Chapter 4.4: Shortest Paths

Chapter 4.4: Shortest Paths

Introduction

Shortest path algorithms are used to find the minimum distance or minimum cost path from a source vertex to a destination vertex in a graph. These algorithms are crucial in various applications such as routing, navigation, and network optimization.

Terminology

- **Path:** A sequence of edges connecting two vertices.

- **Shortest Path:** A path with the minimum sum of edge weights.

- **Weighted Graph:** A graph where each edge has an associated numerical value (weight).

Single-Source Shortest Path

The single-source shortest path problem involves finding the shortest paths from a source vertex to all other vertices in the graph.

Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights.

Dijkstra's Algorithm Steps

1. Initialize distances from the source to all vertices as infinity, except the source vertex itself, which is set to 0.

2. Use a priority queue to select the vertex with the minimum distance.

3. Update the distances to all adjacent vertices of the selected vertex.

4. Repeat steps 2 and 3 until the priority queue is empty.

Dijkstra's Algorithm Implementation

```java
public class DijkstraSP {
    private double[] distTo;
    private Edge[] edgeTo;
    private IndexMinPQ<Double> pq;

    public DijkstraSP(EdgeWeightedGraph G, int s) {
        distTo = new double[G.V()];
        edgeTo = new Edge[G.V()];
        for (int v = 0; v < G.V(); v++) {
            distTo[v] = Double.POSITIVE_INFINITY;
        }
        distTo[s] = 0.0;
        pq = new IndexMinPQ<Double>(G.V());
        pq.insert(s, 0.0);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            for (Edge e : G.adj(v)) {
                relax(e, v);
            }
        }
    }

    private void relax(Edge e, int v) {
```

```
        int w = e.other(v);
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
            else pq.insert(w, distTo[w]);
        }
    }

    public double distTo(int v) {
        return distTo[v];
    }

    public boolean hasPathTo(int v) {
        return distTo[v] < Double.POSITIVE_INFINITY;
    }

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

Bellman-Ford Algorithm

The Bellman-Ford algorithm handles graphs with negative edge weights and can detect negative weight cycles.

Bellman-Ford Algorithm Steps

1. Initialize distances from the source to all vertices as infinity, except the source vertex itself, which is set to 0.
2. Relax all edges |V|-1 times.
3. Check for negative weight cycles.

Bellman-Ford Algorithm Implementation

```java
public class BellmanFordSP {
    private double[] distTo;
    private Edge[] edgeTo;
    private boolean[] onQueue;
    private Queue<Integer> queue;
    private int cost;
    private Iterable<Edge> cycle;

    public BellmanFordSP(EdgeWeightedDigraph G, int s) {
        distTo = new double[G.V()];
        edgeTo = new Edge[G.V()];
        onQueue = new boolean[G.V()];
        queue = new LinkedList<Integer>();
        for (int v = 0; v < G.V(); v++) {
```

```
        distTo[v] = Double.POSITIVE_INFINITY;
    }
    distTo[s] = 0.0;
    queue.add(s);
    onQueue[s] = true;
    while (!queue.isEmpty() && !hasNegativeCycle()) {
        int v = queue.poll();
        onQueue[v] = false;
        relax(G, v);
    }
}

private void relax(EdgeWeightedDigraph G, int v) {
    for (Edge e : G.adj(v)) {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (!onQueue[w]) {
                queue.add(w);
                onQueue[w] = true;
            }
        }
        if (cost++ % G.V() == 0) {
            findNegativeCycle();
            if (hasNegativeCycle()) return;
        }
```

```
        }
    }

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

    private void findNegativeCycle() {
        int V = edgeTo.length;
        EdgeWeightedDigraph spt = new EdgeWeightedDigraph(V);
        for (int v = 0; v < V; v++) {
            if (edgeTo[v] != null) {
                spt.addEdge(edgeTo[v]);
            }
        }
        EdgeWeightedCycleFinder cf = new EdgeWeightedCycleFinder(spt);
        cycle = cf.cycle();
    }

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

All-Pairs Shortest Path

The all-pairs shortest path problem involves finding the shortest paths between every pair of vertices in the graph.

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is a dynamic programming algorithm used to solve the all-pairs shortest path problem.

Floyd-Warshall Algorithm Steps

1. Initialize the distance matrix with edge weights and set the distance from a vertex to itself as 0.
2. Update the distance matrix by considering all pairs of vertices and checking if a shorter path exists through an intermediate vertex.

Floyd-Warshall Algorithm Implementation

```java
public class FloydWarshall {
    private boolean hasNegativeCycle;
    private double[][] distTo;
    private DirectedEdge[][] edgeTo;

    public FloydWarshall(EdgeWeightedDigraph G) {
        int V = G.V();
        distTo = new double[V][V];
        edgeTo = new DirectedEdge[V][V];

        for (int v = 0; v < V; v++) {
            for (int w = 0; w < V; w++) {
                distTo[v][w] = Double.POSITIVE_INFINITY;
            }
```

```
}


for (int v = 0; v < V; v++) {
    for (DirectedEdge e : G.adj(v)) {
        distTo[e.from()][e.to()] = e.weight();
        edgeTo[e.from()][e.to()] = e;
    }
    if (distTo[v][v] >= 0.0) {
        distTo[v][v] = 0.0;
        edgeTo[v][v] = null;
    }
}


for (int i = 0; i < V; i++) {
    for (int v = 0; v < V; v++) {
        if (edgeTo[v][i] == null) continue;
        for (int w = 0; w < V; w++) {
            if (distTo[v][w] > distTo[v][i] + distTo[i][w]) {
                distTo[v][w] = distTo[v][i] + distTo[i][w];
                edgeTo[v][w] = edgeTo[i][w];
            }
        }
        if (distTo[v][v] < 0.0) {
            hasNegativeCycle = true;
            return;
        }
    }
```

```
        }
    }

    public boolean hasNegativeCycle() {
        return hasNegativeCycle;
    }

    public double dist(int v, int w) {
        return distTo[v][w];
    }

    public boolean hasPath(int v, int w) {
        return distTo[v][w] < Double.POSITIVE_INFINITY;
    }

    public Iterable<DirectedEdge> path(int v, int w) {
        if (!hasPath(v, w)) return null;
        Stack<DirectedEdge> path = new Stack<DirectedEdge>();
        for (DirectedEdge e = edgeTo[v][w]; e != null; e = edgeTo[e.from()][w]) {
            path.push(e);
        }
        return path;
    }
}
```

Applications

1. **Routing and Navigation:** Finding the shortest route between two locations.

2. **Network Optimization:** Optimizing the cost of data transfer in networks.

3. **Project Scheduling:** Determining the shortest time to complete tasks with dependencies.

Conclusion

Shortest path algorithms are essential in various fields, providing efficient solutions to complex problems. Dijkstra's, Bellman-Ford, and Floyd-Warshall algorithms offer robust methods for finding shortest paths in different types of graphs.