Version: 12-5-2015 9:15 AM

CS 2223 B15 Term. Homework 5

Homework Instructions

- This homework is to be completed individually. If you have any questions as to what constitutes improper behavior, review the examples as I have posted online http://web.cs.wpi.edu/~heineman/html/teaching_/cs2223/b15/#policies .
- Due Date for this assignment is 2PM Friday December 11th. Homeworks received after 2PM receive a 25% late penalty. Homeworks received after 6PM will receive zero credit.
- Submit your assignments electronically using the blackboard site for CS2223. Login to **my.wpi.edu** and go to CS2223 under "My Courses" then go to "Assignments" and submit your homework under "HW5". You must submit a single ZIP file that contains all of your code as well as the written answers to the assignment.
- All of your Java classes must be defined in a packager USERID.hw5 where USERID is your CCC user id (i.e., your email address without the @wpi.edu).

Primary Instructions

Submit your written answers in a file writtenAnswers.txt that you submit with your assignment.

Version: 12-5-2015 9:15 AM

Q1. Graph Exercise (20 pts)

(a) Write a method in **`Graph`** that returns a new **`Graph`** object which is a complement of the graph. Define $\bar{g}$ to be the complement of a graph, g, as follows:

- $\bar{g}$ has the same number of vertices as g, identified by index 0 to V-1 where V is the number of vertices in g
- In $\bar{g}$, the edge (*u*, *v*) exists if the edge (*u*, *v*) does not exist in g
- In $\bar{g}$, the edge (*u*, *v*) does not exist if the edge (*u*, *v*) exists in g

```
public Graph complement() { … }
```

(b) Demonstrate this method works by computing the complement of an empty graph with five vertices. Output the contents of both graphs (g and $\bar{g}$) using the **`toString()`** representation

(c) Write a method that determines whether a graph is *connected*, that is, if there is a path from every vertex to every other vertex in the graph.

```
public boolean connected() { … }
```

(d) Construct an example graph, g, with five vertices that has the following property:

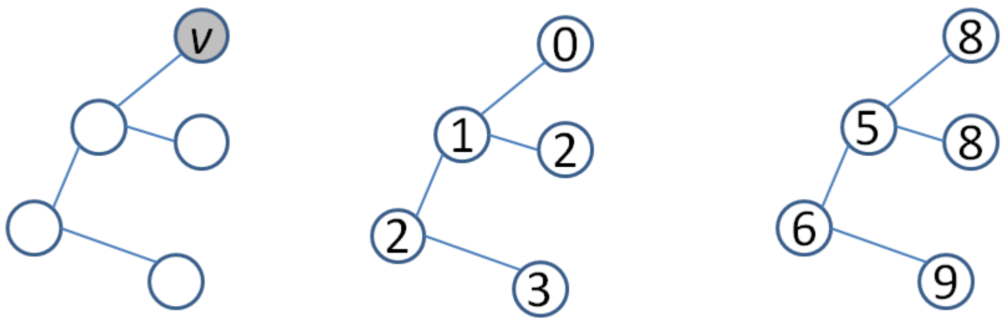
- g is connected
- $\bar{g}$ is connected

Validate your result by writing a program that shows the **`toString()`** representation of these graphs as well as the output of calling the **`connected()`** method on each graph.

Version: 12-5-2015 9:15 AM

Q2. Status Injective Graphs (25 pts)

You are given a simple, undirected connected graph G = (V, E). For any vertex, v, in this graph you can compute its *status*. The *status* of *v* is the sum of the shortest distances to every vertex in the graph (recall that the distance of a vertex to itself is zero). Consider the graph below and designated vertex *v*.



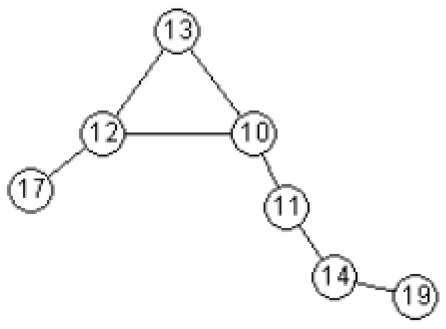
Status[v] = 0+1+2+2+3 =8 Shortest distances from *v* All Status Values

The status of *v* is 8, since that is the accumulated sum of the shortest distance from *v* to every other vertex in the graph. Note that this is **not** a status injective graph because two vertices have a computed status of 8 as shown above.

