HW 5

Tuesday, April 27, 2021 2:58 PM

CSCE 221-200: Honors Data Structures and Algorithms
Assignment Cover Page
Spring 2021

Name:	Alexander Akomer
Email:	alexakomer@tamu.edu
Assignment:	HW 5
Grade (filled in by grader):	

Please list below all sources (people, books, webpages, etc.) consulted regarding this assignment (use the back if necessary):

CSCE 221 Students	Other People	Printed Material	Web Material (give URL)	Other Sources
1.	1.	1.	1. https://www.geeksforgeeks.org/merge-sort/	1.
2.	2.	2.	2.https://www.geeksforgeeks.org/topological-sorting/	2.
3.	3.	3.	3.	3.

Recall that TAMU Student Rules define academic misconduct to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. *Disciplinary actions range from grade penalty to expulsion.*

"On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work. In particular, I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received or given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment."

Signature:	Alexander Akomer
Date:	04/29/2021

Exercise 7.2:
What is the running time of insertion sort if all elements are equal?

Insertion sort will have a running time of O(n) if all elements are equal because, though n comparisons are made, no swaps are required. Therefore, the algorithm touches all n elements a single time and performs no more computations, leaving the time complexity at O(n).

Exercise 7.17(a):
Determine the running time of Mergesort for
a. sorted input

Mergesort has the same time complexity in its worst, average, and best case due to the fact that the algorithm always divides the array into two partitions and takes linear time to sort and merge each of the halves. Because of this algorithm that performs a linear operation to merge partitions (which is O(n) time) on log n partitions (due to the nature of recursively halving the array), the algorithm will always take an O(n log n) time. Therefore, a sorted input will still have a time complexity of O(n log n)

Exercise 7.33:
Prove that any algorithm that finds an element X in a sorted list of N elements requires Ω(logN) comparisons.

Because this is a sorted list of elements, simpler algorithms such as linear search can be discarded for more efficient algorithms such as binary search which can find an element X in Ω(log n) time. Clearly, this is a better time complexity than that of a linear search, which would be O(n). However, the reason that an element cannot be found faster than with log n comparisons is due to the nature of comparison-based algorithms. An element cannot be found without a comparison-based algorithm, and the nature of comparison-based algorithms requires that either all elements must be compared or simply the relevant elements must be compared (which is why a binary search allows for log n time). Binary search is widely known as the fastest method to find an element in a sorted list of n elements due to this key aspect of comparison-based algorithms.

Exercise 9.1:
Find a topological ordering for the graph in Figure 9.81.

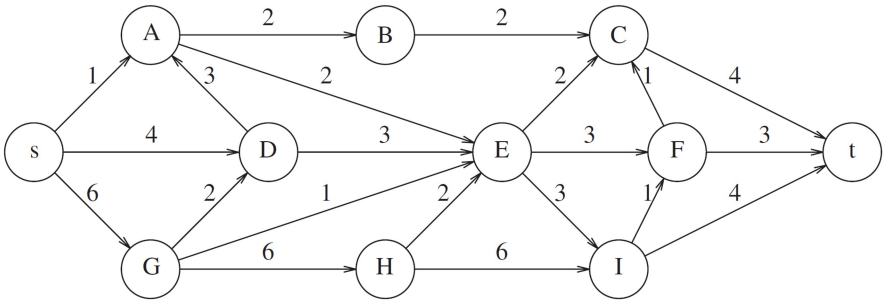


Figure 9.81 Graph used in Exercises 9.1 and 9.11

Because this is a directed acyclic graph (DAG), the topological sorting can be performed.
Also, topological sorting is not unique by nature, so several answers could be correct.
Finally, the topological ordering completely disregards the weights associated with each edge.

- The first vertex to be removed is s, as it is the only vertex with zero incoming edges.
- The next vertex to be removed is G, as it now has zero incoming edges.
- The next vertex to be removed is H, as it now has zero incoming edges.
- The next vertex to be removed is D, as it now has zero incoming edges.
- The next vertex to be removed is A, as it now has zero incoming edges.
- The next vertex to be removed is E, as it now has zero incoming edges.
- The next vertex to be removed is I, as it now has zero incoming edges.
- The next vertex to be removed is F, as it now has zero incoming edges.
- The next vertex to be removed is B, as it now has zero incoming edges.
- The next vertex to be removed is C, as it now has zero incoming edges.
- The final vertex to be removed is t, as it is the only remaining vertex.

Therefore, my topological ordering of the graph from Figure 9.81 is:

{s, G, H, D, A, E, I, F, B, C, t}

Exercise 9.5(b):
b. Find the shortest unweighted path from B to all other vertices for the graph in Figure 9.82.

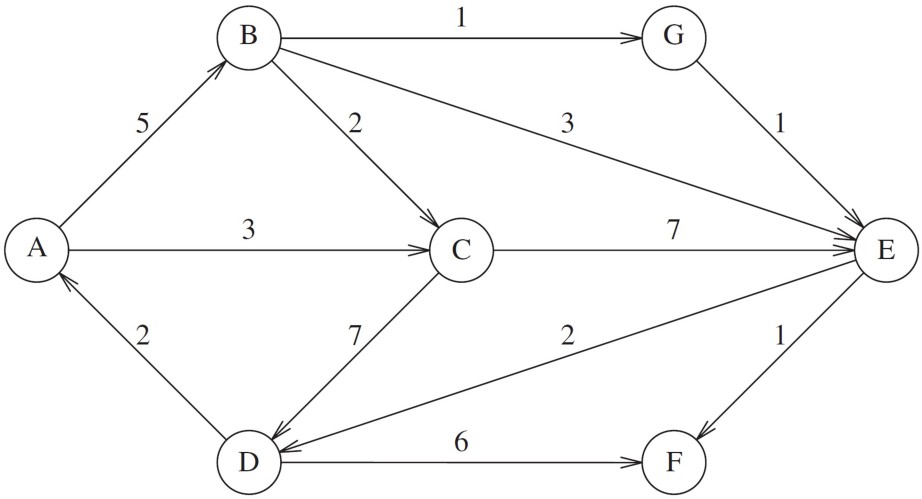


Figure 9.82 Graph used in Exercise 9.5

This exercise is looking to find the shortest path from B to all other vertices while completely ignoring weights; in this case, each edge acts as if it has a weight of one when calculating the length of the path.

