Q3

Boruvka’s Algorithm

package Q3;

import java.util.\*;

public class Boruvka {

private static class Edge {

int u, v, weight;

Edge(int u, int v, int weight) {

this.u = u;

this.v = v;

this.weight = weight;

}

}

public static List<Edge> boruvka(List<Edge> edges, int n) {

UnionFind uf = new UnionFind(n);

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

int numTrees = n;

int[] cheapest = new int[n];

while (numTrees > 1) {

Arrays.fill(cheapest, -1);

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

Edge edge = edges.get(i);

int set1 = uf.find(edge.u);

int set2 = uf.find(edge.v);

if (set1 != set2) {

if (cheapest[set1] == -1 || edge.weight < edges.get(cheapest[set1]).weight) {

cheapest[set1] = i;

}

if (cheapest[set2] == -1 || edge.weight < edges.get(cheapest[set2]).weight) {

cheapest[set2] = i;

}

}

}

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

if (cheapest[i] != -1) {

Edge edge = edges.get(cheapest[i]);

int set1 = uf.find(edge.u);

int set2 = uf.find(edge.v);

if (set1 != set2) {

mst.add(edge);

uf.union(set1, set2);

numTrees--;

}

}

}

}

return mst;

}

private static class UnionFind {

private int[] parent, rank;

UnionFind(int size) {

parent = new int[size];

rank = new int[size];

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

parent[i] = i;

}

}

int find(int u) {

if (parent[u] != u) {

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

}

return parent[u];

}

void union(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

} else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

}

public static void main(String[] args) {

// Example graph

List<Edge> edges = Arrays.asList(

new Edge(0, 1, 4),

new Edge(0, 2, 1),

new Edge(1, 2, 2),

new Edge(1, 3, 5),

new Edge(2, 3, 8)

);

List<Edge> mst = boruvka(edges, 4);

System.out.println("MST Edges: ");

for (Edge edge : mst) {

System.out.println(edge.u + " - " + edge.v + " : " + edge.weight);

}

}

}al’s and Prim’s algorithms.

1. Time Complexity of Boruvka’s algorithm is O(E log V) which is the same as Kruskal’s and Prim’s algorithms.

**Space complexity:** The space complexity of Boruvka’s algorithm is O(V).

Dijkstra’s Algorithm

package Q3;

import java.util.\*;

public class Dijkstra {

private static final int INF = Integer.MAX\_VALUE;

public static int[] dijkstra(List<List<int[]>> graph, int source) {

int n = graph.size();

int[] dist = new int[n];

Arrays.fill(dist, INF);

dist[source] = 0;

PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));

pq.add(new int[]{source, 0});

while (!pq.isEmpty()) {

int[] curr = pq.poll();

int u = curr[0], d = curr[1];

if (d > dist[u]) continue;

for (int[] edge : graph.get(u)) {

int v = edge[0], weight = edge[1];

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

pq.add(new int[]{v, dist[v]});

}

}

}

return dist;

}

public static void main(String[] args) {

// Example graph with non-negative weights

List<List<int[]>> graph = Arrays.asList(

Arrays.asList(new int[]{1, 4}, new int[]{2, 1}),

Arrays.asList(new int[]{2, 2}),

Arrays.asList(new int[]{3, 5}),

Arrays.asList()

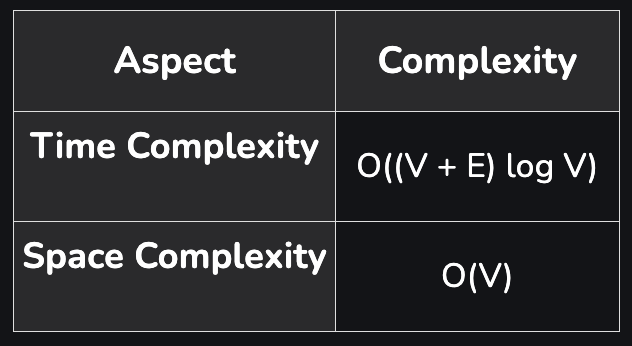
);

int[] dist = dijkstra(graph, 0);

System.out.println("Distances from source 0: " + Arrays.toString(dist));

