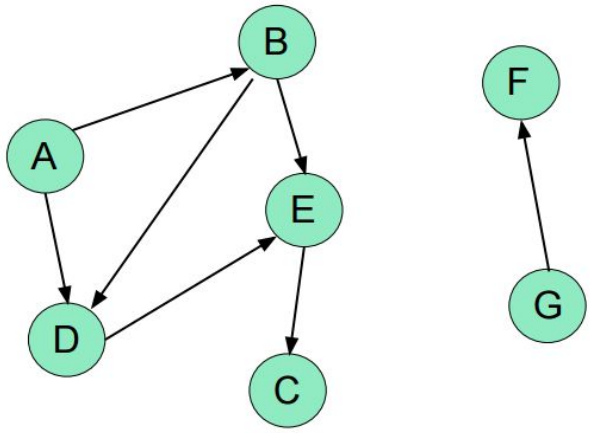
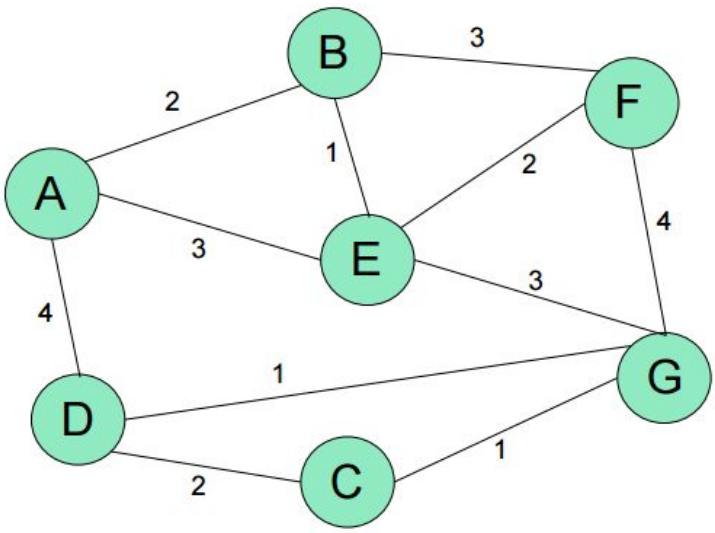
1. Topological Sort: Determine a topological ordering for the following directed acyclic graph (DAG). Show your work by computing the in-degree value of each vertex and keeping track of the vertices with in-degree of 0 in a queue.



Node	A	B	C	D	E	F	G
Removed?	X	X	X	X	X	X	X
In-Degree	0	~~1~~ 0	~~1~~ 0	~~2~~ ~~1~~ 0	~~2~~ ~~1~~ 0	~~1~~ 0	0

Result: A B D E C G F

2. **Minimum Spanning Trees:** Use the following graph for both of the following minimum spanning tree algorithms.



a. Build a minimum spanning tree for the graph using Kruskal's algorithm. Number each edge according to when it is entered into the minimum spanning tree. The first edge will get number 1, etc. Show the ordering of the edges you consider by showing the state of the pending set of edges to consider.

Edges in Sorted Order: 1: (B, E), (D, G), (C, G)
2: (A, B), (E, F), (C, D)
3: (B, F), (A, E), (E, G)
4: (A, D), (F, G)

Sets: (B, E), (D, G), (C, G) (A, B), (E, F), (C, D) (B, F), (A, E), (E, G) (A, D), (F, G)

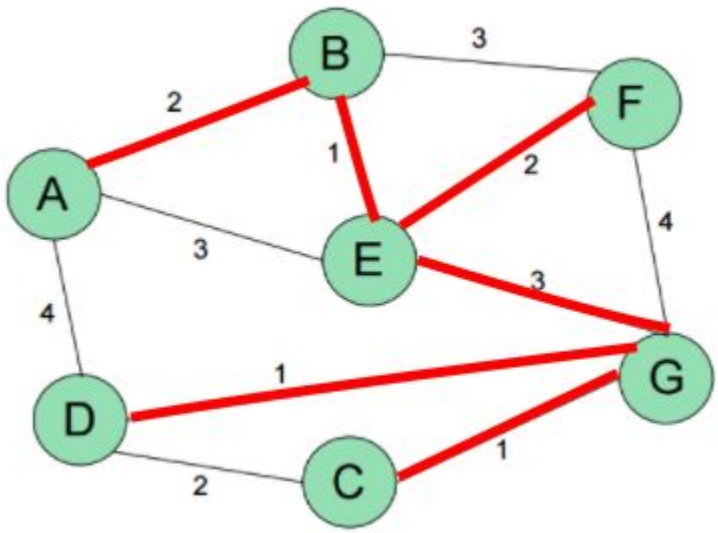
(B, E) (C, D, G) (A, B), (E, F), (C, D) (B, F), (A, E), (E, G) (A, D), (F, G)