This uses the breadth first search (BFS with source B).

B -> A: 3 (B -> C -> D -> A)
B -> C: 1 (B -> C)
B -> D: 2 (B -> C -> D) or (B -> E -> D)
B -> E: 1 (B -> E)
B -> F: 2 (B -> E -> F)
B -> G: 1 (B -> G)

Exercise 9.25:
Give an algorithm to decide whether an edge (v, w) in a depth-first spanning forest of a directed graph is a tree, back, cross, or forward edge.

After a directed graph is transformed into a depth-first spanning forest using DFS,

Pseudocode:

```
// function determines the lower vertex and then iterates up from it to see if other vertex exists along the same path
bool shareCommonAncestor(Vertex v, Vertex w)
{
	if (depth(v) <= depth(w))
	{
		while(v.parent() != null || v.parent() != w)
			v = v.parent();
		if (v == w)
```

```
                return true;
        else
                return false;
    }
    else
    {
        while(w.parent() != null || w.parent() != v)
                w = w.parent();
        if (v == w)
                return true;
        else
                return false;
    }
}

// finds the depth of the vertex
int depth(Vertex vert)
{
    Int counter = 0;
    while (vert.parent() != null) //parent() is a typical parent method that points up the tree
    {
        counter++;
        vert = vert.parent();
    }
    return counter;
}

// determines what type of edge is provided given the two vertices
string typeEdge(Vertex v, Vertex w) // assuming all provided edges are real
{
    If (v.child() == w || w.child() == v) // child() is a typical child method that points down the tree
        return "tree";
    If (shareCommonAncestor(v, w)) // method defined previously
        {
            if(depth(v) <= depth(w)) // method defined previously
                return "forward";
            else
                Return "back";
        }
    else
        return "cross";
}
```

Exercise 9.5(a):
a. Find the shortest path from A to all other vertices for the graph in Figure 9.82.

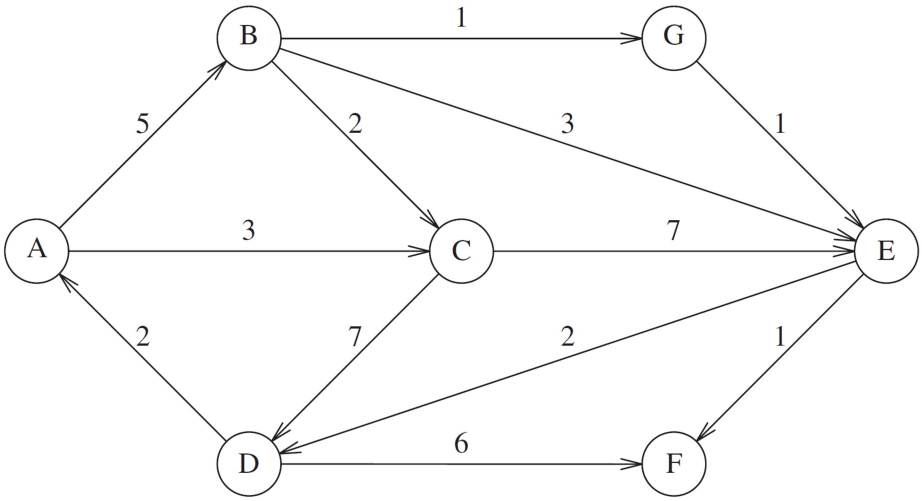


Figure 9.82 Graph used in Exercise 9.5

This exercise is looking to find the shortest path from A to all other vertices while accounting for weights; in this case, each edge retains the given weight value when calculating the length of the path.

This uses Dijkstra's SSSP Algorithm (with source A).

A -> B: 5 (A -> B)
A -> C: 3 (A -> C)
A -> D: 9 (A -> B -> G -> E -> D)
A -> E: 7 (A -> B -> G -> E)
A -> F: 8 (A -> B -> G -> E -> F)
A -> G: 6 (A -> B -> G)

Exercise 9.15(a):
a. Find a minimum spanning tree for the graph in Figure 9.84 using both Prim's and Kruskal's algorithms.