}

}



Tarjan’s Strongly Connected Components

package Q3;

import java.util.\*;

public class TarjanSCC {

private static int index = 0;

private static Stack<Integer> stack = new Stack<>();

private List<List<Integer>> graph; // Remove static keyword

private static int[] indices, lowLink;

private static boolean[] onStack;

private static List<List<Integer>> sccList = new ArrayList<>();

public TarjanSCC(List<List<Integer>> graph) {

this.graph = graph;

indices = new int[graph.size()];

lowLink = new int[graph.size()];

onStack = new boolean[graph.size()];

Arrays.fill(indices, -1);

}

public List<List<Integer>> findSCC() {

for (int v = 0; v < graph.size(); v++) {

if (indices[v] == -1) {

strongConnect(v);

}

}

return sccList;

}

private void strongConnect(int v) {

indices[v] = lowLink[v] = index++;

stack.push(v);

onStack[v] = true;

for (int w : graph.get(v)) {

if (indices[w] == -1) {

strongConnect(w);

lowLink[v] = Math.min(lowLink[v], lowLink[w]);

} else if (onStack[w]) {

lowLink[v] = Math.min(lowLink[v], indices[w]);

}

}

if (lowLink[v] == indices[v]) {

List<Integer> scc = new ArrayList<>();

int w;

do {

w = stack.pop();

onStack[w] = false;

scc.add(w);

} while (w != v);

sccList.add(scc);

}

}

public static void main(String[] args) {

// Test with an example graph

List<List<Integer>> graph = Arrays.asList(

Arrays.asList(1),

Arrays.asList(2),

Arrays.asList(0, 3),

Arrays.asList(4),

Arrays.asList(5),

Arrays.asList(4)

);

TarjanSCC tarjan = new TarjanSCC(graph);

List<List<Integer>> sccs = tarjan.findSCC();

System.out.println("SCCs: " + sccs);

}

}

**Time Complexity:** The above algorithm mainly calls DFS, DFS takes O(V+E) for a graph represented using an adjacency list.

**Auxiliary Space:** O(V)

Performance Testbench

package Q3;

import java.util.\*;

public class PerformanceTest {

// Tarjan's SCC Algorithm

static class TarjanSCC {

private static int index = 0;

private static Stack<Integer> stack = new Stack<>();

private static List<List<Integer>> graph;

private static int[] indices, lowLink;

private static boolean[] onStack;

private static List<List<Integer>> sccList = new ArrayList<>();

public TarjanSCC(List<List<Integer>> graph) {

TarjanSCC.graph = graph;

indices = new int[graph.size()];

lowLink = new int[graph.size()];

onStack = new boolean[graph.size()];

Arrays.fill(indices, -1);

}

public List<List<Integer>> findSCC() {

for (int v = 0; v < graph.size(); v++) {

if (indices[v] == -1) {

strongConnect(v);

}

}

return sccList;

}

private void strongConnect(int v) {

indices[v] = lowLink[v] = index++;

stack.push(v);

onStack[v] = true;

for (int w : graph.get(v)) {

if (indices[w] == -1) {

strongConnect(w);

lowLink[v] = Math.min(lowLink[v], lowLink[w]);

} else if (onStack[w]) {

lowLink[v] = Math.min(lowLink[v], indices[w]);

}

}

if (lowLink[v] == indices[v]) {

List<Integer> scc = new ArrayList<>();

int w;

do {

w = stack.pop();

onStack[w] = false;

scc.add(w);

} while (w != v);

sccList.add(scc);

}

}

}

// Dijkstra's Algorithm

static class Dijkstra {

private static final int INF = Integer.MAX\_VALUE;

public static int[] dijkstra(List<List<int[]>> graph, int source) {

int n = graph.size();

int[] dist = new int[n];

Arrays.fill(dist, INF);

dist[source] = 0;

PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));

pq.add(new int[]{source, 0});

while (!pq.isEmpty()) {

int[] curr = pq.poll();

int u = curr[0], d = curr[1];

if (d > dist[u]) continue;

for (int[] edge : graph.get(u)) {

int v = edge[0], weight = edge[1];

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

pq.add(new int[]{v, dist[v]});

}

}

}

return dist;

