#1 STRONGLY CONNECTED DIRECTED GRAPH

Create a $O(m+n)$ algorithm to a) determine if there is a path from s to v, b) if there is a path from v to s, and c) prove that a directed graph G is strongly connecting if there is a path from each vertex v to s and from s to each vertex v.

PSEUDO CODE:

```
1  public boolean isStronglyConnected() {
2      boolean visited[] = new boolean[V];
3      for (int i = 0; i < V; i++)
4          visited[i] = false;
5
6      depthFirstSearch(0, visited);
7      for (int i = 0; i < V; i++)
8          if (visited[i] == false)
9              return false;
10
11     Graph transpose = getTranspose();
12     for (int i = 0; i < V; i++)
13         visited[i] = false;
14
15     transpose.depthFirstSearch(0, visited);
16     for (int i = 0; i < V; i++)
17         if (visited[i] == false)
18             return false;
19
20     return true;
21 }
22
23 private Graph getTranspose() {
24     Graph transpose = new Graph(V);
25     for (int v = 0; v < V; v++) {
26         Iterator<Integer> iterator = adj[v]. listIterator ();
27         while (iterator.hasNext())
28             transpose.adj[iterator.next()].add(v);
29     }
30     return transpose;
31 }
32
33 private void depthFirstSearch(int v, boolean visited[]) {
34     visited[v] = true;
35     int node;
36
37     Iterator<Integer> iterator = adj[v]. iterator ();
38     while (iterator.hasNext()) {
39         node = iterator.next();
40         if (! visited[node])
41             depthFirstSearch(node, visited);
42     }
43 }
```

DESCRIPTION:

The first half of the algorithm (up to line 10) accomplishes requirement a by checking to see if each vertex v can be reached from a starting point s. The second part of the algorithm (lines 11 through 21) accomplishes requirement b by reversing the edges on the graph and trying to visit each vertex v starting at s. If each vertex is visited on a reversed graph, then each vertex has a path that leads back to s on the original graph.

ANALYSIS:

A depth first search on a graph takes $O(m+n)$ time. The DFS is used twice in this algorithm. Transposing a graph also takes $O(m+n)$ time and is used once in this algorithm. An array of size n is used to track if a vertex has been visited. This takes $O(n)$ to traverse, and is traversed four times outside of the DFS searches. Thus the total time complexity can be determined as:

$$\begin{aligned} Time: & O(3*(m+n)+4*n) \\ & O(3*m+7*n) \\ & O(m+n) \end{aligned}$$

#2 NUMBER OF SHORTEST PATHS

Design an $O(m+n)$ algorithm to calculate the number of shortest paths from one vertex to another.

PSEUDO CODE:

```
1  int countShortestPaths(int source, int destination) {
2      boolean[] visited = new boolean[V];
3      int [] pathLength = new int[V];
4      int [] numPaths = new int[V];
5      LinkedList<Integer> queue = new LinkedList<>();
6      visited [source]=true;
7      queue.add(source);
8
9      while (queue.size() != 0) {
10         source = queue.poll();
11         Iterator<Integer> iterator = adj[source]. listIterator ();
12         while (iterator.hasNext()) {
13             int node = i.next();
14             if (! visited [node]) {
15                 visited [node] = true;
16                 pathLength[node] = pathLength[source] + 1;
17                 numPaths[node] = numPaths[source];
18                 queue.add(node);
19             }
20             else if (pathLength[node] == pathLength[source] + 1) {
21                 numPaths[node] += numPaths[source];
22             }
23             else if (pathLength[node] < pathLength[source] + 1) {
24                 pathLength[node] = pathLength[source] + 1;
25                 numPaths[node] = numPaths[source];
26             }
27         }
28     }
29     return numPaths[destination];
30 }
```

DESCRIPTION:

The *countShortestPaths* algorithm uses the $BreadthFirstSearch$ algorithm as its base. Two arrays, *numPaths* and *pathLength* of size n are used to track the length of a given path from s to d as well as the number of paths that have that length. The length of the path is stored anytime a node is visited for the first time. If a node is visited a subsequent time, the path length of the new visit is compared with the old visit. If the paths are the same length, the number of paths for that vertex is incremented. If the new path is shorter, the path information is replaced.

ANALYSIS:

A breadth first search takes $O(m+n)$ time. This algorithm uses the same skeleton as a BFS with some additional calculations along the way that doesn't add to the total complexity of the algorithm.

$$Time : O(m+n)$$

#3 NUMBER OF PATHS

Design an $O(m+n)$ algorithm to calculate the number of paths from one vertex to another.