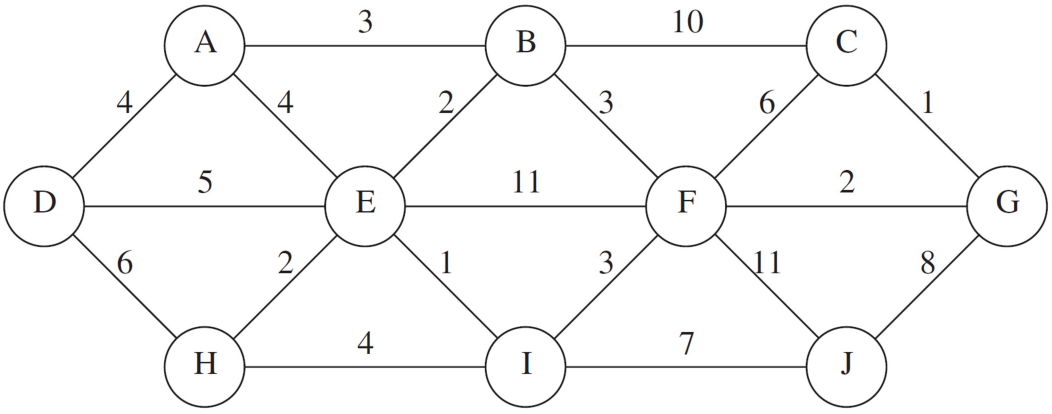
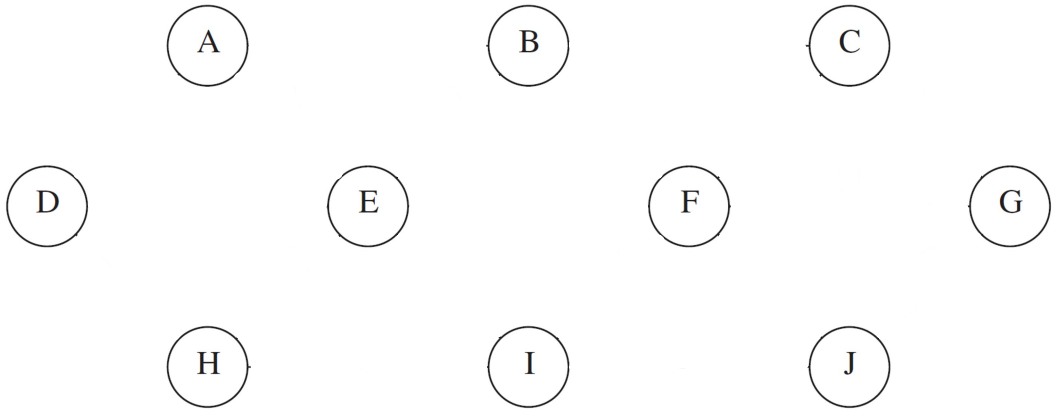


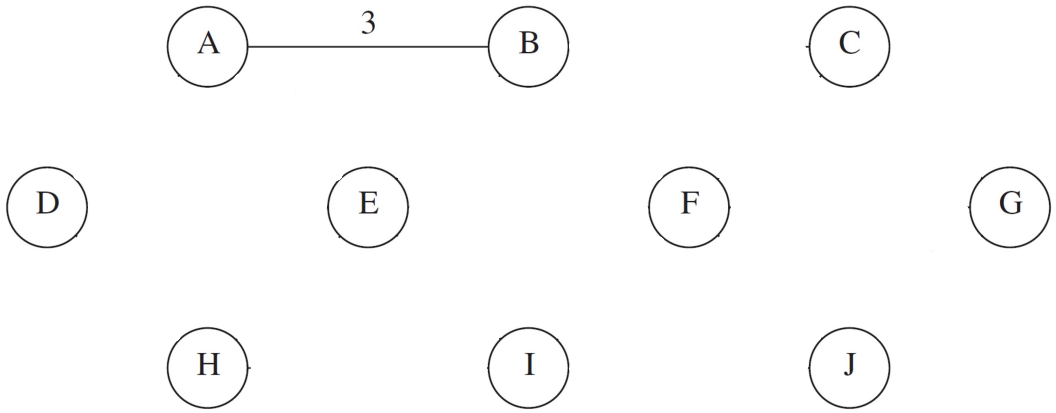
Figure 9.84 Graph used in Exercise 9.15

Prim's algorithm:

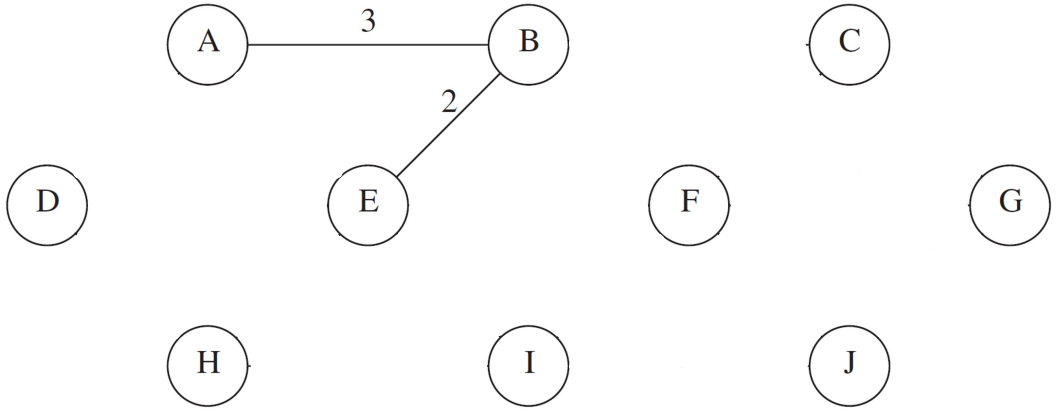
Prim's algorithm:



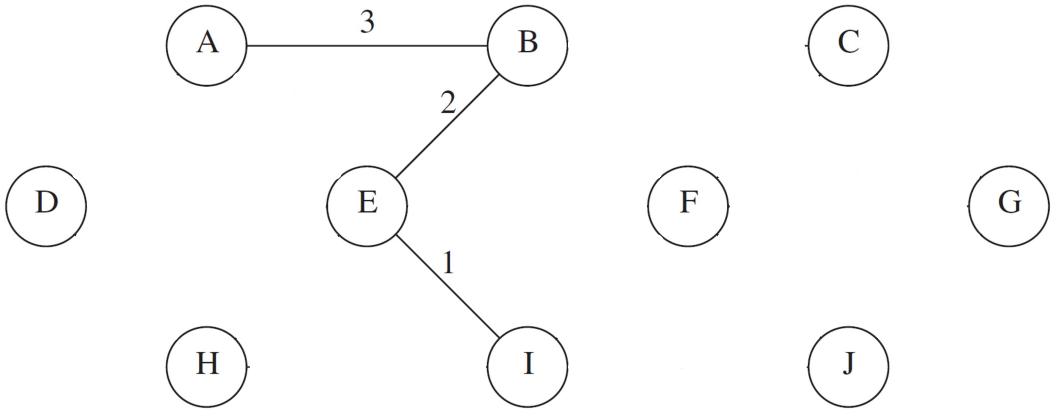
Node A is chosen randomly and the lowest weight is chosen from its edges (3).



