Chapter 4.2: Directed Graphs

Chapter 4.2: Directed Graphs

Introduction

A directed graph, or digraph, is a set of vertices connected by edges, where the edges have a direction. They are used to represent various systems such as web pages (with hyperlinks), road maps, and project schedules.

Terminology

- **Vertex (Node):** Fundamental unit represented as a point.
- **Directed Edge (Arc):** Ordered pair of vertices (v -> w) indicating a one-way relationship from vertex v to vertex w.
- **Path:** Sequence of directed edges connecting two vertices.
- **Cycle:** Path that starts and ends at the same vertex.
- **Strongly Connected:** There is a path in each direction between each pair of vertices.
- **DAG (Directed Acyclic Graph):** A directed graph with no cycles.

Graph Representation

Directed graphs can be represented similarly to undirected graphs, using adjacency matrices or adjacency lists.

Example: Adjacency List Representation

```java
public class Digraph {
    private final int V; // number of vertices
    private int E; // number of edges
```

```
    private Bag<Integer>[] adj; // adjacency lists

    public Digraph(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);
        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 in directed graphs is used similarly to undirected graphs, exploring as far as possible along each branch before backtracking.

DFS Implementation

```java
public class DirectedDFS {
    private boolean[] marked;

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

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

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

Breadth-First Search (BFS)

BFS in directed graphs is used similarly to undirected graphs, exploring the neighbor nodes at the present depth prior to moving on to nodes at the next depth level.

BFS Implementation

```java
public class DirectedBFS {
    private boolean[] marked;
    private int[] edgeTo;
    private final int s;

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

    private void bfs(Digraph 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;
    }
}
```

Topological Sort

A topological sort of a directed graph is a linear ordering of its vertices such that for every directed edge v -> w, vertex v comes before w in the ordering. This is possible if and only if the graph is a DAG.

Topological Sort Implementation

```java
public class Topological {
    private Stack<Integer> order;

    public Topological(Digraph G) {
        DirectedCycle cycleFinder = new DirectedCycle(G);
        if (!cycleFinder.hasCycle()) {
            DepthFirstOrder dfs = new DepthFirstOrder(G);
            order = dfs.reversePost();
        }
    }

    public Iterable<Integer> order() {
        return order;
    }

    public boolean isDAG() {
        return order != null;
    }
}
```

Strongly Connected Components (SCC)

SCCs are maximal subgraphs where every vertex is reachable from every other vertex in the subgraph. Kosaraju's algorithm is commonly used to find SCCs.

Kosaraju's Algorithm Implementation

```java
public class KosarajuSCC {
    private boolean[] marked;
    private int[] id;
    private int count;

    public KosarajuSCC(Digraph G) {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        DepthFirstOrder order = new DepthFirstOrder(G.reverse());
        for (int s : order.reversePost()) {
            if (!marked[s]) {
                dfs(G, s);
                count++;
            }
        }
    }

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

        }
    }

    public boolean stronglyConnected(int v, int w) {
        return id[v] == id[w];
    }

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

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

## Cycle Detection

Cycle detection in a directed graph can be done using DFS.

### Cycle Detection Implementation

```java
public class DirectedCycle {
    private boolean[] marked;
    private int[] edgeTo;
    private Stack<Integer> cycle;
    private boolean[] onStack;

```
public DirectedCycle(Digraph G) {
    onStack = new boolean[G.V()];
    edgeTo = new int[G.V()];
    marked = new boolean[G.V()];
    for (int v = 0; v < G.V(); v++) {
        if (!marked[v]) dfs(G, v);
    }
}

private void dfs(Digraph G, int v) {
    onStack[v] = true;
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (this.hasCycle()) return;
        else if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        } else if (onStack[w]) {
            cycle = new Stack<Integer>();
            for (int x = v; x != w; x = edgeTo[x]) {
                cycle.push(x);
            }
            cycle.push(w);
            cycle.push(v);
        }
    }
```

```
        onStack[v] = false;
    }

    public boolean hasCycle() {
        return cycle != null;
    }

    public Iterable<Integer> cycle() {
        return cycle;
    }
}
```

Applications

1. **Web Crawling:** Navigating the web by following hyperlinks.

2. **Task Scheduling:** Ordering tasks with dependencies.

3. **Network Routing:** Determining paths for data transfer.

Conclusion

Directed graphs are crucial for modeling systems where direction matters. With algorithms like DFS, BFS, topological sort, and SCC detection, we can analyze and process directed graph data efficiently.