1. Depth first search with backtracking can be used here. Keep an array of boolean values to keep track of whether you visited a node before. If you run out of new nodes to go to (without reaching a node you have already been), then just backtrack and try a different branch. Below is the pseudocode for this:

```
Algorithm 1: findCycles
  Data: Graph
  Result: Print all cycles in the graph
  visited ← [NOT_VISITED];
  for vertex v ∈ Graph.vertices do
    if visited[v] == NOT_VISITED then
      stack ← [];
      stack.push(v);
      visited[v] ← IN_STACK;
      processDFSTree(Graph, stack, visited);
    end
  end
```

```
Algorithm 2: processDFSTree
  Data: Graph, Stack, visited
  Result: Print all cycles in the current DFS Tree
  for vertex v in Graph.adjacent[Stack.top] do
    if visited[v] == IN_STACK then
      printCycle(Stack, v);
    else if visited[v] == NOT_VISITED then
      Stack.push(v);
      visited[v] ← IN_STACK;
      processDFSTree(Graph, Stack, visited);
  end
  visited[Stack.top] = DONE;
  Stack.pop();
```

```
Algorithm 3: printCycle
  Data: Stack, v
  Result: print the cycle that lives in the stack starting from vertex v
  Stack2 ← [];
  Stack2.push(Stack.top);
  Stack.pop();
  while Stack2.top ≠ v do
    Stack2.push(Stack.top);
    Stack.pop()
  end
  while not Stack2.empty() do
    print(Stack2.top);
    Stack.push(Stack2.top);
    Stack2.pop()
  end
```

Time Complexity: $O(V+E)$ same as the time complexity of DFS traversal.

2. To solve this question we will utilize the fact that the given graph is a DAG, and use DFS to sort the vertices in topological order, and then visit vertices in reverse topological order and increment a paths counter for each vertex as follows:

Algorithm 1 PATHS(G)

```
topologically sort the vertices of G
for each vertex u, taken in reverse topologically sorted order do
    for each vertex v ∈ G.Adj[u] do
        u.paths = u.paths + 1 + v.paths
    end for
end for
```

Time Complexity: $O(V + E)$ mostly due to topological sort.

3. The input is a graph, a source vertex, and the output of a single-source shortest path algorithm in the form of a d and a π array. The task if to provide and algorithm to verify if the d and π arrays are indeed valid outputs of a single-source shortest path algorithm.

   (a) Verify that $s.d = 0$ and $s.\pi = Null$

   (b) Verify that $v.d = v.\pi.d + w(v.\pi, v)\ \forall\ v! = s$

   (c) Verify that $v.d = \infty$ if and only if $v.\pi = Null\ \forall\ v! = s$

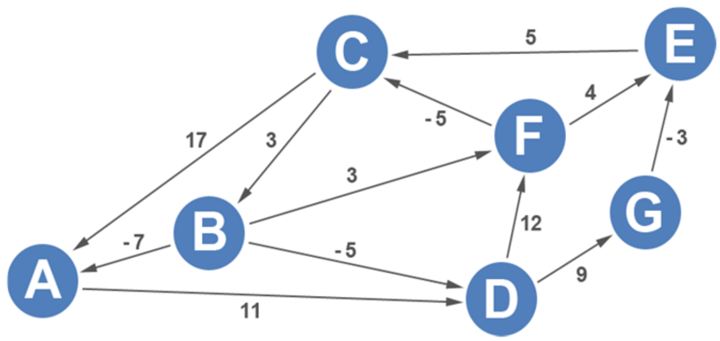
   (d) If any of above verification tests fail, declare the output to be incorrect. Otherwise, run one pass of Bellman-Ford, i.e. relax each edge$(u, v) \in E$ one time. If any values of $v.d$ changes, then declare the output to be incorrect; otherwise, declare the output to be correct.

   Time Complexity: $O(V + E)$

4. Assume that $\infty - \infty$ is undefined; in particular, it's not 0.

   Let $G = (V, E)$, where $V = t, u,\ E = (u, t)$ and $w(u, t) = 0$. There is only one edge, and it enters t. When we run Bellman-Ford from t, we get $h(t) = \delta(t, t) = 0$ and $h(u) = \delta(t, u) = \infty$. When we re-weight, we get $\hat{w}(u, t) = 0 + \infty - 0 = \infty$. We compute $\hat{\delta} = \infty$, and so we compute $d_{ut} = \infty + 0 - \neq 0$, we get an incorrect answer

   Johnson's algorithm avoids this problem by adding a new vertex s to the graph, with zero-weight edges going from s to every other vertex, but no edges going back into s. This addition does not change the shortest paths between any other pair of vertices, because there are no paths into s.
   For example:



Johnson's algorithm starts by introducing a source vertex. The problem with choosing an existing vertex is that it might not be able to reach all the vertices in the graph. To guarantee that a single source vertex can reach all vertices in the graph, a new vertex is introduced. The new vertex, s, is introduced to all vertices in the graph with weight zero.

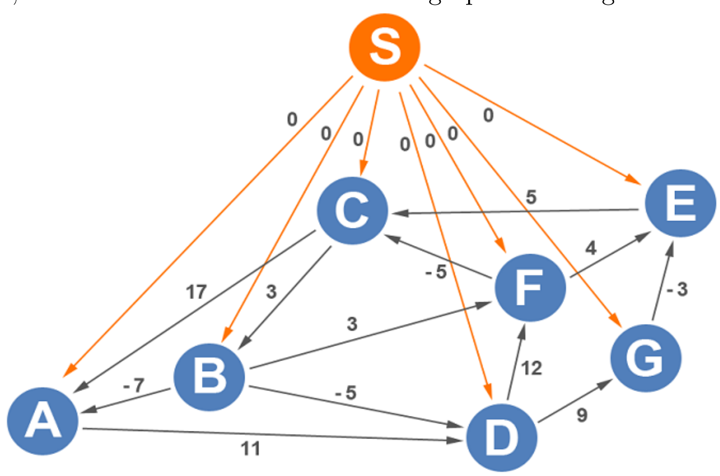


Image reference: https://bit.ly/2UHoktl

5. If all the edge weights are non-negative, then the values computed as the shortest distance when running Bellman-Ford will be all 0. This is because when constructing G' on the first line of Johnson's algorithm, we place an edge of weight 0 from s to every other vertex. Since any path within the graph has no negative edges, its cost cannot be negative, and so, cannot beat the trivial path that goes straight from s to any given vertex. Since we have that $h(u) = h(v)$ for every u and v, the re-weighting that occurs only adds and subtracts 0, and so we have that $w(u, v) = \hat{w}(u, v)$.