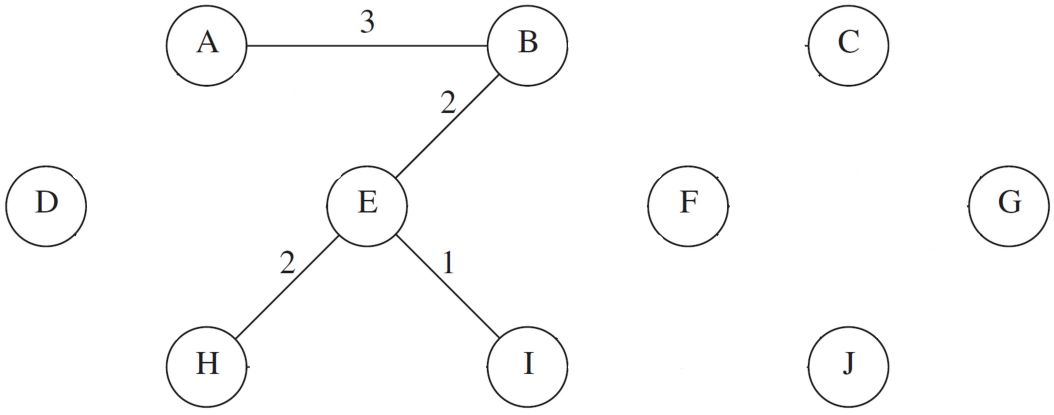
Node E is then connected due to having the next lowest weighted edge (2).



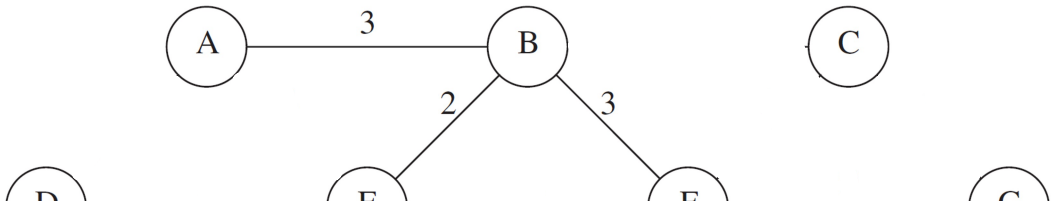
Node I is then connected due to having the next lowest weighted edge (1).

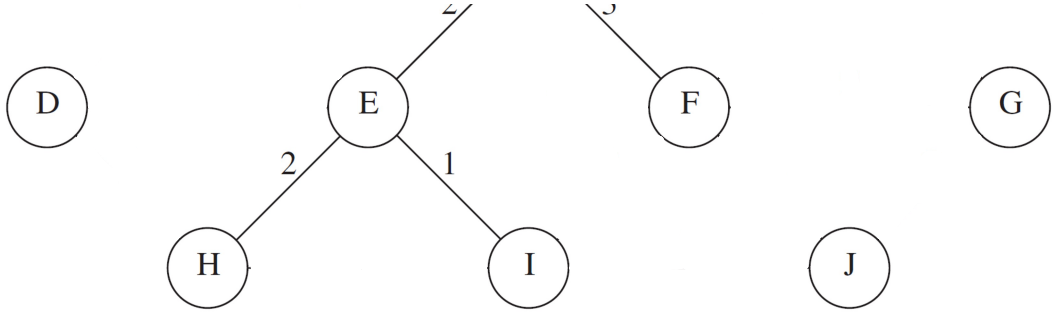


Node H is then connected due to having the next lowest weighted edge (2).

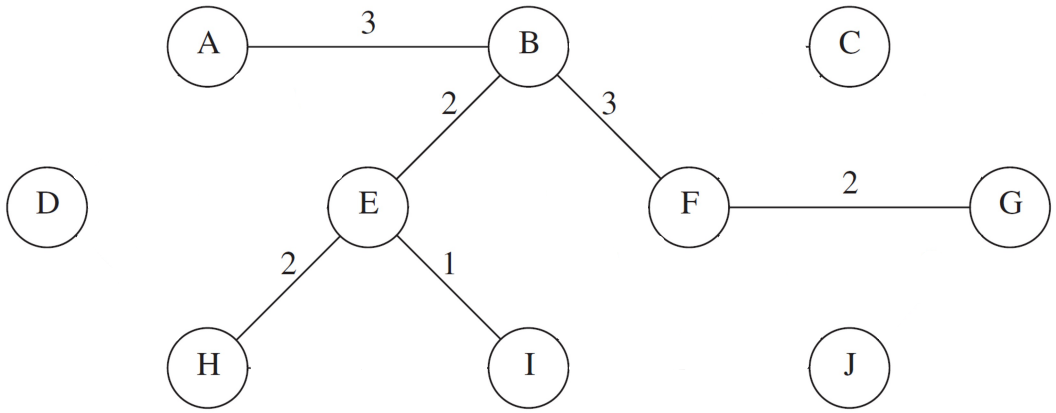


Node F is then connected due to having the next lowest weighted edge (3).

