Problem #1 **Name:** Alexander Michelbrink

Pseudocode:

```
Algorithm RTTnetworks(G)
      Input graph G
      Output sequence A of sequences each containing all visitable
             stations within 4 links of a station.
      A ← new empty sequence of sequences
      for all u in G.vertices()
            for all v in G.vertices()
                  setLabel(v,UNEXPLORED)
            for all e in G.edges()
                  setLabel(e,UNEXPLORED)
            A.insertLast(RTTnetworks(u,G))
      return A

Algorithm RTTnetworks(s,G)
      L0 ← new empty sequence
      L0.insertLast(s)
      R.insertLast(s)
      setLabel(s,VISITED)
      i ← 0
      while (!Li.isEmpty() AND i < 4)
            Li+1 ← new empty sequence
            for all v in Li.elements()
                  for all e in G.incidentEdges(v)
                        if (getLabel(e) = UNEXPLORED)
                              w ← opposite(v,e)
                              if (getLabel(w) = UNEXPLORED)
                                    setLabel(e,DISCOVERY)
                                    setLabel(w,VISITED)
                                    Li+1.insertLast(w)
                                    R.insertLast(w)
                              else
                                    setLabel(e,CROSS)
            i ← i + 1
      return R
```

For this problem, I decided to use a variation of BFS when designing the algorithm.

The result will be a sequence A of n sequences. Each sequence in A contains the set of stations visitable (within four connections) by each of the n stations. The first element of each sequence indicates the starting station of that set.

My modifications to the algorithm included utilizing a new sequence R. Whenever a new vertex is discovered, the vertex is added to R. I also added a variable i to track

how many edges away from the starting vertex v each newly discovered vertex was. Once i reaches 4, the while-loop ends and R is returned. R will contain all stations visitable within 4 connections of s.

I also modified the first calling function to re-label all edges and vertices as unexplored whenever RTTnetworks(s,G) was called for the next vertex u.

The runtime of this algorithm is O(n(n+m)).

In the worst-case, the graph is made up of stations that can all reach each other in four connections or less. In this situation, we essentially carry out the BFS algorithm n times (once per station/vertex). Since my modifications of BFS only add a few lines that operate in O(1) time, we can simply multiply BFS's runtime of O(n+m) by n. Therefore, the worst-case runtime of my algorithm is O(n(n+m)).

Problem #2

Assumptions: The following algorithm assumes that G is a connected graph.

Pseudocode

```
Algorithm findEulerianCircuit(G)
      for all v in G.vertices()
            if (inDegree(v) != outDegree(v))
                  throw noEulerianCycleException
      for all e in G.edges()
            setLabel(e, UNEXPLORED)
      v ← G.aVertex()
      S ← new empty stack
      findEulerianCircuit(G,v,S)
      S.pop()
      return S

Algorithm findEulerianCircuit(G,v,S)
      S.push(v)
      for all e in outIncidentEdges(v)
            if (getLabel(e) = UNEXPLORED)
                  w ← destination(e)
                  setLabel(e, DISCOVERY)
                  S.push(e)
                  S ← findEulerianCircuit(G,w,S)
      return S
```

The above algorithm works by first testing if the graph G has a Eulerian Cycle. Since a directed graph has a Eulerian cycle if and only if each vertex has an equal in-degree and out-degree, the algorithm throws an exception if this is not true for any vertex in G. This is done by the first for-all loop and executes in O(n) time.

