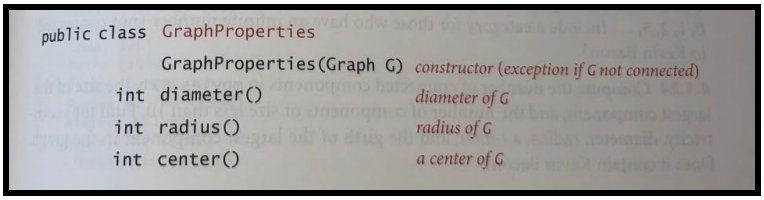
Nathan Stouffer
CSCI 232, Homework 3

Exercise 4.1.16

Problem. The *eccentricity* of a vertex v is the length of the shortest path from that vertex to the furthest vertex from v. The *diameter* of a graph is the maximum eccentricity of any vertex. The *radius* of a graph is the smallest eccentricity of a vertex. A *center* is a vertex whose eccentricity is the radius. Describe the algorithms you would implement to have the following API.



```
public class GraphProperties
                GraphProperties(Graph G)  constructor (exception if G not connected)
           int  diameter()                diameter of G
           int  radius()                  radius of G
           int  center()                  a center of G
```

Answer. The description of each method is listed below.

Constructor:

Perform a one time calculation on the graph G that computes the eccentricity of every vertex. The constructor should do this by performing Dijkstra's algorithm on each vertex v and then insert the longest path found from Dijkstra's algorithm for vertex v into an array of the form {eccentricity, source}. Then insert this value into a global array *arr* that is sorted by eccentricity.

Diameter:

Return the 0th value of the maximum value in the global array *arr*.

Radius:

Return the 0th value of the minimum value in the global array *arr*.

Center:

Returns the 1st value of minimum value in the global array *arr.* (Arbitrary decision of which center vertex to return)

Nathan Stouffer
CSCI 232, Homework 3

Exercise 4.1.19

Problem. Show, in the style of the figure on page 545, a detailed trace of CC for finding the connected components in the graph built by Graph's input stream constructor for the file tinyGex2.txt.

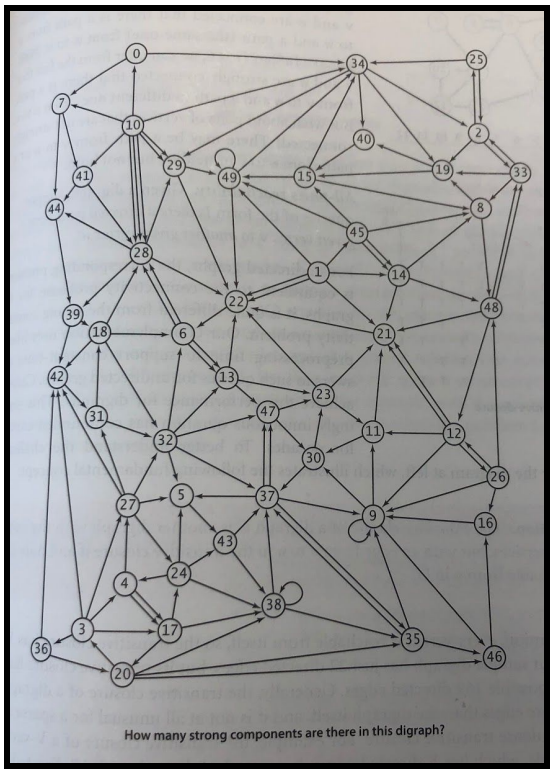
Solution. The trace is in the table below

		count	marked[]											id[]										
			0	1	2	3	4	5	6	7	8	9	10	11	0	1	2	3	4	5	6	7	8	9	10	11
dfs(0)		0	T												0											
	dfs(6)	0	T						T						0						0					
	check 0																									
	dfs(3)	0	T			T			T						0			0			0					
	check 6																									
	dfs(10)	0	T			T			T				T		0			0			0				0	
	check 3																									
	dfs(5)	0	T			T		T	T				T		0			0		0	0				0	
	dfs(2)	0	T		T	T		T	T				T		0		0	0		0	0				0	
	check 3																									
	check 0																									
	check 6																									
	check 5																									
	2 done																									
	check 10																									
	check 0																									
	5 done																									
	check 5																									
	10 done																									
	check 2																									
	check 10																									
	3 done																									
	check 2																									
	6 done																									
	check 2																									
	check 5																									
0 done																										
dfs(1)		1	T	T	T	T		T	T				T		0	1	0	0		0	0				0	
	dfs(11)	1	T	T	T	T		T	T				T	T	0	1	0	0							0	1
	check 1																									
	dfs(7)	1	T	T	T	T		T	T	T			T	T	0	1	0	0		0	0	1			0	1
	check 11																									
	dfs(8)	1	T	T	T	T		T	T	T	T		T	T	0	1	0	0		0	0	1	1		0	1
	dfs(4)	1	T	T	T	T	T	T	T	T	T		T	T	0	1	0	0	1	0	0	1	1		0	1
	check 8																									
	check 8																									
	4 done																									
	check 7																									
	check 11																									
	check 1																									
	8 done																									
	7 done																									
	check 8																									
	11 done																									
	check 8																									
	check 4																									
1 done																										
dfs(9)		2	T	T	T	T	T	T	T	T	T	T	T	T	0	1	0	0	1	0	0	1	1	2	0	1
9 done																										