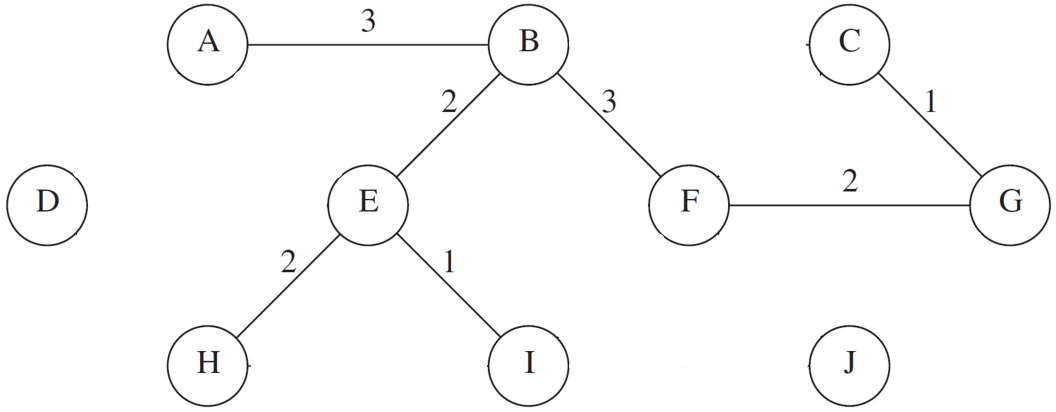




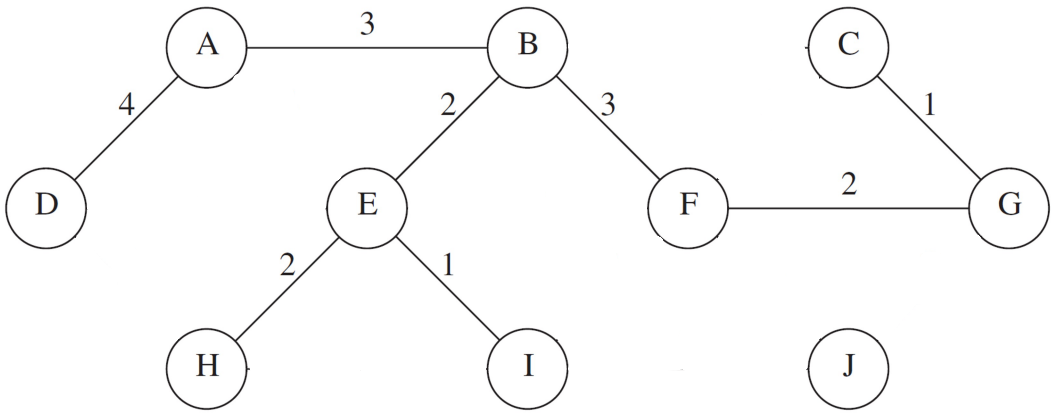
Node G is then connected due to having the next lowest weighted edge (2).



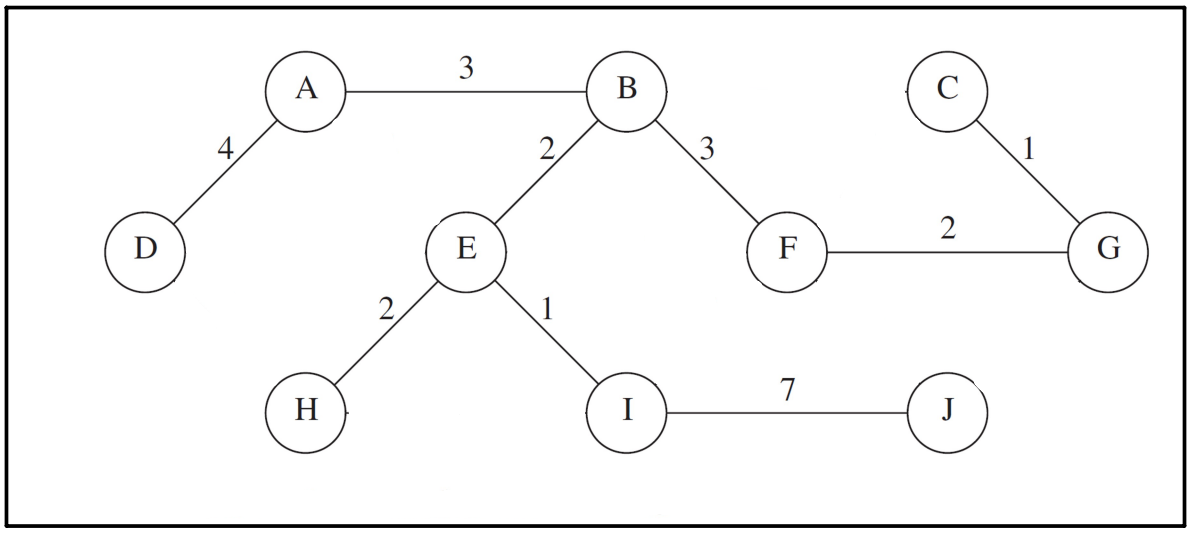
Node C is then connected due to having the next lowest weighted edge (1).



Node D is then connected due to having the next lowest weighted edge (4).



Finally, Node J is connected from its lowest weighted edge (7).



Kruskal's Algorithm:

All vertex pairs are placed into a forest, so that each edge and its respective weight are represented.