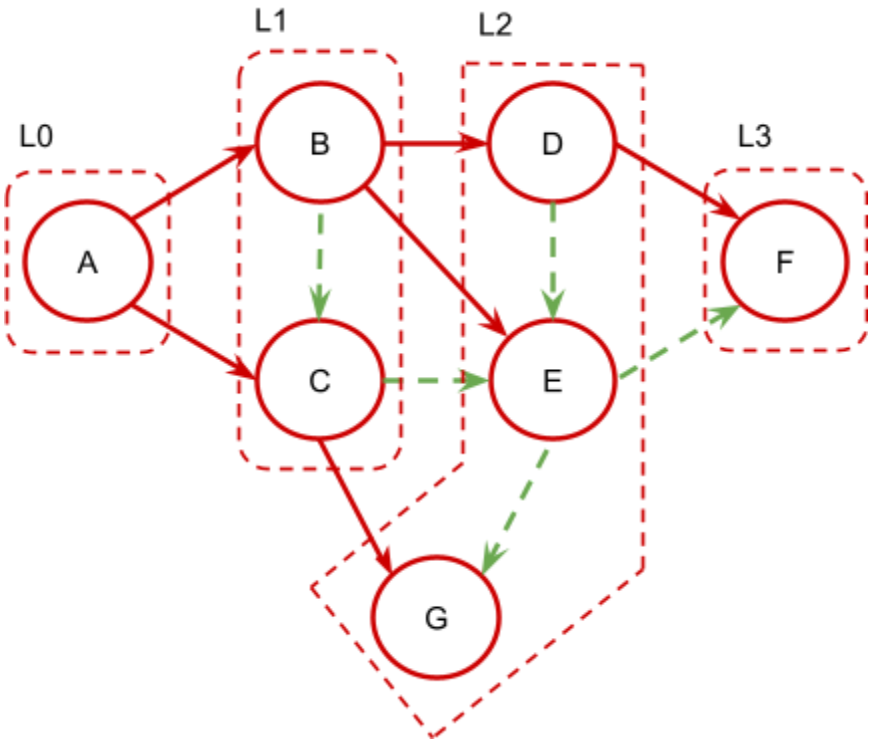
The algorithm then labels all edges to be unexplored in the second for-all loop. This is done in O(m) time. Next, an arbitrary vertex is selected, stack S is declared, and the next part of the algorithm is called. These actions are done in O(1) time.

The algorithm then pushes the current vertex v onto the stack and loops through all edges directed out of v. Assuming an adjacency list structure is used for the graph, outIncidentEdges() runs in O(deg(v)) time. From here, an unexplored edge is marked as explored before findEulerianCircuit(G,v,S) recursively calls itself. These recursive calls repeat until no more undiscovered edges exist (meaning the cycle is complete). This pattern works since the out-degree equals the in-degree for each vertex. Since a recursive call is done once per edge, this section of the algorithm runs in O(m) time.

Finally, the stack has its last element (the first step of the cycle repeated) popped off before it is returned by findEulerianCircuit(G). S will hold a possible eulerian circuit within the graph. Adding together all the elements of the algorithm considered above, we can see that the algorithm runs in O(n+m) time.

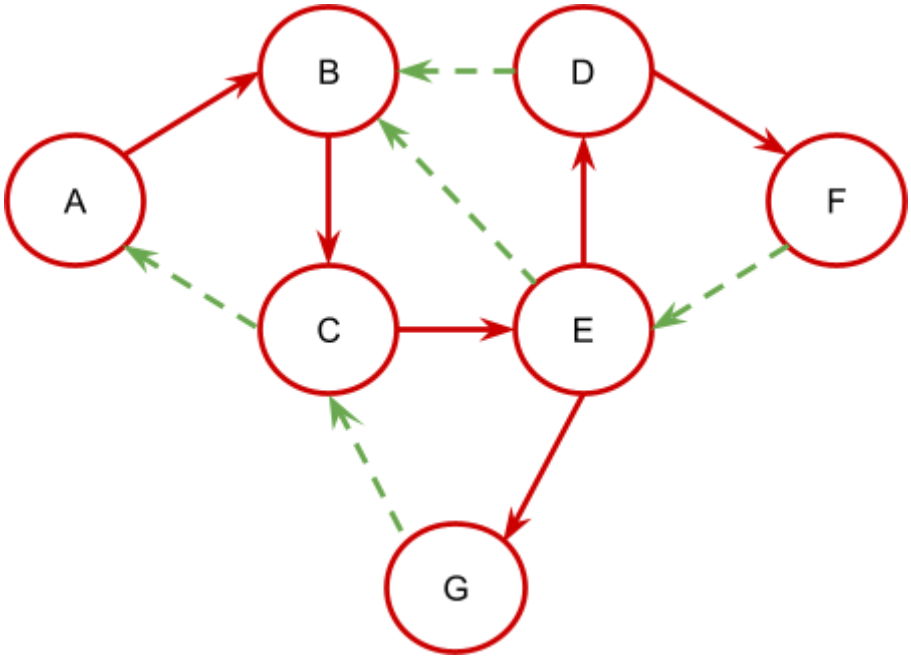
Problem #3

A.)