Nathan Stouffer
CSCI 232, Homework 3

Exercise 4.2.17

Problem. How many strong components are there in the digraph on page 591?



Solution. To find the number of strong components in the digraph on page 591, I downloaded CC.java and necessary files for CC.java to run. I then renamed CC.java to StronglyCC.java and modified the file to return the number of strongly connected components in a digraph. As an input file, I used the text file that the online textbook claimed related to the digraph shown above. However, I discovered there were some differences between the two graphs. Grade this as you see fit. Based on the input text file, my algorithm said there were 3 strongly connected components in the graph. The files are attached to this submission.

Nathan Stouffer
CSCI 232, Homework 3

Exercise 4.2.18

Problem. What are the strong components of a DAG?

Solution. The strong components of a DAG are the vertices themselves. In other words, an arbitrary vertex *v* is only strongly connected to that same vertex *v.* This is shown through proof by contradiction. That is, assume that some vertex *v* in a Directed Acyclic Graph *G* is strongly connected with another vertex *w* in *G*. Since a strong connection is symmetric, vertex *w* is also connected to *v.* Since *v* is connected to *w* and *w* is connected to *v,* a cycle exists within *G*. Yet, by definition, *G* is acyclic, thus a contradiction is found and no vertex *v* is strongly connected to a vertex *w,* leaving only vertices connected to themselves.

Nathan Stouffer
CSCI 232, Homework 3

Exercise 4.3.3

Problem. Show that if a graph's edges all have distinct weights, the MST is unique.

Solution. Let *G* be a graph where are all edges have distinct weights and *M* be an arbitrary minimum spanning tree. To show that *M* is unique, we will use proof by contradiction. That is, we assume that *M* is not unique. If *M* is not unique, then there exists a spanning tree *N* with at least one edge *e* not in *M* where the weight of *N* is equal to the the weight of *M*. However, recall that each edge in a minimum spanning tree must be the edge with the smallest cost that connects two partitions of a graph. Since we know that *e* is not in *M*, *e* can only be such an edge if the weight of *e* is the same as the weight of an edge in *M.* This cannot be the case since each edge in *G* has a distinct weight, and thus the contradiction is found. Therefore, if a graph's edges all have distinct weights, the MST is unique.

Nathan Stouffer
CSCI 232, Homework 3

Exercise 4.3.7

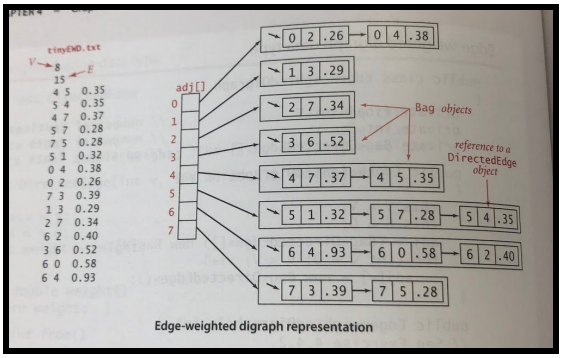
Problem. How would you find a maximum spanning tree of an edge-weighted graph?

Solution. To find a maximum spanning tree of an edge-weighted graph, I would modify input to Prim's algorithm. Prim's algorithm considers the existing tree and all edges that connect to vertices not in the tree, adding the edge to the minimum spanning tree that has the smallest cost. Instead of using the edge weight as the cost, use the edge weight multiplied by -1. Then run Prim's algorithm on the graph, this will then find the most negative path since the largest (or most positive) cost will be mapped to the smallest (or most negative) cost.