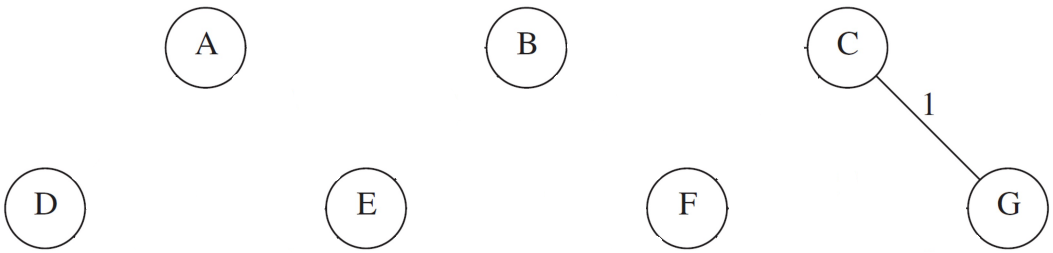
AB - 3
AD - 4
AE - 4
BC - 10
BF - 3
BE - 2
CG - 1
CF - 6
DE - 5
DH - 6
EH - 2
EI - 1

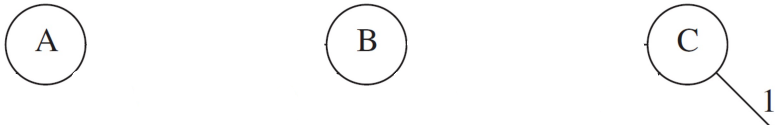
EF - 11
FG - 2
FJ - 11
FI - 3
GJ - 8
HI - 4
IJ - 7

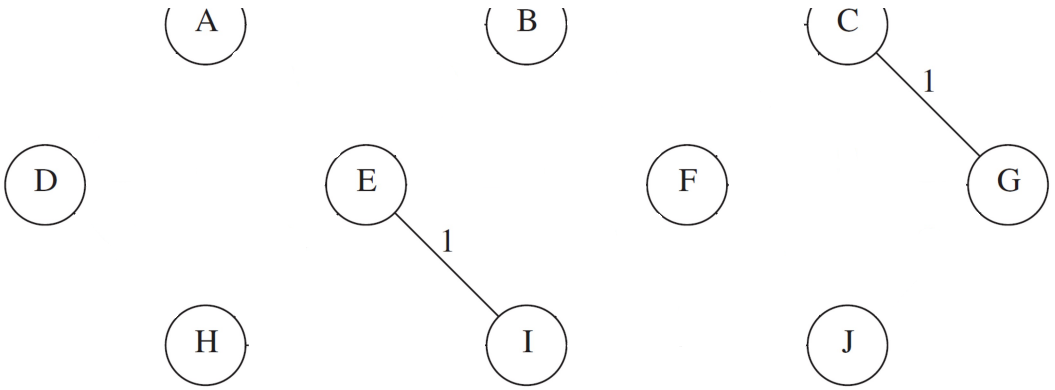
Next, the forest is sorted so that the edges are in increasing order (w.r.t. weight).

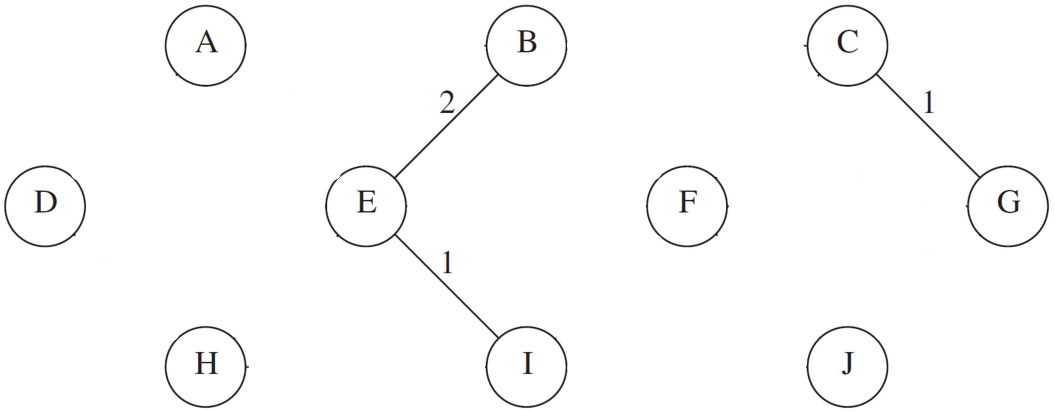
CG - 1
EI - 1
EH - 2
BE - 2
FG - 2
AB - 3
BF - 3
FI - 3
AD - 4
AE - 4
HI - 4
DE - 5
CF - 6
DH - 6
IJ - 7
GJ - 8
BC - 10
EF - 11
FJ - 11

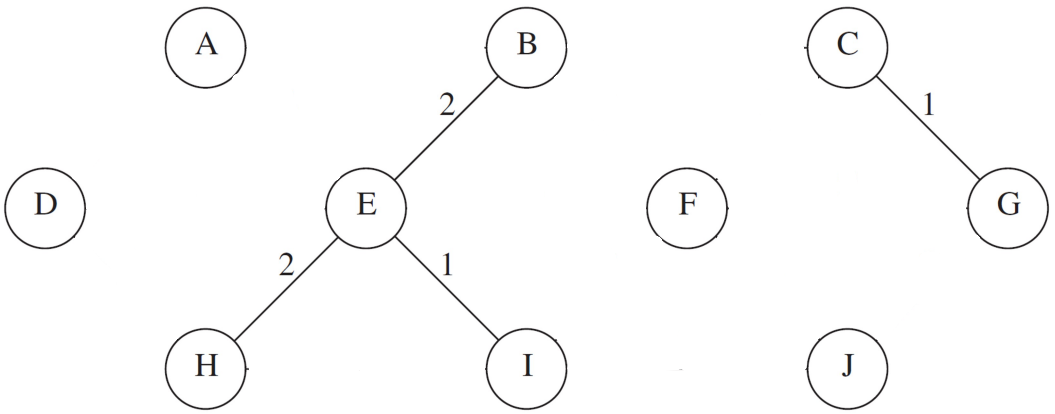
Finally, connections are made to vertices that disconnected starting from the top of this sorted list, rejecting vertices that have already been connected or edges that would create a cycle. This process is repeated until every vertex is connected without cycles forming.