}

}

// Boruvka's MST Algorithm

static class Boruvka {

private static class Edge {

int u, v, weight;

Edge(int u, int v, int weight) {

this.u = u;

this.v = v;

this.weight = weight;

}

}

public static List<Edge> boruvka(List<Edge> edges, int n) {

UnionFind uf = new UnionFind(n);

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

int numTrees = n;

int[] cheapest = new int[n];

while (numTrees > 1) {

Arrays.fill(cheapest, -1);

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

Edge edge = edges.get(i);

int set1 = uf.find(edge.u);

int set2 = uf.find(edge.v);

if (set1 != set2) {

if (cheapest[set1] == -1 || edge.weight < edges.get(cheapest[set1]).weight) {

cheapest[set1] = i;

}

if (cheapest[set2] == -1 || edge.weight < edges.get(cheapest[set2]).weight) {

cheapest[set2] = i;

}

}

}

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

if (cheapest[i] != -1) {

Edge edge = edges.get(cheapest[i]);

int set1 = uf.find(edge.u);

int set2 = uf.find(edge.v);

if (set1 != set2) {

mst.add(edge);

uf.union(set1, set2);

numTrees--;

}

}

}

}

return mst;

}

private static class UnionFind {

private int[] parent, rank;

UnionFind(int size) {

parent = new int[size];

rank = new int[size];

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

parent[i] = i;

}

}

int find(int u) {

if (parent[u] != u) {

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

}

return parent[u];

}

void union(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

} else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

}

}

// Main method for performance testing

public static void main(String[] args) {

// Tarjan's SCC Example

List<List<Integer>> graphSCC = Arrays.asList(

Arrays.asList(1),

Arrays.asList(2),

Arrays.asList(0, 3),

Arrays.asList(4),

Arrays.asList(5),

Arrays.asList(4)

);

long startTime = System.nanoTime();

TarjanSCC tarjan = new TarjanSCC(graphSCC);

List<List<Integer>> sccs = tarjan.findSCC();

long endTime = System.nanoTime();

long duration = endTime - startTime;

long memoryUsage = (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) / 1024;

System.out.println("Tarjan's SCC Execution Time: " + (duration / 1\_000\_000) + " ms");

System.out.println("Tarjan's SCC Memory Usage: " + memoryUsage + " KB");

System.out.println("SCCs: " + sccs);

System.out.println("=========================================");

// Dijkstra's Algorithm Example

List<List<int[]>> graphDijkstra = Arrays.asList(

Arrays.asList(new int[]{1, 4}, new int[]{2, 1}),

Arrays.asList(new int[]{2, 2}),

Arrays.asList(new int[]{3, 5}),

Arrays.asList()

);

startTime = System.nanoTime();

int[] dist = Dijkstra.dijkstra(graphDijkstra, 0);

endTime = System.nanoTime();

duration = endTime - startTime;

memoryUsage = (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) / 1024;

System.out.println("Dijkstra's Algorithm Execution Time: " + (duration / 1\_000\_000) + " ms");

System.out.println("Dijkstra's Algorithm Memory Usage: " + memoryUsage + " KB");

System.out.println("Distances from source 0: " + Arrays.toString(dist));

System.out.println("=========================================");

// Boruvka's MST Example

List<Boruvka.Edge> edges = Arrays.asList(

new Boruvka.Edge(0, 1, 4),

new Boruvka.Edge(0, 2, 1),

new Boruvka.Edge(1, 2, 2),

new Boruvka.Edge(1, 3, 5),

new Boruvka.Edge(2, 3, 8)

);

startTime = System.nanoTime();

List<Boruvka.Edge> mst = Boruvka.boruvka(edges, 4);

endTime = System.nanoTime();

duration = endTime - startTime;

memoryUsage = (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) / 1024;

System.out.println("Boruvka's Algorithm Execution Time: " + (duration / 1\_000\_000) + " ms");

System.out.println("Boruvka's Algorithm Memory Usage: " + memoryUsage + " KB");

System.out.println("MST Edges:");

for (Boruvka.Edge edge : mst) {

System.out.println(" " + edge.u + " - " + edge.v + " : " + edge.weight);

}

}

}

Output:

