import java.util.\*;

import java.util.stream.Collectors;

/\*\*  
\* Utility class to determine allocations of units to transporters  
\* based on characteristics of suppliers.  
\* Also includes utility functions associated with the domain.  
\*-  
\* Different allocator methods may address different goal functions.  
\* For example, one goal may be achieving the least expensive allocation;  
\* whereas another goal may be robust to potential underestimates of demand.  
\*-  
\* Allocations must account for all constraints, such as  
\* maximum transport capacity and maximum storage capacity.  
\*-  
\* @author CS4050  
\* @author Dr. Jody Paul  
\* @version 20231114  
\* Student William Hellems-Moody  
\* Note - UPDATED WITH REVISED CONTENT FROM 4050 LECTURE MODULE (11-27-23)  
\* - Comments added to required methods CheapestPath and allocateForDemand  
\* - Public interface methods are unchanged  
\* - Private helper methods are implemented to support the public interface methods \*/

public class SimpleAllocator {

// ========= PRIMARY METHODS (BELOW) =========

public static Collection<Transporter> cheapestPath(Collection<Supplier>

suppliers,

transporters,

Collection<Transporter>

Supplier source,

Supplier destination) {

// Keep track of the lowest cost to reach each supplier.

Map<Supplier, Integer> minCost = new HashMap<>();

// Remember which connection was used to reach each supplier at the lowest

cost.

Map<Supplier, Edge> previousEdge = new HashMap<>();

// A set of all suppliers that haven't been visitedSuppliers yet.

Set<Supplier> unvisitedSuppliers = new HashSet<>(suppliers);

// Create a graph (network of suppliers and connections).

Graph graph = new Graph();

for (Transporter transporter : transporters) {

// Add the transporter to the graph as an edge between the source and

destination suppliers.

graph.addCheapestPathEdge(transporter);

}

// Initially, set the cost to reach each supplier to the highest possible

value.

for (Supplier supplier : suppliers) {

minCost.put(supplier, Integer.MAX\_VALUE);

}

// Set the cost to reach the starting supplier (source) to 0.

minCost.put(source, 0);

// Loop until all suppliers have been visitedSuppliers.

List<Transporter> allocatedTransPath = determineCheapestPath(graph,

unvisitedSuppliers, minCost, previousEdge, source, destination);

// Return the list of transporters that form the cheapest path.

return allocatedTransPath;

}

public static Collection<Transporter> allocateForDemand(Collection<Supplier>

suppliers,

transporters) {

Collection<Transporter>

// TODO-Complete: Return an empty collection if either suppliers or

transporters is null

int surplus;

Collection<Transporter> allocation = new HashSet<>();

Supplier superSource = new Supplier("Super Source", 0, 0, 0,

Integer.MAX\_VALUE);

Supplier superSink = new Supplier("Super Sink", 0, 0, 0,

Integer.MAX\_VALUE);

// TODO-Complete: Return an empty collection if either suppliers or

transporters is null, stopping the method execution

if (suppliers == null || transporters == null) return allocation;

// Initialize the graph and remaining demand map

Graph graph = new Graph();

Map<Supplier, Integer> surplusMap = new HashMap<>();

// Initialize surplus map

for (Supplier supplier : suppliers) {

surplus = supplier.inventory() - supplier.demand();

surplusMap.put(supplier, surplus);

}

// TODO: Identify the source (manufacturer) and add a sink supplier(s) to

the graph

sink

graph.addSupplier(superSource);

graph.addSupplier(superSink);

// Create a relationship between suppliers and the super source and super

for (Supplier supplier : suppliers) {

graph.addSupplier(supplier);

if (supplier.demand() <= 0) {

// TODO-Complete: Add an edge from the source to each supplier with

a capacity equal to the supplier's demand and a cost of 0

graph.addEdge(superSource, supplier, surplusMap.get(supplier),

0); // Using surplus as capacity for manufacturers

} else {

// TODO-Complete: Add an edge from each supplier to the sink with a

capacity equal to the supplier's demand and a cost of 0

graph.addEdge(supplier, superSink, supplier.demand(), 0); // Using

demand as capacity for distributors

}

}

// Add transporters to the graph

for (Transporter transporter : transporters) {

graph.addEdge(transporter.from(), transporter.to(),

transporter.maxCapacity(), transporter.costPerUnit());

}

// TODO-Complete: Determine the max flow path and update the residual graph

while (findAugPath(graph, superSource, superSink)) updateResidual(graph,

superSink);

// Set allocations for transporters based on flow

for (Transporter transporter : transporters) {

Edge edge = graph.getEdge(transporter.from(), transporter.to());

if (edge != null) {

transporter.setAllocation(edge.flow);

allocation.add(transporter);

}

}

// Redistribute surplus inventory from manufacturers or depots if there's

any remaining capacity in the network

redistribSurp(suppliers, graph, surplusMap);

// Return transporters that have an allocation along the max flow path.

return allocation;

