

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

```
1  import java.util.ArrayList;
2  import java.util.HashMap;
3  import java.util.HashSet;
4  import java.util.LinkedList;
5  import java.util.List;
6  import java.util.Map;
7  import java.util.PriorityQueue;
8  import java.util.Queue;
9  import java.util.Set;
10
11  /**
12   * Your implementation of various different graph algorithms.
13   *
14   * @author Vidit Pokharna
15   * @userid vpokharna3
16   * @GTID 903772087
17   * @version 1.0
18   */
19  public class GraphAlgorithms {
20
21      /**
22       * Performs a breadth first search (bfs) on the input graph, starting at
23       * the parameterized starting vertex.
24       *
25       * When exploring a vertex, explore in the order of neighbors returned by
26       * the adjacency list. Failure to do so may cause you to lose points.
27       *
28       * You may import/use java.util.Set, java.util.List, java.util.Queue, and
29       * any classes that implement the aforementioned interfaces, as long as they
30       * are efficient.
31       *
32       * The only instance of java.util.Map that you may use is the
33       * adjacency list from graph. DO NOT create new instances of Map
34       * for BFS (storing the adjacency list in a variable is fine).
35       *
36       * DO NOT modify the structure of the graph. The graph should be unmodified
37       * after this method terminates.
38       *
39       * @param <T> the generic typing of the data
40       * @param start the vertex to begin the bfs on
41       * @param graph the graph to search through
42       * @return list of vertices in visited order
43       * @throws IllegalArgumentException if any input is null, or if start
44       *         doesn't exist in the graph
45       */
46      public static <T> List<Vertex<T>> bfs(Vertex<T> start, Graph<T> graph) {
```

```

47     if (start == null || graph == null) {
48         throw new IllegalArgumentException("At least one of the inputted parameters is null");
49     } else if (!(graph.getVertices().contains(start))) {
50         throw new IllegalArgumentException("The start vertex is not within the graph");
51     }
52
53     HashSet<Vertex<T>> visitedSet = new HashSet<>();
54     Queue<Vertex<T>> queue = new LinkedList<>();
55     List<Vertex<T>> finalList = new ArrayList<>();
56
57     visitedSet.add(start);
58     queue.add(start);
59
60     while (!queue.isEmpty()) {
61         Vertex<T> t = queue.remove();
62         finalList.add(t);
63         for (VertexDistance<T> w : graph.getAdjList().get(t)) {
64             if (!(visitedSet.contains(w.getVertex()))) {
65                 queue.add(w.getVertex());
66                 visitedSet.add(w.getVertex());
67             }
68         }
69     }
70
71     return finalList;
72 }
73
74 /**
75  * Performs a depth first search (dfs) on the input graph, starting at
76  * the parameterized starting vertex.
77  *
78  * When exploring a vertex, explore in the order of neighbors returned by
79  * the adjacency list. Failure to do so may cause you to lose points.
80  *
81  * *NOTE* You MUST implement this method recursively, or else you will lose
82  * all points for this method.
83  *
84  * You may import/use java.util.Set, java.util.List, and
85  * any classes that implement the aforementioned interfaces, as long as they
86  * are efficient.
87  *
88  * The only instance of java.util.Map that you may use is the
89  * adjacency list from graph. DO NOT create new instances of Map
90  * for DFS (storing the adjacency list in a variable is fine).
91  *
92  * DO NOT modify the structure of the graph. The graph should be unmodified
93  * after this method terminates.
94  *
95  * @param <T> the generic typing of the data

```

```

96  * @param start the vertex to begin the dfs on
97  * @param graph the graph to search through
98  * @return list of vertices in visited order
99  * @throws IllegalArgumentException if any input is null, or if start
100 *         doesn't exist in the graph
101 */
102 public static <T> List<Vertex<T>> dfs(Vertex<T> start, Graph<T> graph) {
103     if (start == null || graph == null) {
104         throw new IllegalArgumentException("At least one of the inputted parameters is null");
105     } else if (!(graph.getVertices().contains(start))) {
106         throw new IllegalArgumentException("The start vertex is not within the graph");
107     }
108
109     HashSet<Vertex<T>> visitedSet = new HashSet<>();
110     List<Vertex<T>> finalList = new ArrayList<>();
111
112     dfsHelper(start, graph, visitedSet, finalList);
113     return finalList;
114 }
115
116 /**
117  * Helper method for dfs
118  *
119  * @param <T> the generic typing of the data
120  * @param start the vertex to begin the dfs on
121  * @param graph the graph to search through
122  * @param visitedSet the set of all visited vertices
123  * @param finalResult the list of vertices to return
124  */
125 public static <T> void dfsHelper(Vertex<T> start, Graph<T> graph,
126                                 HashSet<Vertex<T>> visitedSet, List<Vertex<T>> finalResult) {
127     visitedSet.add(start);
128     finalResult.add(start);
129
130     for (VertexDistance<T> w : graph.getAdjList().get(start)) {
131         if (!(visitedSet.contains(w.getVertex())))) {
132             dfsHelper(w.getVertex(), graph, visitedSet, finalResult);
133         }
134     }
135 }
136
137 /**
138  * Finds the single-source shortest distance between the start vertex and
139  * all vertices given a weighted graph (you may assume non-negative edge
140  * weights).
141  *
142  * Return a map of the shortest distances such that the key of each entry
143  * is a node in the graph and the value for the key is the shortest distance
144  * to that node from start, or Integer.MAX_VALUE (representing