(A, B, E) (C, D, G) (E, F) (B, F) (A, E), (E, G) (A, D), (F, G)

(A, B, E, F) (C, D, G) (A, E) (E, G) (A, D) (F, G)

(A, B, C, D, E, F, G)

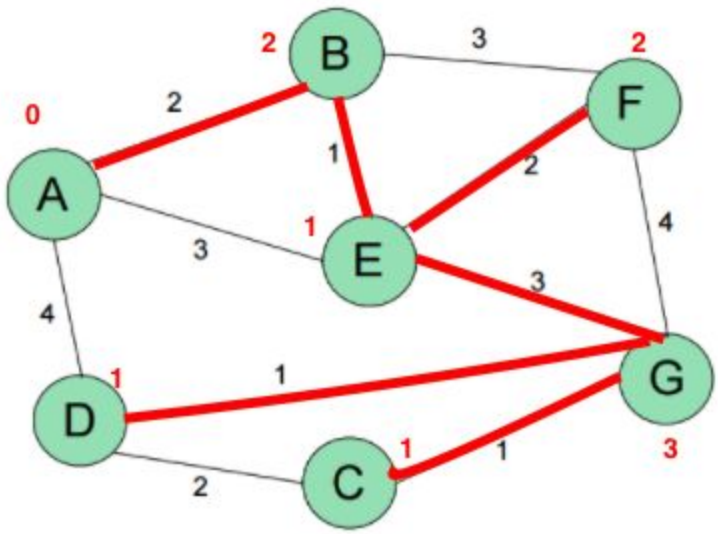
Output: (B, E) (D, G) (C, G) (A, B) (E, F) (E, G)



b. Now build a minimum spanning tree for the graph using Prim's algorithm, starting with vertex A. Again, number each edge according to when it is entered into the set of edges. The first edge will get number 1, etc. Show the partial state in a table keeping track of known, cost, and path similar to the lecture slides.

Node	Known?	Cost	Previous Node
A	Y	~~∞~~ 0	
B	Y	~~∞~~ 2	A
C	Y	~~∞~~ 1	G
D	D	~~∞~~ ~~4~~ 1	~~A~~ G
E	Y	~~∞~~ ~~3~~ 1	~~A~~ B
F	Y	~~∞~~ ~~3~~ 2	~~B~~ E
G	Y	~~∞~~ 3	E

Output: (A, B) (B, E) (E, F) (E, G) (G, C) (G, D)



3. Graph Pseudocode:

For the following problems, assume you have a directed unweighted graph stored in a structure:Map<Vertex, Set<Vertex>> graph; Where the keys of the outer map represent the source vertices in the graph. The values of the map represent the adjacent neighbors on the out edges for each vertex. For example, the graph above pictured for #1 in the topological sort problem might be represented in the graph structure (with no particular ordering) as:

```
graph -> [
        A -> [B, D],
        B -> [E, D],
        C -> [],
        D -> [E],
        E -> [C],
        F -> [],
        G -> [F]
      ]
```

a. Write the pseudocode for a method reverse(Map<Vertex, Set<Vertex>> graph) which returns a new Map where each edge (u, v) has the reverse direction (v, u).

```
function reverse {
        create new Map<Vertex, Set<Vertex>> rev.
        for each vertex source in graph's keySet{
                for each vertex destination in the set returned by graph.get(source) {
                        if rev.keySet does not contain destination{
                                add destination to graph.keySet
                        }
                        add source to graph.get(destination);
                }
        }
        return rev;
 }
```

