Homework 10: Graph Algorithms

● Graded

Student

Vidit Dharmendra Pokharna

Total Points

99 / 100 pts

Autograder Score

99.0 / 100.0

Failed Tests

Encapsulation (-1/0)

Question 2

Feedback & Manual Grading

0 / 0 pts

✓ **+ 0 pts** Correct

[-1] encapsulation: dfsHelper should be a private helper method

Great work :) -Isabelle □‿□
You’re done with 1332 homework!

Autograder Results

Autograder Output

If you're seeing this message, everything compiled and ran properly!
-CS1332 TAs

Encapsulation (-1/0)

Added non-private method: dfsHelper (line 125)

Submitted Files

GraphAlgorithms.java

Download

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Set;

/**
 * Your implementation of various different graph algorithms.
 *
 * @author Vidit Pokharna
 * @userid vpokharna3
 * @GTID 903772087
 * @version 1.0
 */
public class GraphAlgorithms {

    /**
     * Performs a breadth first search (bfs) on the input graph, starting at
     * the parameterized starting vertex.
     *
     * When exploring a vertex, explore in the order of neighbors returned by
     * the adjacency list. Failure to do so may cause you to lose points.
     *
     * You may import/use java.util.Set, java.util.List, java.util.Queue, and
     * any classes that implement the aforementioned interfaces, as long as they
     * are efficient.
     *
     * The only instance of java.util.Map that you may use is the
     * adjacency list from graph. DO NOT create new instances of Map
     * for BFS (storing the adjacency list in a variable is fine).
     *
     * DO NOT modify the structure of the graph. The graph should be unmodified
     * after this method terminates.
     *
     * @param <T>   the generic typing of the data
     * @param start the vertex to begin the bfs on
     * @param graph the graph to search through
     * @return list of vertices in visited order
     * @throws IllegalArgumentException if any input is null, or if start
     *                                  doesn't exist in the graph
     */
    public static <T> List<Vertex<T>> bfs(Vertex<T> start, Graph<T> graph) {
        if (start == null || graph == null) {
            throw new IllegalArgumentException("At least one of the inputted parameters is null");
        } else if (!(graph.getVertices().contains(start))) {
            throw new IllegalArgumentException("The start vertex is not within the graph");
        }

        HashSet<Vertex<T>> visitedSet = new HashSet<>();
        Queue<Vertex<T>> queue = new LinkedList<>();
        List<Vertex<T>> finalList = new ArrayList<>();

        visitedSet.add(start);
        queue.add(start);

        while (!queue.isEmpty()) {
            Vertex<T> t = queue.remove();
            finalList.add(t);
            for (VertexDistance<T> w : graph.getAdjList().get(t)) {
                if (!(visitedSet.contains(w.getVertex()))) {
                    queue.add(w.getVertex());
                    visitedSet.add(w.getVertex());
                }
            }
        }

        return finalList;
    }

    /**
     * Performs a depth first search (dfs) on the input graph, starting at
     * the parameterized starting vertex.
     *
     * When exploring a vertex, explore in the order of neighbors returned by
     * the adjacency list. Failure to do so may cause you to lose points.
     *
     * *NOTE* You MUST implement this method recursively, or else you will lose
     * all points for this method.
     *
     * You may import/use java.util.Set, java.util.List, and
     * any classes that implement the aforementioned interfaces, as long as they
     * are efficient.
     *
     * The only instance of java.util.Map that you may use is the
     * adjacency list from graph. DO NOT create new instances of Map
     * for DFS (storing the adjacency list in a variable is fine).
     *
     * DO NOT modify the structure of the graph. The graph should be unmodified
     * after this method terminates.
     *
     * @param <T>   the generic typing of the data
     * @param start the vertex to begin the dfs on
     * @param graph the graph to search through
     * @return list of vertices in visited order
     * @throws IllegalArgumentException if any input is null, or if start
     *                                  doesn't exist in the graph
     */
    public static <T> List<Vertex<T>> dfs(Vertex<T> start, Graph<T> graph) {
        if (start == null || graph == null) {
            throw new IllegalArgumentException("At least one of the inputted parameters is null");
        } else if (!(graph.getVertices().contains(start))) {
            throw new IllegalArgumentException("The start vertex is not within the graph");
        }

        HashSet<Vertex<T>> visitedSet = new HashSet<>();
        List<Vertex<T>> finalList = new ArrayList<>();

        dfsHelper(start, graph, visitedSet, finalList);
        return finalList;
    }

    /**
     * Helper method for dfs
     *
     * @param <T>   the generic typing of the data
     * @param start the vertex to begin the dfs on
     * @param graph the graph to search through
     * @param visitedSet the set of all visited vertices
     * @param finalResult the list of vertices to return
     */
    public static <T> void dfsHelper(Vertex<T> start, Graph<T> graph,
                                         HashSet<Vertex<T>> visitedSet, List<Vertex<T>> finalResult) {
        visitedSet.add(start);
        finalResult.add(start);

        for (VertexDistance<T> w : graph.getAdjList().get(start)) {
            if (!(visitedSet.contains(w.getVertex()))) {
                dfsHelper(w.getVertex(), graph, visitedSet, finalResult);
            }
        }
    }