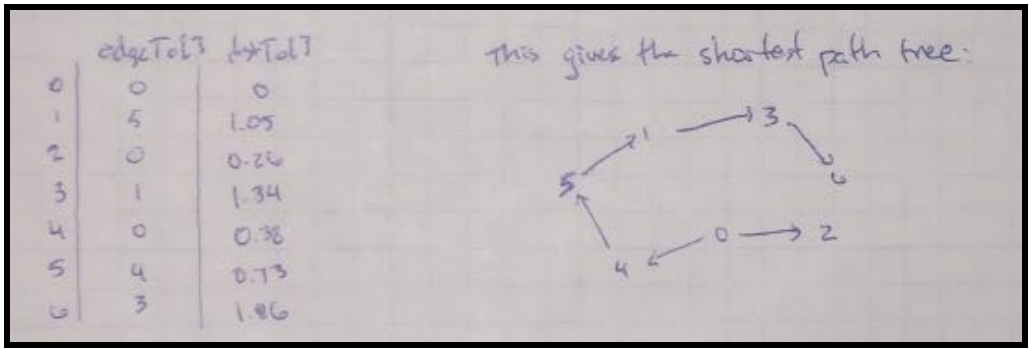
Nathan Stouffer
CSCI 232, Homework 3

Exercise 4.4.4

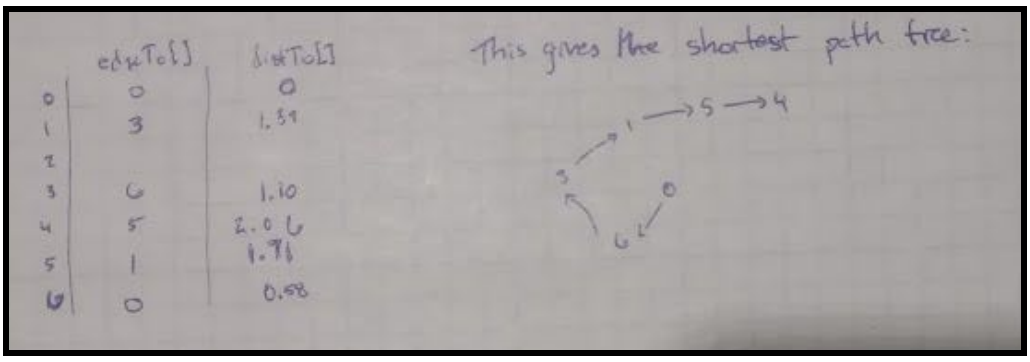
Problem. Draw the (unique) SPT for source 0 of the edge weighted digraph obtained by deleting vertex 7 from tinyEWD.txt, and give the parent-link representation of the SPT. Answer the question for the same graph with all the edges reversed.



Solution.
Edges consistent with text input:



With reversed edges:



Nathan Stouffer
CSCI 232, Homework 3

Creative Problem 2.5.30

Problem. *Boerner's theorem.* True or false: If you sort each column of a matrix, then sort each row, the columns are still sorted. Justify your answer.

Solution. Boerner's Theorem is true. Consider a valid, arbitrary matrix M where the columns are sorted. Now apply bubble sort to each row in M, iterating from the first row. In a single row, choose two consecutive values a and b in their respective columns c and c_1 without loss of generality. Then evaluate the pair using Bubble Sort. If Bubble Sort does not initiate a swap, then the matrix did not change and the columns must still be sorted. If Bubble Sort initiates a swap, then $b < a$ and we must check the corresponding values in the next row to see if each column in the matrix is still sorted. Consider a_1 and b_1 where a_1 follows a in column c and b_1 follows b in column c_1. There are then two cases.

Case 1: $a_1 \leq b_1$.
Since $a_1 \leq b_1$ and $b < a$, a swap will result in b preceding a_1 in c, which stays sorted since $b < a \leq a_1$. Additionally, c_1 stays sorted after the swap since $a \leq a_1 < b_1$.

Case 2: $a_1 > b_1$.
Since $a_1 > b_1$ and $b < a$, a swap will result in b preceding a_1 in c and a preceding b_1 in c_1. The two columns c and c_1 are not necessarily sorted. If not, the values a_1 and b_1 will be swapped when bubble sort is applied to the next row. This will have caused a to swap positions with b and a_1 to swap positions with b_1, in essence, only changing which column the values were in. Since the original matrix had valid sorted columns, this new matrix with both pairs of values swapped must also have sorted columns.

We now know that applying Bubble Sort to each row of M does not change the fact that the columns of M are sorted. Since Bubble Sort will return the same sorted row as any other sorting algorithm, it can be said that sorting the rows of M leaves the columns of M in a sorted state. Since M is an arbitrary matrix with sorted columns, the statement "If you sort each column of a matrix, then sort each row, the columns are still sorted." is true.

This is tested on square matrices of various sizes in the file HW3Stouffer.java, which gives the following output.

```
For a randomly populated 5x5 matrix, Boener's Theorem proved to be true.
For a randomly populated 10x10 matrix, Boener's Theorem proved to be true.
For a randomly populated 50x50 matrix, Boener's Theorem proved to be true.
For a randomly populated 100x100 matrix, Boener's Theorem proved to be true.
For a randomly populated 1000x1000 matrix, Boener's Theorem proved to be true.
BUILD SUCCESSFUL (total time: 0 seconds)
```