Red arrows are discovery edges and green arrows are cross edges.

B.)



Red arrows are discovery edges and green arrows are back edges.

Problem #4

A.)_____

A spanning tree of G can be found using DFS and only discovery-edges.

After DFS has been conducted, we can traverse through the graph using only the discovery edges. Starting at any vertex, we can add it to the sequence then add all discovery edges incident upon it to the sequence as well. We can repeat this exact pattern for any vertex at the opposite ends of any discovery edges found. Once there are no more discovery edges (no vertices have any more to add to the sequence), our sequence will contain a spanning tree of G. This is possible since the discovery edges labeled by DFS form a spanning tree of the connected component of v.

B)_____

Determining if G is acyclic can be done using DFS and only back-edges.

After DFS has been conducted, we can check all edges incident on each vertex in G. If any of these edges are back-edges, we can return false to indicate that G is not acyclic. If all vertices have had the edges incident on them checked and no back-edges have been found, we can return true to indicate that G is acyclic. This is possible since, if a graph has a back-edge, the graph has a cycle.

C.)_____

A path from vertex u to vertex v can be found using DFS and only discovery edges.

After conducting DFS, we first start from vertex u. We then follow a path of discovery-edges and the vertices at the opposite ends of them, adding each to a stack as we go. If a vertex has more than one discovery edge, we choose one and take note of which one we followed. If we reach vertex v, we can simply return the stack. If we reach a vertex that has no discovery edges, we can travel backwards along our path (popping off vertices and edges off the stack as needed) until we reach a vertex that had an alternate discovery edge to traverse. We continue this process of popping off needed vertices and edges and traveling down the alternate discovery-edge paths until v is found. In the end, S will have a path from u to v.

D.)_____

Shortest path from vertex u to vertex v can be found via BFS and only discovery edges.

After conducting BFS, we first start from vertex u. We then follow a path of discovery-edges and the vertices at the opposite ends of them, adding each to a stack as we go. Each time we add a new edge to the stack, we can increment a variable “d” that keeps track of how many edges away from u we are. If we reach vertex v, we can simply return the stack and variable d. If we reach a vertex that has no discovery edges, we can travel backwards along our path (popping off vertices and edges off the stack as needed and decrementing d whenever we pop off an edge) until we reach a vertex that had an alternate discovery edge to traverse. We continue this process of popping off needed vertices/edges, decrementing d, and traveling down the alternate discovery-edge paths until v is found. In the end, S will hold the shortest path from u to v and d will hold the length of this path.

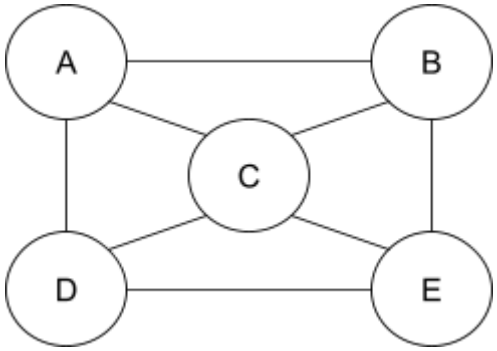
E.)_____

All connected components of G can be found using DFS and only discovery edges.

After conducting DFS, we can start from any vertex v. We can then add it to a sequence then add all discovery edges incident upon it to the sequence as well. We can repeat this exact pattern for any vertex at the opposite ends of any discovery edges found. While doing this, we will need to mark the vertices we add to our sequence as visited. Once there are no more new discovery edges left to follow, our sequence will hold one connected component of the graph. We can then repeat the above process for a vertex in the graph that has not yet been marked as visited. The next sequence we get will be the other connected component that contains that previously unvisited vertex. We do this until all vertices are marked as visited, resulting in separate sequences each containing a different connected component of the graph. This is possible since DFS (starting at any v) visits all the vertices and edges in the connected component of v.