(a) Modify the **Graph** class to complete the implementation of the **`status(v)`** method which returns the computed status for a given vertex, *v*, in the graph.

```
public int status(int v) { … }
```

(b) A graph is classified as being *status injective* if the calculated status values for all nodes in the graph are different integers. To convince you that such graphs exist, consider the following graph which draws each vertex with its computed status value. As you can see, all of these values are distinct, thus this is an example of a status injective graph.



```
public boolean statusInjective() { … }
```

(c) **Bonus Question (5 pts.)**: Write a program to find a status injective graph with six vertices, or demonstrate that no such graph can be found. Hint: Try to methodically create all 2^{15} possible graphs.

Q3. Graph Proof and Demonstration (25 pts)

We have begun introducing proofs into the lectures, and the textbook also contains samples throughout the different chapters. It is time for you to try your hand at a proof as it relates to an undirected graph.

(a) **[10 pts]** Prove that every connected graph of N vertices contains some *safe vertex*, **v**, whose removal (including all edges incident **v**) will not disconnect the graph.

You may make use of the following facts:

- A graph with no vertices is also, by definition, considered to be connected
- A graph with a single vertex is, by definition, connected. Note that there are no edges in such a graph

Hint: Depth First Search should be useful. Consider a vertex whose adjacent vertices are all marked.

(b) **[5 pts]** Write a method **`findSafeVertex()`** in the **`Graph`** class that returns a vertex in a connected graph which can be deleted and enable the graph to remain connected. Note if the graph is initially not connected, then -1 must be returned. You should check this first.

```
public int findSafeVertex () { … }
```

(c) **[10 pts]** The *eccentricity* of a vertex, *v*, is the length of the shortest path from that vertex to the furthest vertex from *v* in the graph. Naturally the graph must be connected for this property to make sense. With this definition in place, we can define the *diameter* of a graph to be the maximum eccentric of any of its vertices. Add the following method to the **`Graph`** class, which returns -1 if the graph is not connected.

```
public int diameter () { … }
```

Version: 12-5-2015 9:15 AM

Q4. Complete DiGraphMatrix implementation – 30 points

The book consistently uses *adjacency lists* to represent a graph. You will gain some experience in writing an implementation that uses a two-dimensional *adjacency matrix* of size V x V to store a directed graph with edge weights.

(a) **[10 pts.]** Complete the implementation of **`DiGraphMatrix`** as found in the **`algs.hw5`** package.

(b) **[15 pts.]** We will cover in class the **Single Source Shortest Path** algorithm also known as **Dijkstra's Algorithm** (p. 652) of the book. You will implement an alternate version that works with an adjacency matrix representation of the graph. The specification for this algorithm is given in pseudocode below. Often you will see algorithms sketched out using similar pseudocode descriptions and so you will need to learn how to convert these high-level specifications into actual code. In this case, your goal is to compute the dist[] and pred[] matrices and output the dist[] values which records the shortest total path lengths from a single source, s, to all other vertices in the graph.

(c) **[5 pts.]** Demonstrate the correctness of your implementation on the sample graph provided in the figure.

Dijkstra's Algorithm DG

Best	Average	Worst
$O(V^2+E)$	$O(V^2+E)$	$O(V^2+E)$

Weighted Directed Graph

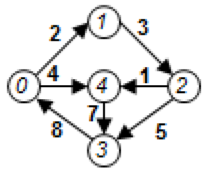
Array

Overflow

```
singleSourceShortest (G, s)
1.  set dist[v] to ∞ for all v∈G
2.  set pred[v] to –1 for all v∈G
3.  set visited[v] to false for all v ∈G
4.  dist[s] = 0

5.  while (true)
6.     determine u whose dist[u] is
       smallest of unvisited vertices
7.     if (dist[u] = ∞) then return
8.     visited[u] = true
9.     foreach neighbor v of u do
10.       w = weight of edge (u,v)
11.       newLen = dist[u]+w
12.       if (newLen < dist[v]) then
13.          dist[v] = newLen
14.          prev[v] = u
end
```

Initialize dist[v] and visited[v] with s=0



dist[v]

0	∞	∞	∞	∞

visited[v]

If total length of the path from (**s**,**u**) followed by (**u**,**v**) is shorter than the best distance from (**s**,**v**) adjust dist[v].

1st iteration: **u** = vertex **0**

(0,0)+(0,4)<(0,4)
(0,0)+(0,1)<(0,1)

dist	0	2	∞	∞	4
visited	✓				

2nd iteration: **u**= vertex **1**

(0,1)+(1,2)<(0,2)

dist	0	2	5	∞	4
visited	✓	✓			

3rd iteration: **u** = vertex **4**

(0,4)+(4,3)<(0,3)

dist	0	2	5	11	4
visited	✓	✓			✓

4th iteration: **u** = vertex **2**

(0,2)+(2,3)<(0,3)

dist	0	2	5	10	4
visited	✓	✓	✓		✓

On 5th iteration: **u** = vertex **3** causes no change to **dist**, but changes **visited**.