PSUEDO CODE:

```
1  int countPaths(int source, int destination) {
2      boolean[] visited = new boolean[V];
3      int [] numPaths = new int[V];
4      LinkedList<Integer> queue = new LinkedList<>();
5
6       visited [source]=true;
7      queue.add(source);
8
9      while (queue.size() != 0) {
10         source = queue.poll();
11          Iterator<Integer> iterator = adj[source]. listIterator ();
12
13         while (iterator.hasNext()) {
14             int node = i.next();
15
16             if (! visited [node]) {
17                 visited [node] = true;
18                 numPaths[node] = numPaths[source];
19                 queue.add(node);
20             }
21             else {
22                 numPaths[node] += numPaths[source];
23             }
24         }
25     }
26
27     return numPaths[destination];
28 }
```

DESCRIPTION:

The *countPaths* algorithm is essentially identical to the *countShortestPaths* algorithm used previously. The only difference is that the length of the path does not need to be tracked. Therefore, when a node that as already been visited is encountered the number of paths to that node is incremented.

ANALYSIS:

This algorithm performes fewer calculations than *countShortestPaths* without modifying the BFS skeleton. Thus the time complexity is the same as before:

$$Time : O(m+n)$$

#4 K-LINK SHORTEST PATH

Given a destination, source, and maximum number of edges (k), determine the shortest path in a directed acyclic graph in $O(k(m+n))$ time which uses no more than k edges.

PSUEDO CODE:

```
1  private int shortestPath(int source, int destination, int maxEdges) {
2      Stack topologicalOrder = new Stack();
3      int cost[] = new int[V];
4      int edgeCount[] = new int[V];
5      Boolean visited[] = new Boolean[V];
6      for (int i = 0; i < V; i++) {
7          visited[i] = false;
8          cost[i] = INF;
9          edgeCount[i] = INF;
10     }
11     for (int i = 0; i < V; i++)
12         if (visited[i] == false)
13             topologicalSort(i, visited, topologicalOrder);
14
15     cost[source] = 0;
16     edgeCount[source] = 0;
17     while (!topologicalOrder.empty()) {
18         source = (int) topologicalOrder.pop();
19         Iterator<AdjListNode> iterator;
20
21         if (cost[source] != INF) {
22             iterator = adj[source].iterator();
23             while (iterator.hasNext()) {
24                 AdjListNode node = iterator.next();
25                 if (cost[node.getV()] > cost[source] + node.getWeight()
26                 && edgeCount[source] < maxEdges)) {
27                     cost[node.getV()] = cost[source] + node.getWeight();
28                     edgeCount[node.getV()] = edgeCount[source] + 1;
29             }
30         }
31     }
32     return cost[destination];
33 }
34
35 private void topologicalSort(int v, Boolean visited[], Stack stack) {
36     visited[v] = true;
37     Iterator<AdjListNode> iterator = adj[v].iterator();
38     while (iterator.hasNext()) {
39         AdjListNode node = iterator.next();
40         if (! visited[node.getV()])
41             topologicalSortUtil(node.getV(), visited, stack);
42     }
43     stack.push(new Integer(v));
44 }
```

```
45
46 public int kLinkShortestPath(int source, int destination, int maxEdges) {
47         int cost[] = new int[maxEdges];
48         for (int i = 1; i <= maxEdges; i++) {
49         cost[i] = shortestPath(source, destination, i);
50     }
51
52     int bestCost = cost[1];
53     for (int i = 2; i <= maxEdges; i++) {
54         if (cost[i] < bestCost) {
55                 bestCost = cost[i];
56         }
57     }
58     return bestCost;
59 }
```

DESCRIPTION:

The *shortestPath* algorithm utilizes topological sorting in order to determine the shortest path from s to t. First the algorithm sorts the graph topologically using the recursive function. Then it visits each vertex in order, checking to see if there is a shorter, or more cost effective path. The cost from the source to any given vertex is stored in the *cost* array. The number of edges from the source to any given vertex is stored in the *edgeCount* array. Whenever the algorithm finds a more cost effective path to the child of the current node, it checks to make sure that adding an extra edge won't exceed the maximum number of edges allowed. If the new path passes these test, the *cost* and *edgeCount* for the child node is updated.

This process is nested inside of a for loop that iterates through values $[1..k]$. The *shortestPath* algorithm will return the shortest path to the destination using only at most the current value given for *maxEdges*. The final *cost* array will contain the shortest path using only the number of edges represented by the index of *cost*. A search through the array will locate and return the minimum value.

ANALYSIS:

A topological sort takes $O(m + n)$ time. The remainder of the *shortestPath* algorithm iterates through the topological sort in $O(m + n)$ time as well. The *shortestPath* algorithm is repeated k times in order to find the shortest path using only $[1..k]$ edges. The minimum value found in the array after an $O(k)$ search is the desired solution.

$$Time : O(2 * k * (m + n) + k)$$
$$O(k(m + n))$$