Digvijay Thakare

[digvijaythakare2017@gmail.com](mailto:digvijaythakare2017@gmail.com)

Day9-10 Core Java

**Task 1: Dijkstra’s Shortest Path Finder**

**Code Dijkstra’s algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.**

**Solution:**

package cam.day9;

import java.util.\*;

public class Dijkstra {

public static Map<Character, Integer> dijkstra(Map<Character, Map<Character, Integer>> graph, char start) {

Map<Character, Integer> distances = new HashMap<>();

PriorityQueue<Node> priorityQueue = new PriorityQueue<>(Comparator.*comparingInt*(node -> node.distance));

Set<Character> visited = new HashSet<>();

for (char node : graph.keySet()) {

distances.put(node, Integer.***MAX\_VALUE***);

}

distances.put(start, 0);

priorityQueue.offer(new Node(start, 0));

while (!priorityQueue.isEmpty()) {

Node current = priorityQueue.poll();

if (visited.contains(current.name)) {

continue;

}

visited.add(current.name);

for (Map.Entry<Character, Integer> neighbor : graph.get(current.name).entrySet()) {

int distance = current.distance + neighbor.getValue();

if (distance < distances.get(neighbor.getKey())) {

distances.put(neighbor.getKey(), distance);

priorityQueue.offer(new Node(neighbor.getKey(), distance));

}

}

}

return distances;

}

static class Node {

char name;

int distance;

Node(char name, int distance) {

this.name = name;

this.distance = distance;

}

}

public static void main(String[] args) {

Map<Character, Map<Character, Integer>> graph = new HashMap<>();

graph.put('A', Map.*of*('B', 2, 'C', 5));

graph.put('B', Map.*of*('A', 2, 'C', 1, 'D', 7));

graph.put('C', Map.*of*('A', 5, 'B', 1, 'D', 3));

graph.put('D', Map.*of*('B', 7, 'C', 3));

char startNode = 'A';

Map<Character, Integer> distances = *dijkstra*(graph, startNode);

System.***out***.println("Shortest distances from node " + startNode + ":");

for (char node : distances.keySet()) {

System.***out***.println("To node " + node + ": " + distances.get(node));

}

}

}

**Output:**

Shortest distances from node A:

To node A: 0

To node B: 2

To node C: 3

To node D: 6

**Task 2: Kruskal’s Algorithm for MST**

**Implement Kruskal’s algorithm to find the minimum spanning tree of a given connected, undirected graph with non-negative edge weights.**

**Solution:**

package cam.day9;

import java.util.\*;

public class Kruskal {

static class Edge {

char source;

char destination;

int weight;

Edge(char source, char destination, int weight) {

this.source = source;

this.destination = destination;

this.weight = weight;

}

}

static class Subset {

char parent;

int rank;

Subset(char parent, int rank) {

this.parent = parent;

this.rank = rank;

}

}

public static List<Edge> kruskalMST(Map<Character, Map<Character, Integer>> graph) {

List<Edge> result = new ArrayList<>();

List<Edge> edges = new ArrayList<>();

// Convert graph to list of edges

for (char source : graph.keySet()) {

for (Map.Entry<Character, Integer> entry : graph.get(source).entrySet()) {

char destination = entry.getKey();

int weight = entry.getValue();

edges.add(new Edge(source, destination, weight));

}

}

// Sort edges by weight

Collections.*sort*(edges, Comparator.*comparingInt*(e -> e.weight));

Map<Character, Subset> subsets = new HashMap<>();

for (char vertex : graph.keySet()) {

subsets.put(vertex, new Subset(vertex, 0));

}

int i = 0;

int numEdges = 0;

while (numEdges < graph.size() - 1 && i < edges.size()) {

Edge nextEdge = edges.get(i++);

char x = *find*(subsets, nextEdge.source);

char y = *find*(subsets, nextEdge.destination);

if (x != y) {

result.add(nextEdge);

*union*(subsets, x, y);

numEdges++;