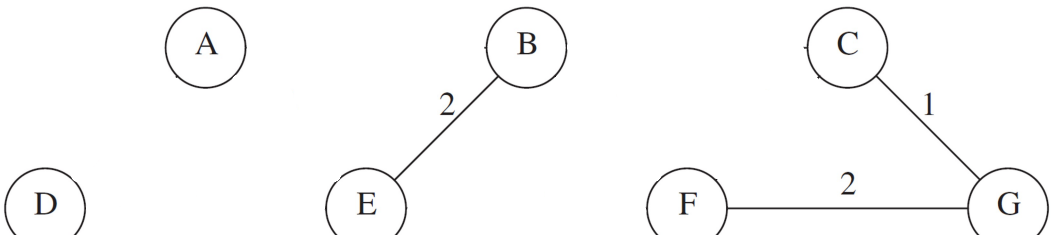


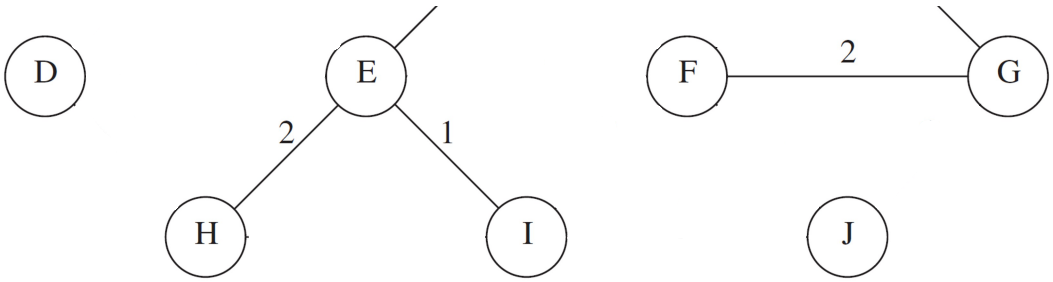


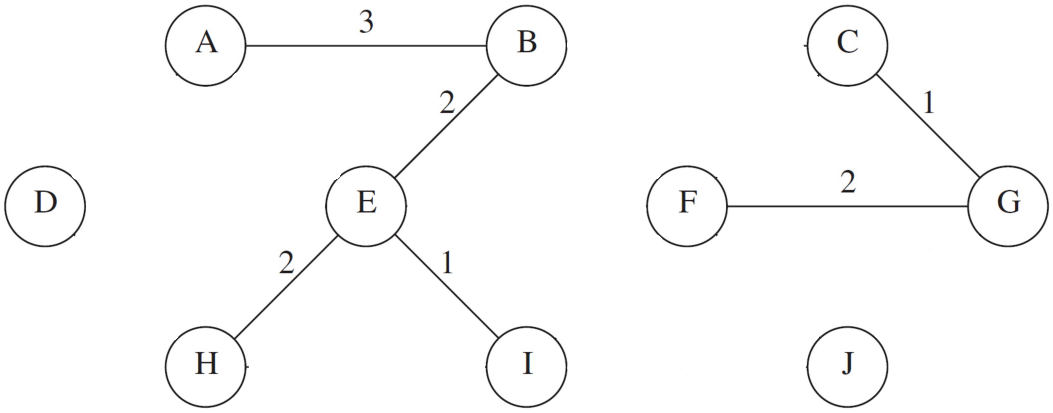

A
B
C
1
D
E
F
G
1
H
I
J

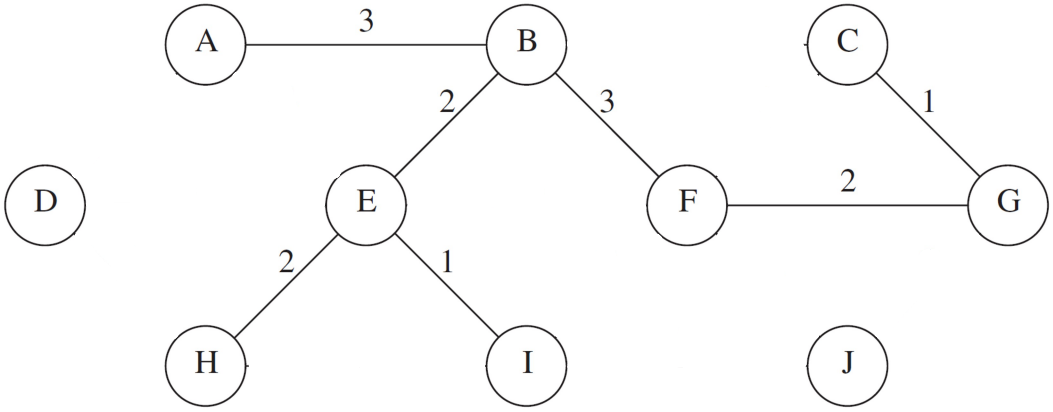

A
B
C
2
1
D
E
F
G
1
H
I
J

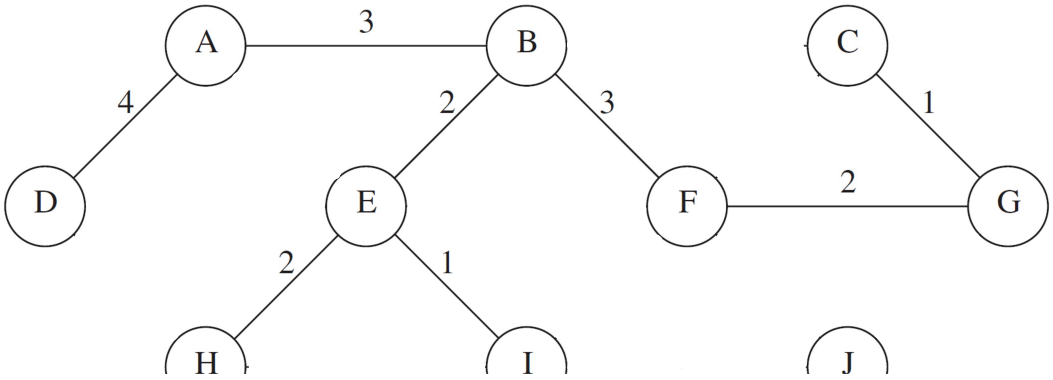

A
B
C
2
1
D
E
F
G
2
1
H
I
J

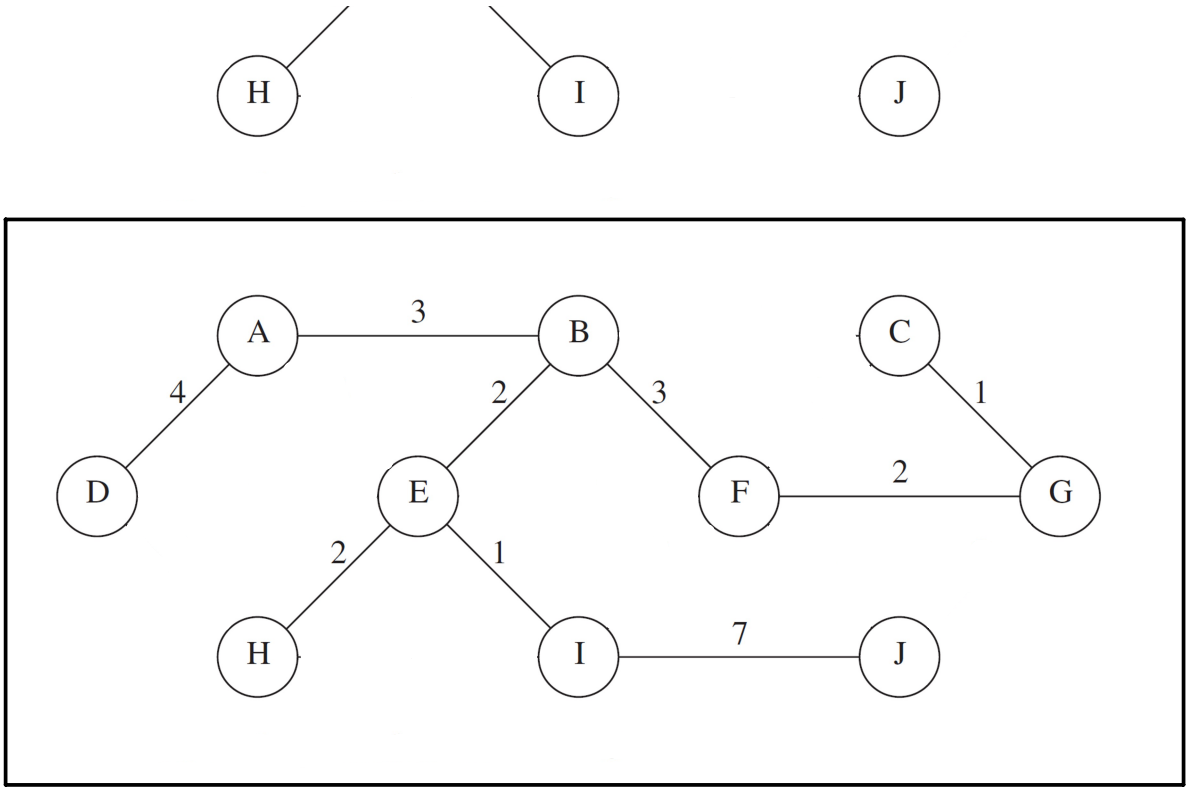

A
B
C
2
1
D
E
F
2
G


D
E
2
1
H
I
F
2
G
J


A
3
B
2
D
E
2
1
H
I
C
1
F
2
G
J


A
3
B
2
3
D
E
F
2
G
2
1
H
I
C
1
J


A
3
B
4
2
3
D
E
F
2
G
2
1
H
I
C
1
J



Exercise 9.39:
A graph is k-colorable if each vertex can be given one of k colors, and no edge connects identically colored vertices. Give a linear-time algorithm to test a graph for two-colorability. Assume graphs are stored in adjacency-list format; you must specify any additional data structures that are needed.

Pseudocode:

```
// this function assumes that each node has a color attribute
// and that each vertex has an associated with an adjacency matrix containing its neighbors
bool twoColorable(Vertex source)
{
        for each node v in the graph
        {
                color(v) = NULL;
        }
        color(s) = BLUE;

        // let Q be an initially empty queue
        Q.enqueue(s)

        while (Q is not empty) do
        {
                v := Q.dequeue()
                for each vertex w adjacent to v do
                {
                        If(color(w) == NULL) //w is not yet visited
                        {
                                if(color(v) == RED)
```

```
                        color(w) = BLUE;
                    else
                        color(w) = RED;
                    Q.enqueue(w);
                }
                else if (color(w) == color(v))
                    return false;
            }
        }
        return true;
}
```