Day 19 Assignment

Name: Mehul Anjikhane Email: mehulanjikhane13@gmail.com

**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.**

**package** algorithms;

**import** java.util.\*;

**public** **class** ShortestPathFinder {

**static** **class** Node **implements** Comparable<Node> {

**int** vertex;

**int** weight;

Node(**int** vertex, **int** weight) {

**this**.vertex = vertex;

**this**.weight = weight;

}

@Override

**public** **int** compareTo(Node other) {

**return** Integer.*compare*(**this**.weight, other.weight);

}

}

**public** **static** **void** dijkstra(**int**[][] graph, **int** startVertex) {

**int** V = graph.length;

**int**[] distances = **new** **int**[V];

**boolean**[] shortestPathTreeSet = **new** **boolean**[V];

Arrays.*fill*(distances, Integer.***MAX\_VALUE***);

distances[startVertex] = 0;

PriorityQueue<Node> priorityQueue = **new** PriorityQueue<>();

priorityQueue.add(**new** Node(startVertex, 0));

**while** (!priorityQueue.isEmpty()) {

**int** u = priorityQueue.poll().vertex;

**if** (!shortestPathTreeSet[u]) {

shortestPathTreeSet[u] = **true**;

**for** (**int** v = 0; v < V; v++) {

**if** (!shortestPathTreeSet[v] && graph[u][v] != 0 && distances[u] != Integer.***MAX\_VALUE*** && distances[u] + graph[u][v] <

distances[v]) { distances[v] = distances[u] + graph[u][v];

priorityQueue.add(**new** Node(v, distances[v]));

}

}

}

}

*printSolution*(distances, V);

}

**private** **static** **void** printSolution(**int**[] distances, **int** V) {

System.***out***.println("Vertex \t Distance from Source");

**for** (**int** i = 0; i < V; i++)

System.***out***.println(i + " \t\t " + distances[i]);

}

**public** **static** **void** main(String[] args) {

**int** graph[][] = **new** **int**[][] { { 0, 10, 0, 30, 100 }, { 10, 0, 50, 0, 0 }, { 0, 50, 0, 20, 10 },{ 30, 0, 20, 0, 60 }, { 100, 0, 10, 60, 0 } };

ShortestPathFinder.*dijkstra*(graph, 0);

}

}

**Output:**

Vertex Distance from Source

0 0

1 10

2 50

3 30

4 60

**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.**

**package** algorithms;

**import** java.util.\*;

**public** **class** Kruskal {

**class** Edge **implements** Comparable<Edge> {

**int** src, dest, weight;

**public** **int** compareTo(Edge compareEdge) {

**return** **this**.weight - compareEdge.weight;

}

}

**class** Subset {

**int** parent, rank;

}

**int** V, E;

Edge edge[];

Kruskal(**int** v, **int** e) {

V = v;

E = e;

edge = **new** Edge[E];

**for** (**int** i = 0; i < e; ++i)

edge[i] = **new** Edge();

}

**int** find(Subset subsets[], **int** i) {

**if** (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

**return** subsets[i].parent;

}

**void** union(Subset subsets[], **int** x, **int** y) {

**int** xroot = find(subsets, x);

**int** yroot = find(subsets, y);

**if** (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

**else** **if** (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

**else** {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

**void** KruskalMST() {

Edge result[] = **new** Edge[V];

**int** e = 0;

**int** i = 0;

**for** (i = 0; i < V; ++i)

result[i] = **new** Edge();

Arrays.*sort*(edge);

Subset subsets[] = **new** Subset[V];

**for** (i = 0; i < V; ++i)

subsets[i] = **new** Subset();

**for** (**int** v = 0; v < V; ++v) {

subsets[v].parent = v;

subsets[v].rank = 0;

}

i = 0;

**while** (e < V - 1) {

Edge next\_edge = edge[i++];

**int** x = find(subsets, next\_edge.src);

**int** y = find(subsets, next\_edge.dest);

**if** (x != y) {

result[e++] = next\_edge;

union(subsets, x, y);

}

}

System.***out***.println("Following are the edges in the constructed "

+ "MST");

**for** (i = 0; i < e; ++i)

System.***out***.println(result[i].src + " -- " + result[i].dest

+ " == " + result[i].weight);

}

**public** **static** **void** main(String[] args) {

**int** V = 4;

**int** E = 5;

Kruskal graph = **new** Kruskal(V, E);

graph.edge[0].src = 0;

graph.edge[0].dest = 1;

graph.edge[0].weight = 10;

graph.edge[1].src = 0;

graph.edge[1].dest = 2;

graph.edge[1].weight = 6;

graph.edge[2].src = 0;

graph.edge[2].dest = 3;

graph.edge[2].weight = 5;

graph.edge[3].src = 1;

graph.edge[3].dest = 3;

graph.edge[3].weight = 15;

graph.edge[4].src = 2;

graph.edge[4].dest = 3;

graph.edge[4].weight = 4;

graph.KruskalMST();

}

}

**Output:**

Following are the edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

**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.**

**package** algorithms;

**import** java.util.\*;

**public** **class** UnionFind {

**class** Edge {

**int** src, dest;

}

**int** V, E;

Edge edge[];

UnionFind(**int** v, **int** e) {

V = v;

E = e;

edge = **new** Edge[E];

**for** (**int** i = 0; i < e; ++i)

edge[i] = **new** Edge();

}

**int** find(**int** parent[], **int** i) {

**if** (parent[i] == -1)

**return** i;

**return** parent[i] = find(parent, parent[i]);

}

**void** union(**int** parent[], **int** x, **int** y) {

**int** xset = find(parent, x);

**int** yset = find(parent, y);

parent[xset] = yset;

}

**int** isCycle(UnionFind graph) {

**int** parent[] = **new** **int**[graph.V];

Arrays.*fill*(parent, -1);

**for** (**int** i = 0; i < graph.E; ++i) {

**int** x = find(parent, graph.edge[i].src);

**int** y = find(parent, graph.edge[i].dest);

**if** (x == y)

**return** 1;

union(parent, x, y);

}

**return** 0;

}

**public** **static** **void** main(String[] args) {

**int** V = 3, E = 3;

UnionFind graph = **new** UnionFind(V, E);

graph.edge[0].src = 0;

graph.edge[0].dest = 1;

graph.edge[1].src = 1;

graph.edge[1].dest = 2;

graph.edge[2].src = 0;

graph.edge[2].dest = 2;

**if** (graph.isCycle(graph) == 1)

System.***out***.println("Graph contains cycle");

**else**

System.***out***.println("Graph doesn't contain cycle");

}

}

**Output:**

Graph contains cycle.