Problem #5

A.)_____



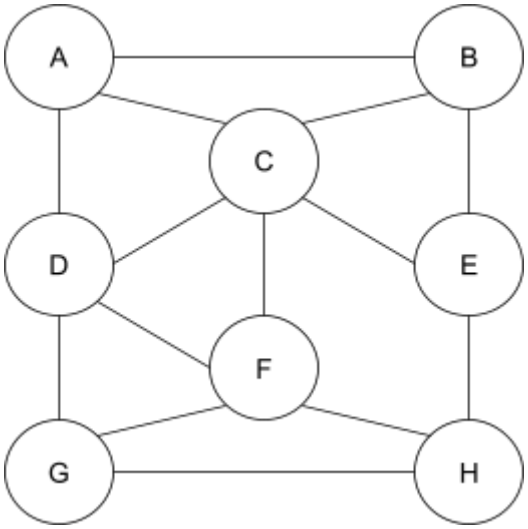
B.)_____

A triconnected graph with exactly 5 vertices and 6 edges is not possible.

This is because, for a graph to be triconnected, there must be at least three disjoint, simple paths between any two vertices u and v. For this to be possible, each vertex must be at least degree 3. If any vertex u had a degree less than 3, it would be impossible for it to have 3 disjoint paths to another vertex u. This is because at least one of the edges incident on v would have to be used two or more times when creating three paths to another vertex. Therefore, the three paths would not all be disjoint.

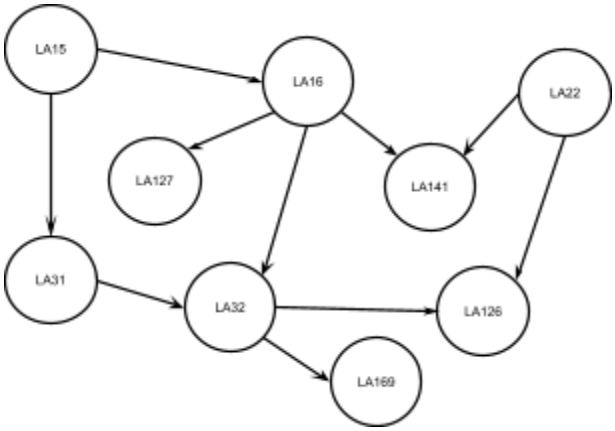
An edge is always incident on one (in the case of self-loops) or two vertices. So, if there are 6 edges, there is a maximum of 12 incidents. For each of the 5 vertices to be at least degree 3 (have 3 different edges incident on them), there would have to be at least 15 incidents. This is not possible with only 6 edges.

C.)_____

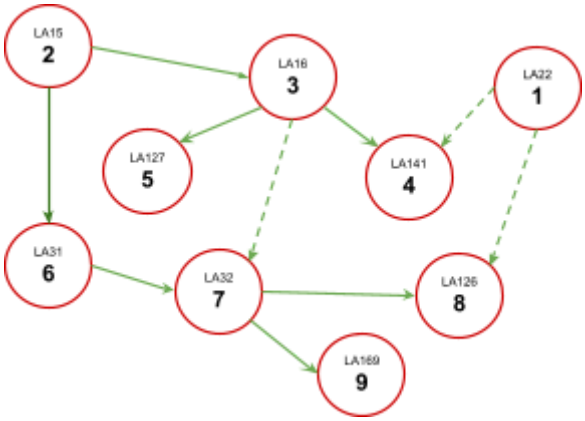


Problem #6

Directed Graph G that Models Situation



Topological Sort of G



Possible Course Sequence:

1. LA22
2. LA15
3. LA16
4. LA141
5. LA127
6. LA31
7. LA32
8. LA126
9. LA169