```

```

145 * infinity) if no path exists.
146 *
147 * You may import/use java.util.PriorityQueue,
148 * java.util.Map, and java.util.Set and any class that
149 * implements the aforementioned interfaces, as long as your use of it
150 * is efficient as possible.
151 *
152 * You should implement the version of Dijkstra's where you use two
153 * termination conditions in conjunction.
154 *
155 * 1) Check if all of the vertices have been visited.
156 * 2) Check if the PQ is empty.
157 *
158 * DO NOT modify the structure of the graph. The graph should be unmodified
159 * after this method terminates.
160 *
161 * @param <T> the generic typing of the data
162 * @param start the vertex to begin the Dijkstra's on (source)
163 * @param graph the graph we are applying Dijkstra's to
164 * @return a map of the shortest distances from start to every
165 * other node in the graph
166 * @throws IllegalArgumentException if any input is null, or if start
167 * doesn't exist in the graph.
168 */
169 public static <T> Map<Vertex<T>, Integer> dijkstras(Vertex<T> start,
170                                     Graph<T> graph) {
171     if (start == null || graph == null) {
172         throw new IllegalArgumentException("At least one of the inputted parameters is null");
173     } else if (!(graph.getVertices().contains(start))) {
174         throw new IllegalArgumentException("The start vertex is not within the graph");
175     }
176
177     HashSet<Vertex<T>> visitedSet = new HashSet<>();
178     HashMap<Vertex<T>, Integer> distanceMap = new HashMap<>();
179     PriorityQueue<VertexDistance<T>> priorityQueue = new PriorityQueue<>();
180
181     for (Vertex<T> v : graph.getAdjList().keySet()) {
182         if (!(v.equals(start))) {
183             distanceMap.put(v, Integer.MAX_VALUE);
184         } else {
185             distanceMap.put(v, 0);
186         }
187     }
188
189     priorityQueue.add(new VertexDistance<>(start, 0));
190
191     int numOfVertices = graph.getVertices().size();
192
193     while (!(priorityQueue.isEmpty()) && (visitedSet.size() <= numOfVertices)) {

```

```

194     VertexDistance<T> ud = priorityQueue.remove();
195     for (VertexDistance<T> w : graph.getAdjList().get(ud.getVertex())) {
196         int distance = w.getDistance() + ud.getDistance();
197         if (distanceMap.get(w.getVertex()) > distance) {
198             distanceMap.put(w.getVertex(), distance);
199             priorityQueue.add(new VertexDistance<>(w.getVertex(), distance));
200         }
201     }
202 }
203
204     return distanceMap;
205 }
206
207 /**
208  * Runs Prim's algorithm on the given graph and returns the Minimum
209  * Spanning Tree (MST) in the form of a set of Edges. If the graph is
210  * disconnected and therefore no valid MST exists, return null.
211  *
212  * You may assume that the passed in graph is undirected. In this framework,
213  * this means that if (u, v, 3) is in the graph, then the opposite edge
214  * (v, u, 3) will also be in the graph, though as a separate Edge object.
215  *
216  * The returned set of edges should form an undirected graph. This means
217  * that every time you add an edge to your return set, you should add the
218  * reverse edge to the set as well. This is for testing purposes. This
219  * reverse edge does not need to be the one from the graph itself; you can
220  * just make a new edge object representing the reverse edge.
221  *
222  * You may assume that there will only be one valid MST that can be formed.
223  *
224  * You should NOT allow self-loops or parallel edges in the MST.
225  *
226  * You may import/use PriorityQueue, java.util.Set, and any class that
227  * implements the aforementioned interface.
228  *
229  * DO NOT modify the structure of the graph. The graph should be unmodified
230  * after this method terminates.
231  *
232  * The only instance of java.util.Map that you may use is the
233  * adjacency list from graph. DO NOT create new instances of Map
234  * for this method (storing the adjacency list in a variable is fine).
235  *
236  * @param <T> the generic typing of the data
237  * @param start the vertex to begin Prim's on
238  * @param graph the graph we are applying Prim's to
239  * @return the MST of the graph or null if there is no valid MST
240  * @throws IllegalArgumentException if any input is null, or if start
241  *         doesn't exist in the graph.
242  */

```

```

243 public static <T> Set<Edge<T>> prims(Vertex<T> start, Graph<T> graph) {
244     if (start == null || graph == null) {
245         throw new IllegalArgumentException("At least one of the inputted parameters is null");
246     } else if (!(graph.getVertices().contains(start))) {
247         throw new IllegalArgumentException("The start vertex is not within the graph");
248     }
249
250     HashSet<Vertex<T>> visitedSet = new HashSet<>();
251     HashSet<Edge<T>> mst = new HashSet<>();
252     PriorityQueue<Edge<T>> priorityQueue = new PriorityQueue<>();
253
254     visitedSet.add(start);
255
256     for (Edge<T> edge : graph.getEdges()) {
257         if (edge.getU().equals(start)) {
258             priorityQueue.add(edge);
259         }
260     }
261
262     while (!priorityQueue.isEmpty()) {
263         Edge<T> uw = priorityQueue.remove();
264         if (!(visitedSet.contains(uw.getV()) || !visitedSet.contains(uw.getU()))) {
265             visitedSet.add(uw.getV());
266             mst.add(uw);
267             mst.add(new Edge<>(uw.getV(), uw.getU(), uw.getWeight()));
268             for (Edge<T> wx : graph.getEdges()) {
269                 if (wx.getU().equals(uw.getV()) && !visitedSet.contains(wx.getV())) {
270                     priorityQueue.add(wx);
271                 }
272             }
273         }
274     }
275
276     if (mst.size() < (graph.getVertices().size() - 1) * 2) {
277         return null;
278     }
279
280     return mst;
281 }
282 }

```