b. Write the pseudocode for a method nextAdjacentVertices(Map<Vertex, Set<Vertex>> graph, Vertex source) that returns a Set<Vertex> of vertices representing all of the vertices that are path length 2 away from the source vertex (all the vertices that are adjacent to the neighbors of the source).

```
function nextAdjacentVertices(Map<Vertex, Set<Vertex>> graph, Vertex source) {
        create new Set<Vertex> nextAdj;
        for each source's neighbor n (n in the set returned by graph.get(source)){
                for each n's neighbour nn (nn in the set returned by graph.get(n)) {
                        add nn to the nextAdj.
                }
        }
        return nextAdj;
}
```

c. Assuming the Map and Set are hash structures, what is the worst case asymptotic tight bound runtime for the pseudocode you've written in parts (a) and (b).

For (a): Ω(|V| + |E|)

For (b): Ω(|V|^2)

4. Dijkstra's and Negative Edges:

a. If there is more than one minimum cost path from v to w, will Dijkstra's algorithm always find the path with the fewest edges? If not, explain in a few sentences how to modify Dijkstra's algorithm so that if there is more than one minimum path from v to w, a path with the fewest edges is chosen. Assume no negative weight edges or negative weight cycles.

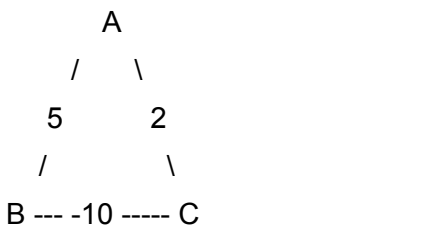
Dijkstra's algorithm does not find the edge with the fewest edges. It only finds the first 'shortest' one.

In order for Dijkstra's method to find the shortedPath with the fewest edges, we can use an integer array numEdges[] to keep track of the number of the smallest number of edges it takes from the source Vertex to the current one.

For example, let *v* be a node that is known and *w* is its adjacent node.

```
if (distance to v  + cost(v, w) = distance to w){
        if (numEdges[v] + 1 < numEdges[w]){
                set w.path to v
                set numEdges[w] = numEdges[v] + 1
        }
} else if (distance to v  + cost(v, w) < distance to w) {
        set w.path to v
        numEdges[w] = numEdges[v] + 1
}
```

b. Give an example where Dijkstra's algorithm gives the wrong answer in the presence of a negative cost edge but no negative-cost cycles. Explain briefly why Dijkstra's algorithm fails on your example. The example need not be complex; it is possible to demonstrate the point using as few as 3 vertices.



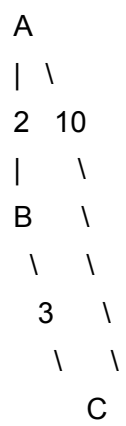
Vertices: {A, B, C}, edges:{(A,C,2), (A,B,5), (B,C,-10)}

In this case, when using Dijkstra's algorithm to find shortest path between A and C, it will find the path from A to C, which has a lower cost than A to B, and fail to find A - B - C (-5).

c. Suppose you are given a graph that has negative-cost edges but no negative-cost cycles. Consider the following strategy to find shortest paths in this graph:

Uniformly add a constant *k* to the cost of every edge, so that all costs become non-negative, then run Dijkstra's algorithm and return that result with the edge costs reverted back to their original values (i.e., with *k* subtracted).

a. Give an example where this technique fails (Dijkstra's would not find what is actually the shortest path) and explain why it fails.



In this case, assume we add a big number k to the cost of each edges. Then the cost from A-C is 10+k, where the cost from A-B-C is 5+2k. The shortest path returned by this algorithm will be A-C with a cost of 10 since 10+k < 5+2k, while the true shortest path is actually A-B-C with a cost of 5.

b. Give a general explanation as to why this technique does not work. Think about your example and why the original least cost path is no longer the least cost path after adding *k*.

Adding k to the cost of each node will make the distance dependent on the number of edges in the path. A path with fewer edges will always be chosen as the shortest one when k is significantly larger than the costs.

Project:

Describe how you tested your shortestPath method.

I completed the findPath.java file and called shortestPath to look for the shortest path between several vertices. I specifically tried the same source/destination pair, the no edge exit pair and other pairs with multiple paths between them.