Chapter 4.1: Undirected Graphs

Chapter 4.1: Undirected Graphs

Introduction

An undirected graph is a set of vertices connected by edges, where the edges have no direction. They are used to represent a variety of systems, including social networks, communication networks, and transportation systems.

Terminology

- **Vertex (Node):** Fundamental unit represented as a point.
- **Edge (Link):** Connection between two vertices.
- **Path:** Sequence of edges connecting two vertices.
- **Cycle:** Path that starts and ends at the same vertex.
- **Connected Graph:** There is a path between any pair of vertices.
- **Acyclic Graph:** A graph with no cycles.
- **Tree:** An acyclic connected graph.

Graph Representation

Graphs can be represented in several ways:

1. **Adjacency Matrix:** A 2D array where matrix[i][j] indicates the presence of an edge between vertices i and j.
2. **Adjacency List:** An array of lists. The list at index i contains the vertices adjacent to vertex i.

Example: Adjacency List Representation

```java
public class Graph {
```

```
    private final int V; // number of vertices
    private int E; // number of edges
    private Bag<Integer>[] adj; // adjacency lists

    public Graph(int V) {
        this.V = V;
        this.E = 0;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Integer>();
        }
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);
        adj[w].add(v);
        E++;
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }

    public int V() { return V; }
    public int E() { return E; }
}
```

Depth-First Search (DFS)

DFS explores as far as possible along each branch before backtracking. It uses a stack (either implicitly via recursion or explicitly).

DFS Implementation

```java
public class DepthFirstSearch {
    private boolean[] marked; // marked[v] = is there an s-v path?
    private int count; // number of vertices connected to s

    public DepthFirstSearch(Graph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Graph G, int v) {
        marked[v] = true;
        count++;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }

    public boolean marked(int v) {
```

```
        return marked[v];
    }

    public int count() {
        return count;
    }
}
```

Breadth-First Search (BFS)

BFS explores the neighbor nodes at the present depth prior to moving on to nodes at the next depth level. It uses a queue.

BFS Implementation

```java
public class BreadthFirstSearch {
    private boolean[] marked; // marked[v] = is there an s-v path?
    private int[] edgeTo; // edgeTo[v] = last edge on s-v path
    private final int s; // source vertex

    public BreadthFirstSearch(Graph G, int s) {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        this.s = s;
        bfs(G, s);
    }
```

```
private void bfs(Graph G, int s) {
    Queue<Integer> queue = new LinkedList<Integer>();
    marked[s] = true;
    queue.add(s);
    while (!queue.isEmpty()) {
        int v = queue.poll();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                edgeTo[w] = v;
                marked[w] = true;
                queue.add(w);
            }
        }
    }
}

public boolean hasPathTo(int v) {
    return marked[v];
}

public Iterable<Integer> pathTo(int v) {
    if (!hasPathTo(v)) return null;
    Stack<Integer> path = new Stack<Integer>();
    for (int x = v; x != s; x = edgeTo[x]) {
        path.push(x);
    }
    path.push(s);
```

```
        return path;
    }
}
```

Connected Components

A connected component is a maximal set of connected vertices.

Finding Connected Components

```java
public class ConnectedComponents {
    private boolean[] marked; // marked[v] = has vertex v been marked?
    private int[] id; // id[v] = id of connected component containing v
    private int count; // number of connected components

    public ConnectedComponents(Graph G) {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
                count++;
            }
        }
    }

    private void dfs(Graph G, int v) {
```

```
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }

    public int id(int v) {
        return id[v];
    }

    public int count() {
        return count;
    }

    public boolean connected(int v, int w) {
        return id(v) == id(w);
    }
}
```

Cycle Detection

Detecting cycles in an undirected graph can be done using DFS.

Cycle Detection Implementation

```java
public class CycleDetection {
    private boolean[] marked;
    private boolean hasCycle;

    public CycleDetection(Graph G) {
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v, v);
            }
        }
    }

    private void dfs(Graph G, int v, int u) {
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w, v);
            } else if (w != u) {
                hasCycle = true;
            }
        }
    }

    public boolean hasCycle() {
        return hasCycle;
```

```
    }
}
```

Bipartite Graphs

A graph is bipartite if it can be colored with two colors such that no two adjacent vertices share the same color.

Bipartite Check Implementation

```java
public class Bipartite {
    private boolean isBipartite = true;
    private boolean[] color;
    private boolean[] marked;

    public Bipartite(Graph G) {
        color = new boolean[G.V()];
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
            }
        }
    }

    private void dfs(Graph G, int v) {
        marked[v] = true;
```

```
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                color[w] = !color[v];
                dfs(G, w);
            } else if (color[w] == color[v]) {
                isBipartite = false;
            }
        }
    }

    public boolean isBipartite() {
        return isBipartite;
    }
}
```

Applications

1. **Network Connectivity:** Checking if a network is fully connected.
2. **Social Networks:** Finding communities or clusters.
3. **Geographical Mapping:** Modeling and analyzing regions and paths.

Conclusion

Undirected graphs are a fundamental data structure that can represent many types of systems. By using algorithms such as DFS, BFS, connected components, cycle detection, and bipartite checking, we can efficiently analyze and process graph data.