    /**
     * Finds the single-source shortest distance between the start vertex and
     * all vertices given a weighted graph (you may assume non-negative edge
     * weights).
     *
     * Return a map of the shortest distances such that the key of each entry
     * is a node in the graph and the value for the key is the shortest distance
     * to that node from start, or Integer.MAX_VALUE (representing
     * infinity) if no path exists.
     *
     * You may import/use java.util.PriorityQueue,
     * java.util.Map, and java.util.Set and any class that
     * implements the aforementioned interfaces, as long as your use of it
     * is efficient as possible.
     *
     * You should implement the version of Dijkstra's where you use two
     * termination conditions in conjunction.
     *
     * 1) Check if all of the vertices have been visited.
     * 2) Check if the PQ is empty.
     *
     * DO NOT modify the structure of the graph. The graph should be unmodified
     * after this method terminates.
     *
     * @param <T>   the generic typing of the data
     * @param start the vertex to begin the Dijkstra's on (source)
     * @param graph the graph we are applying Dijkstra's to
     * @return a map of the shortest distances from start to every
     * other node in the graph
     * @throws IllegalArgumentException if any input is null, or if start
     *                                  doesn't exist in the graph.
     */
    public static <T> Map<Vertex<T>, Integer> dijkstras(Vertex<T> start,
                                                        Graph<T> graph) {
        if (start == null || graph == null) {
            throw new IllegalArgumentException("At least one of the inputted parameters is null");
        } else if (!(graph.getVertices().contains(start))) {
            throw new IllegalArgumentException("The start vertex is not within the graph");
        }

        HashSet<Vertex<T>> visitedSet = new HashSet<>();
        HashMap<Vertex<T>, Integer> distanceMap = new HashMap<>();
        PriorityQueue<VertexDistance<T>> priorityQueue = new PriorityQueue<>();

        for (Vertex<T> v : graph.getAdjList().keySet()) {
            if (!(v.equals(start))) {
                distanceMap.put(v, Integer.MAX_VALUE);
            } else {
                distanceMap.put(v, 0);
            }
        }

        priorityQueue.add(new VertexDistance<>(start, 0));

        int numOfVertices = graph.getVertices().size();

        while (!(priorityQueue.isEmpty()) && (visitedSet.size() <= numOfVertices)) {
            VertexDistance<T> ud = priorityQueue.remove();
            for (VertexDistance<T> w : graph.getAdjList().get(ud.getVertex())) {
                int distance = w.getDistance() + ud.getDistance();
                if (distanceMap.get(w.getVertex()) > distance) {
                    distanceMap.put(w.getVertex(), distance);
                    priorityQueue.add(new VertexDistance<>(w.getVertex(), distance));
                }
            }
        }

        return distanceMap;
    }

    /**
     * Runs Prim's algorithm on the given graph and returns the Minimum
     * Spanning Tree (MST) in the form of a set of Edges. If the graph is
     * disconnected and therefore no valid MST exists, return null.
     *
     * You may assume that the passed in graph is undirected. In this framework,
     * this means that if (u, v, 3) is in the graph, then the opposite edge
     * (v, u, 3) will also be in the graph, though as a separate Edge object.
     *
     * The returned set of edges should form an undirected graph. This means
     * that every time you add an edge to your return set, you should add the
     * reverse edge to the set as well. This is for testing purposes. This
     * reverse edge does not need to be the one from the graph itself; you can
     * just make a new edge object representing the reverse edge.
     *
     * You may assume that there will only be one valid MST that can be formed.
     *
     * You should NOT allow self-loops or parallel edges in the MST.
     *
     * You may import/use PriorityQueue, java.util.Set, and any class that
     * implements the aforementioned interface.
     *
     * DO NOT modify the structure of the graph. The graph should be unmodified
     * after this method terminates.
     *
     * The only instance of java.util.Map that you may use is the
     * adjacency list from graph. DO NOT create new instances of Map
     * for this method (storing the adjacency list in a variable is fine).
     *
     * @param <T> the generic typing of the data
     * @param start the vertex to begin Prims on
     * @param graph the graph we are applying Prims to
     * @return the MST of the graph or null if there is no valid MST
     * @throws IllegalArgumentException if any input is null, or if start
     *                                  doesn't exist in the graph.
     */
```

```java
    public static <T> Set<Edge<T>> prims(Vertex<T> start, Graph<T> graph) {
        if (start == null || graph == null) {
            throw new IllegalArgumentException("At least one of the inputted parameters is null");
        } else if (!(graph.getVertices().contains(start))) {
            throw new IllegalArgumentException("The start vertex is not within the graph");
        }

        HashSet<Vertex<T>> visitedSet = new HashSet<>();
        HashSet<Edge<T>> mst = new HashSet<>();
        PriorityQueue<Edge<T>> priorityQueue = new PriorityQueue<>();

        visitedSet.add(start);

        for (Edge<T> edge : graph.getEdges()) {
            if (edge.getU().equals(start)) {
                priorityQueue.add(edge);
            }
        }

        while (!priorityQueue.isEmpty()) {
            Edge<T> uw = priorityQueue.remove();
            if (!(visitedSet.contains(uw.getV())) || !(visitedSet.contains(uw.getU()))) {
                visitedSet.add(uw.getV());
                mst.add(uw);
                mst.add(new Edge<>(uw.getV(), uw.getU(), uw.getWeight()));
                for (Edge<T> wx : graph.getEdges()) {
                    if (wx.getU().equals(uw.getV()) && !visitedSet.contains(wx.getV())) {
                        priorityQueue.add(wx);
                    }
                }
            }
        }

        if (mst.size() < (graph.getVertices().size() - 1) * 2) {
            return null;
        }

        return mst;
    }
}
```