}

}

return result;

}

public static char find(Map<Character, Subset> subsets, char vertex) {

if (subsets.get(vertex).parent != vertex) {

subsets.get(vertex).parent = *find*(subsets, subsets.get(vertex).parent);

}

return subsets.get(vertex).parent;

}

public static void union(Map<Character, Subset> subsets, char x, char y) {

char xRoot = *find*(subsets, x);

char yRoot = *find*(subsets, y);

if (subsets.get(xRoot).rank < subsets.get(yRoot).rank) {

subsets.get(xRoot).parent = yRoot;

} else if (subsets.get(xRoot).rank > subsets.get(yRoot).rank) {

subsets.get(yRoot).parent = xRoot;

} else {

subsets.get(yRoot).parent = xRoot;

subsets.get(xRoot).rank++;

}

}

public static void main(String[] args) {

Map<Character, Map<Character, Integer>> graph = new HashMap<>();

graph.put('A', Map.*of*('B', 4, 'C', 1));

graph.put('B', Map.*of*('A', 4, 'C', 2, 'D', 1));

graph.put('C', Map.*of*('A', 1, 'B', 2, 'D', 5));

graph.put('D', Map.*of*('B', 1, 'C', 5));

List<Edge> mst = *kruskalMST*(graph);

System.***out***.println("Edges in the Minimum Spanning Tree:");

for (Edge edge : mst) {

System.***out***.println(edge.source + " - " + edge.destination + " : " + edge.weight);

}

}

}

**Output:**

Edges in the Minimum Spanning Tree:

A - C : 1

B - D : 1

B - C : 2

**Task 3: Union-Find for Cycle Detection**

**Write a Union-Find data structure with path compression. Use this data structure to detect a cycle in an undirected graph.**

**Solution:**

package cam.day9;

import java.util.\*;

public class UnionFind {

private int[] parent;

private int[] rank;

public UnionFind(int size) {

parent = new int[size];

rank = new int[size];

for (int i = 0; i < size; i++) {

parent[i] = i;

rank[i] = 0;

}

}

public int find(int x) {

if (parent[x] != x) {

parent[x] = find(parent[x]);

}

return parent[x];

}

public void union(int x, int y) {

int xRoot = find(x);

int yRoot = find(y);

if (xRoot == yRoot) {

return;

}

if (rank[xRoot] < rank[yRoot]) {

parent[xRoot] = yRoot;

} else if (rank[xRoot] > rank[yRoot]) {

parent[yRoot] = xRoot;

} else {

parent[yRoot] = xRoot;

rank[xRoot]++;

}

}

public static boolean hasCycle(Map<Character, List<Character>> graph) {

UnionFind uf = new UnionFind(graph.size());

for (char node : graph.keySet()) {

int parentX = uf.find(node - 'A');

for (char neighbor : graph.get(node)) {

int parentY = uf.find(neighbor - 'A');

if (parentX == parentY) {

return true; // Cycle detected

}

uf.union(parentX, parentY);

}

}

return false; // No cycle detected

}

public static void main(String[] args) {

// Example graph represented as an adjacency list

Map<Character, List<Character>> graph = new HashMap<>();

graph.put('A', Arrays.*asList*('B', 'C'));

graph.put('B', Arrays.*asList*('A', 'C', 'D'));

graph.put('C', Arrays.*asList*('A', 'B', 'D'));

graph.put('D', Arrays.*asList*('B', 'C'));

System.***out***.println("Graph:");

for (char node : graph.keySet()) {

System.***out***.print(node + " --- ");

for (char neighbor : graph.get(node)) {

System.***out***.print(neighbor + " ");

}

System.***out***.println();

}

if (*hasCycle*(graph)) {

System.***out***.println("The graph contains a cycle.");

} else {

System.***out***.println("The graph does not contain a cycle.");

}

}

}

**Output:**

Graph:

A --- B C

B --- A C D

C --- A B D

D --- B C

The graph contains a cycle.