEdge(2, 4);
        graph.addEdge(3, 5);
        graph.addEdge(4, 6);

        int vertex = 0; // Choose the vertex whose eccentricity you want to
            find
        int eccentricity = bfsEccentricity(graph, vertex);
        System.out.printf("Eccentricity of vertex %d: %d\n", vertex,
            eccentricity);
    }
}
```

Diameter, radius & center of a graph

- In graph theory, the terms "diameter," "radius," and "center" refer to measures that provide insights into the structure of a graph.



Diameter

- The diameter of a graph is the longest shortest path distance between any two vertices in the graph.
- In other words, it represents the maximum distance between any pair of vertices.
- Calculating the diameter is often used to understand the "spread" or "extent" of the graph.

Radius

- The radius of a graph is the minimum eccentricity among all vertices in the graph.
- It represents the "centermost" part of the graph, as it indicates the smallest maximum distance from any vertex to all other vertices.

Center

- The center of a graph is the set of vertices with eccentricity equal to the radius.
- In other words, it's the set of vertices that are at the "center" of the graph in terms of their distance from other vertices.

Diameter, radius & center of a graph

```java
public class Main {
    static final int MAX_VERTICES = 100;

    static void initializeGraph(Graph graph, int numVertices) {
        graph.numVertices = numVertices;
        for (int i = 0; i < numVertices; i++) {
            graph.adjacencyList[i] = new LinkedList();
        }
    }

    public static void main(String[] args) {
        int numVertices = 7;
        Graph graph = new Graph(numVertices);

        initializeGraph(graph, numVertices);
```

Diameter, radius & center of a graph

```
graph.addEdge(0, 1);
graph.addEdge(0, 2);
graph.addEdge(1, 2);
graph.addEdge(2, 3);
graph.addEdge(2, 4);
graph.addEdge(3, 5);
graph.addEdge(4, 6);

int diameter = 0;
int radius = Integer.MAX_VALUE;
List<Integer> centers = new ArrayList<>();

for (int vertex = 0; vertex < numVertices; vertex++) {
    int eccentricity = graph.bfsEccentricity(vertex);

    if (eccentricity > diameter) {
        diameter = eccentricity;
    }
```

Diameter, radius & center of a graph

```
                if (eccentricity < radius) {
                    radius = eccentricity;
                    centers.clear();
                    centers.add(vertex);
                } else if (eccentricity == radius) {
                    centers.add(vertex);
                }
            }

            System.out.println("Diameter: " + diameter);
            System.out.println("Radius: " + radius);
            System.out.print("Center(s): ");
            for (int center : centers) {
                System.out.print(center + " ");
            }
            System.out.println();
        }
    }
```

Shortest path problem

- The shortest path problem involves finding the shortest path between two vertices in a graph, where the length of a path is determined by the sum of the weights (or costs) of the edges along the path.
- There are several algorithms to solve this problem, with Dijkstra's algorithm and Bellman-Ford algorithm being two common approaches.

Dijkstra's Algorithm

- Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edge weights.
- **Algorithm Steps**:
  - Create a set of unvisited vertices and initialize the distances to all vertices as infinity (except the source vertex which is 0).
  - While there are unvisited vertices, choose the vertex with the minimum distance.
  - For the chosen vertex, update the distances of its neighbors by considering the weight of the edge and the distance to the chosen vertex.
  - Mark the chosen vertex as visited.

Dijkstra's Algorithm

```
import java.util.*;

class Node {
    int vertex;
    int weight;
    Node next;

    Node(int vertex, int weight) {
        this.vertex = vertex;
        this.weight = weight;
        this.next = null;
    }
}

class Graph {
    final int V;
    Node[] adjacencyList;

    Graph(int V) {
        this.V = V;
        adjacencyList = new Node[V];
        for (int i = 0; i < V; i++) {
            adjacencyList[i] = null;
        }
    }
```

Dijkstra's Algorithm

```java
    void addEdge(int src, int dest, int weight) {
        Node newNode = new Node(dest, weight);
        newNode.next = adjacencyList[src];
        adjacencyList[src] = newNode;
    }
}

public class DijkstraAlgorithm {
    static final int V = 5; // Number of vertices

    static int minDistance(int dist[], boolean visited[]) {
        int min = Integer.MAX_VALUE, minIndex = -1;
        for (int v = 0; v < V; v++) {
            if (!visited[v] && dist[v] < min) {
                min = dist[v];
                minIndex = v;
            }
        }
        return minIndex;
    }

    static void dijkstra(Graph graph, int source) {
        int dist[] = new int[V];
        boolean visited[] = new boolean[V];
```

Dijkstra's Algorithm