}

// ========= PRIMARY METHODS (ABOVE) =========

// ========= PRIVATE METHODS (BELOW) =========

private static void redistribSurp(Collection<Supplier> suppliers, Graph graph,

Map<Supplier, Integer> surplusMap) {

int surplus, allocation;

for (Supplier supplier : suppliers) {

// TODO-Complete: Iterate through the edges of the current supplier and

add the destination to the queue, and update the surplus map

if (surplusMap.get(supplier) > 0) {

for (Edge edge : graph.getEdgesFrom(supplier)) {

// Check if there's any remaining capacity in the network and

redistribute surplus inventory

if (edge.capacity > edge.flow) {

surplus = surplusMap.get(supplier);

allocation = Math.min(surplus, edge.capacity - edge.flow);

edge.flow += allocation;

surplusMap.put(supplier, surplus - allocation);

} }

} }

}

// TODO-Complete: This is a modified version of Dijkstra's algorithm - this

method finds the cheapest path from the source to each supplier.

private static List<Transporter> determineCheapestPath(Graph graph,

unvisitedSuppliers,

minCost,

Set<Supplier>

Map<Supplier, Integer>

Map<Supplier, Edge>

previousEdge,

Supplier destination) {

Supplier source,

// TODO-Complete: This list stores cheapest path from source -> destination

List<Transporter> cheapestTransPath = new ArrayList<>();

// Initialize a priority queue (typical impl for Dijkstra's) for selecting

next supplier by the lowest known cost

PriorityQueue<Supplier> queue = new

PriorityQueue<>(Comparator.comparing(minCost::get));

// Begin at the source supplier - then add the source to the priority queue

queue.add(source);

while (!queue.isEmpty()) {

// Poll the next for the next supplier with the lowest known cost

Supplier current = queue.poll();

// Mark the current supplier as visitedSuppliers

unvisitedSuppliers.remove(current);

// Break loop if the current supplier is the destination

if (current.equals(destination)) break;

// TODO-Complete: Verify that transporters (edges) that are connected

to the current supplier

for (Edge edge : graph.getEdgesFrom(current)) {

Supplier nextSupplier = edge.dest;

int newCost = minCost.get(current) + edge.cost; // Calculate new

cost to next supplier

// Update cost if the new cost is less than the current cost

if (newCost < minCost.get(nextSupplier)) {

// TODO-Complete: Update the cost to reach each supplier from

minCost.put(nextSupplier, newCost);

// Update the path to the next supplier

previousEdge.put(nextSupplier, edge);

// Add the next supplier to the queue to be processed next in

queue.add(nextSupplier);

}

} }

// TODO-Complete: Rebuild path from the dest. back to the source

Supplier currentSupplier = destination;

while (currentSupplier != null) {

// Get the edge leading to the current supplier

Edge edgeLeadingToCurrent = previousEdge.get(currentSupplier);

// If no edge leading to the current supplier, then exit the loop as we

have reached the source

if (edgeLeadingToCurrent == null) break;

// Add the transporter to the path

Transporter transporterForEdge =

graph.storeEdgeTransMap.get(edgeLeadingToCurrent);

cheapestTransPath.add(transporterForEdge);

// Move to the next supplier in the path

currentSupplier = edgeLeadingToCurrent.source;

the source

the loop

}

// Reverse the path to start from the source

Collections.reverse(cheapestTransPath);

// TODO-Complete: Return list of transporters that form the cheapest path

from source -> destination

return cheapestTransPath;

}

private static boolean findAugPath(Graph graph, Supplier source, Supplier sink)

{

Set<Supplier> visitedSuppliers = new HashSet<>();

Queue<Supplier> queue = new LinkedList<>();

Map<Supplier, Edge> p\_Map = new HashMap<>();

// Add the source and mark it as visitedSuppliers. reps. the starting point

for the BFS.

queue.add(source);

visitedSuppliers.add(source);

while (!queue.isEmpty()) {

Supplier current = queue.remove();

// TODO-Complete: Iterate through the edges of the current supplier and

add the destination to the queue

// - if it has not been visitedSuppliers yet and the capacity - flow

is greater than 0.

for (Edge edge : graph.getEdgesFrom(current)) {

if (visitedSuppliers.contains(edge.dest) || edge.capacity -

edge.flow <= 0) continue;

visitedSuppliers.add(edge.dest);

p\_Map.put(edge.dest, edge);

if (edge.dest.equals(sink)) {

// TODO-Complete: Set the parent map and return true if the

sink is found

}

graph.setP\_Map(p\_Map);

return true;

queue.add(edge.dest);

}

}

return false;

}

// TODO-Complete: This method updates the residual graph based on the max flow

path.

private static void updateResidual(Graph graph, Supplier sink) {

// TODO-Complete: Get the parent map from the graph

Map<Supplier, Edge> p\_Map = graph.getP\_Map();

// List of edges in the path and store the min flow.