```
for (int i = 0; i < V; i++) {
    dist[i] = Integer.MAX_VALUE;
    visited[i] = false;
}

dist[source] = 0;

for (int count = 0; count < V - 1; count++) {
    int u = minDistance(dist, visited);
    visited[u] = true;

    Node current = graph.adjacencyList[u];
    while (current != null) {
        int v = current.vertex;
        int weight = current.weight;

        if (!visited[v] && dist[u] != Integer.MAX_VALUE && dist[u] +
            weight < dist[v]) {
            dist[v] = dist[u] + weight;
        }

        current = current.next;
    }
}
System.out.println("Vertex   Distance from Source");
```

Dijkstra's Algorithm

```
        System.out.println("Vertex   Distance from Source");
        for (int i = 0; i < V; i++) {
            System.out.println(i + "\t\t" + dist[i]);
        }
    }

    public static void main(String args[]) {
        Graph graph = new Graph(V);

        graph.addEdge(0, 1, 4);
        graph.addEdge(0, 4, 2);
        graph.addEdge(1, 2, 8);
        graph.addEdge(1, 3, 7);
        graph.addEdge(2, 3, 5);
        graph.addEdge(2, 4, 6);
        graph.addEdge(3, 4, 9);

        int source = 0;
        dijkstra(graph, source);
    }
}
```

Bellman-Ford Algorithm

- Bellman-Ford algorithm handles graphs with negative weight edges and can detect negative weight cycles.
- **Algorithm Steps**:
  - Initialize distances to all vertices as infinity (except the source vertex which is 0).
  - Repeat V-1 times (V is the number of vertices): a. For each edge (u, v) with weight w, relax the edge: If distance[u] + w < distance[v], update distance[v] to distance[u] + w.
  - Check for negative weight cycles.

Bellman-Ford Algorithm

```java
public class BellmanFordAlgorithm {
    static final int V = 5; // Number of vertices

    static void bellmanFord(Graph graph, int source) {
        int dist[] = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[source] = 0;

        // Relax all edges V - 1 times
        for (int i = 1; i < V; i++) {
            for (int j = 0; j < V; j++) {
                Node current = graph.adjacencyList[j];
                while (current != null) {
                    int u = j;
                    int v = current.vertex;
                    int weight = current.weight;

                    if (dist[u] != Integer.MAX_VALUE && dist[u] + weight <
                        dist[v]) {
                        dist[v] = dist[u] + weight;
                    }
                    current = current.next;
                }
            }
        }
```

Bellman-Ford Algorithm

```
    // Check for negative-weight cycles
    for (int i = 0; i < V; i++) {
        Node current = graph.adjacencyList[i];
        while (current != null) {
            int u = i;
            int v = current.vertex;
            int weight = current.weight;

            if (dist[u] != Integer.MAX_VALUE && dist[u] + weight < dist[v])
                {
                System.out.println("Graph contains negative-weight cycle");
                return;
            }
            current = current.next;
        }
    }
    System.out.println("Vertex   Distance from Source");
    for (int i = 0; i < V; i++) {
        System.out.println(i + "\t\t" + dist[i]);
    }
}
```

Bellman-Ford Algorithm

```
    public static void main(String args[]) {
        Graph graph = new Graph(V);

        graph.addEdge(0, 1, -1);
        graph.addEdge(0, 2, 4);
        graph.addEdge(1, 2, 3);
        graph.addEdge(1, 3, 2);
        graph.addEdge(1, 4, 2);
        graph.addEdge(3, 2, 5);
        graph.addEdge(3, 1, 1);
        graph.addEdge(4, 3, -3);

        int source = 0;
        bellmanFord(graph, source);
    }
}
```

Summary

- Define and traversal in a binary tree.
- Span a tree with different algorithms.
- Calculate the radius, diameter and the center oth a tree.
- Find the shortest path from one node to any other node In the tree.

Brief of the Next Lecture

- Graph coloring, planer graphs and Euclidian algorithms will be covered in next lecture.

Q & A

Thank You.