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

int minFlow = Integer.MAX\_VALUE;

// Compute the minimum flow and store the edges in the path

Supplier current = sink;

while (p\_Map.containsKey(current)) {

edge  
}

}

edge.flow += minFlow; // Increase flow in forward edge

edge.reverseEdge.flow -= minFlow; // Reduction of the flow in reverse

Edge edge = p\_Map.get(current);

minFlow = Math.min(minFlow, edge.capacity - edge.flow);

// Add edge for updating the flow later on.

pathEdges.add(edge);

current = edge.source;

}

// Take the edge and update the flow in the forward and reverse edges.

for (Edge edge : pathEdges) {

// Private static inner class Graph

private static class Graph {

private final Map<Supplier, List<Edge>> adjacencyList;

private final Map<Edge, Transporter> storeEdgeTransMap;

private Map<Supplier, Edge> p\_Map;

private Graph() {

adjacencyList = new HashMap<>();

p\_Map = new HashMap<>();

storeEdgeTransMap = new HashMap<>();

}

private void addSupplier(Supplier supplier) {

adjacencyList.putIfAbsent(supplier, new ArrayList<>());

}

// add edge for cheapest path algorithm (Modified Dijkstra's)

private void addCheapestPathEdge(Transporter transporter) {

this.adjacencyList.putIfAbsent(transporter.from(), new ArrayList<>());

Edge edge = new Edge(transporter.from(), transporter.to(),

transporter.maxCapacity(), transporter.costPerUnit());

this.adjacencyList.get(transporter.from()).add(edge);

this.storeEdgeTransMap.put(edge, transporter); // Storing the

transporter corresponding to the edge

}

// add edge for ford-fulkerson algorithm

private void addEdge(Supplier source, Supplier dest, int capacity, int

cost) {

Edge edge = new Edge(source, dest, capacity, cost);

Edge reverseEdge = new Edge(dest, source, 0, -cost);

// TODO: Set the reverse edge for each edge

edge.reverseEdge = reverseEdge;

reverseEdge.reverseEdge = edge;

// TODO: Add the edge and reverse edge to the adjacency list

adjacencyList.get(source).add(edge);

// TODO-Complete: Add the reverse edge to the adjacency list

adjacencyList.get(dest).add(reverseEdge);

}

private Edge getEdge(Supplier source, Supplier dest) {

for (Edge edge : adjacencyList.get(source)) {

if (edge.dest.equals(dest)) {

return edge;

} }

return null;

}

private List<Edge> getEdgesFrom(Supplier source) {

return adjacencyList.getOrDefault(source, new ArrayList<>());

}

private void setP\_Map(Map<Supplier, Edge> p\_Map) {

this.p\_Map = p\_Map;

}

private Map<Supplier, Edge> getP\_Map() {

return this.p\_Map;

}

}

// Private static inner class Edge

private static class Edge {

private final Supplier source;

private final Supplier dest;

private final int capacity;

private final int cost;

private int flow;

private Edge reverseEdge;

private Edge(Supplier source, Supplier dest, int capacity, int cost) {

this.source = source;

this.dest = dest;

this.capacity = capacity;

this.cost = cost;

this.flow = 0;

}

}

// ========= PRIVATE METHODS (ABOVE) =========

// ========= UNMODIFIED UTILITIES (BELOW) =========

/\*\*

\* Utility that determines the total demand of a collection of suppliers.

\* @param suppliers the suppliers

\* @return the sum of the demands of the suppliers

\*/

public static int totalDemand(Collection<Supplier> suppliers) {

int sum = 0;

for(Supplier s : suppliers) {

sum += s.demand();

}

return sum; }

/\*\*

\* Utility that determines the total amount shipped by transporters.

\* @param transporters the transporters

\* @return the sum of the allocations of the transporters

\*/

public static int totalAmountShipped(Collection<Transporter> transporters) {

int sum = 0;

for(Transporter t : transporters) {

sum += t.allocation();

}

return sum; }

/\*\*

\* Utility that determines the total transportation cost of all allocated

shipments.

\* @param transporters the transporters

\* @return the sum of the transportation costs based on allocations of the

transporters

\*/

public static int totalTransporterCost(Collection<Transporter> transporters) {

return transporters.stream()

.map(t -> t.allocation() \* t.costPerUnit())

.reduce(0, Integer::sum);

// Explicit loop version

// int sum = 0;

// for(Transporter t : transporters) {

// sum += t.allocation() \* t.costPerUnit();

// }

// return sum;

}

/\*\*

\* Generate a printable string with a line for each

\* transporter showing its name, allocation, and cost per unit.

\* @param transporters the transporters

\* @return a displayable string showing allocation and unit-cost for each

transporter

\*/

public static String displayAllocations(Collection<Transporter> transporters) {

return transporters.stream()

.map(t -> String.format("%s: %s $%s", t.name(), t.allocation(),

t.costPerUnit()))

.collect(Collectors.joining("\n"));

// Explicit loop version

// String str = "";

// for(Transporter t : transporters) {

// str += String.format("%s: %s $%s\n", t.name(), t.allocation(),

t.costPerUnit());

// }

// return str;

}

/\*\* Hide constructor of this utility class. \*/

private SimpleAllocator() { }

}