Homework 4
aryanrao

1. **Pseudocode:**

```
function hasCycle(graph):
    visited = array of size V (number of vertices), initialized as false
    parent = array of size V, initialized as -1

    for each vertex v in graph:
        if not visited[v]:
            if dfs(v, visited, parent):
                return true

    return false

function dfs(vertex, visited, parent):
    visited[vertex] = true

    for each adjacent vertex u of vertex:
        if not visited[u]:
            if dfs(u, visited, parent):
                return true
        else if u is not parent[vertex]:
            return true

    return false
```

Algorithm:

The following describes the algorithm for using DFS to find a cycle in an undirected graph:

- To keep track of visited vertices, create a visited array. Make sure to initialize all vertices as unvisited.
- If a vertex v in the graph is not visited, run the DFS function for that vertex.
- Initialize a parent array to keep track of the parent of each visited vertex and mark the current vertex as visited in the DFS function. Iterate through all adjacent vertices of the current vertex: a. If the adjacent vertex u is not visited, recursively call the DFS subroutine on u, and set the parent of u as the current vertex. b. If the adjacent vertex u is visited and u is not the parent of the current vertex, then a cycle exists.
- If no cycle is found after exploring all vertices, return false.

V represents the number of vertices, and the graph is assumed to be represented using an adjacency list or matrix.
By invoking the **hasCycle** function with the graph as input, it will return **true** if a cycle exists in the graph, and **false** otherwise.

2. **Pseudocode:**

```
function totalDistanceTraveled(graph, doctorResidence):
    distances = array of size V (number of vertices), initialized with infinity
    distances[doctorResidence] = 0

    priorityQueue = MinHeap data structure, initially empty
    priorityQueue.insert((0, doctorResidence))

    while priorityQueue is not empty:
        (currentDistance, currentVertex) = priorityQueue.removeMin()

        if currentDistance > distances[currentVertex]:
            continue

        for each neighbor in graph.adjacentVertices(currentVertex):
            edgeWeight = L (equal length of roads)
            newDistance = currentDistance + edgeWeight

            if newDistance < distances[neighbor]:
                distances[neighbor] = newDistance
                priorityQueue.insert((newDistance, neighbor))

    totalDistance = 0

    for each house in the village:
        totalDistance += (2 * distances[house])

    return totalDistance
```

Algorithm:

In this algorithm, V represents the number of vertices (houses) in the village. The graph is assumed to be represented using an adjacency list or matrix, and **graph.adjacentVertices(currentVertex)** returns a list of neighbors of the current vertex.

The algorithm starts with initializing the distances array with infinity, except for the doctor's residence, which is set to 0. It uses a priority queue (MinHeap) to process vertices in order of their distances from the doctor's residence. It iterates until the priority queue is empty, updating the distances as it finds shorter paths. Finally, it calculates the total distance traveled by summing up twice the distances to each house in the village.

By invoking the **totalDistanceTraveled** function with the graph and the doctor's residence as input, it will return the total distance traveled by the doctor every day in the village.

3. **Pseudocode:**

```
function countShortestPaths(graph, s):
    n = number of vertices in the graph
    distances = array of size n, initialized with infinity
    counts = array of size n, initialized with 0
    counts[s] = 1

    queue = empty queue data structure
    queue.enqueue(s)
    distances[s] = 0

    while queue is not empty:
        current = queue.dequeue()

        for each neighbor in graph.adjacentVertices(current):
            if distances[neighbor] == infinity:
                queue.enqueue(neighbor)
                distances[neighbor] = distances[current] + 1

            if distances[neighbor] == distances[current] + 1:
                counts[neighbor] += counts[current]

    return counts
```

Algorithm:

In this algorithm, **graph.adjacentVertices(current)** returns a list of neighbors of the current vertex.
The algorithm initializes two arrays: **distances** to store the shortest distances from the source node **s** to each vertex, and **counts** to store the number of shortest paths from **s** to each vertex. Initially, all distances are set to infinity, except for **distances[s]** which is set to 0, and **counts[s]** which is set to 1.
The algorithm then uses a queue to perform BFS traversal. For each vertex dequeued from the queue, it examines its neighbors. If a neighbor has a distance of infinity, it means it hasn't been visited yet, so it is enqueued and its distance is updated. If the neighbor has a distance that is one unit greater than the current vertex, it means it is on the next layer of the BFS tree. In this case, the count of shortest paths from **s** to the neighbor is incremented by adding the count of shortest paths from **s** to the current vertex.
Finally, the algorithm returns the **counts** array, which represents the number of shortest paths from **s** to each vertex in the graph.
The runtime complexity of this algorithm is O(V + E), where V is the number of vertices and E is the number of edges in the graph. This is because the algorithm performs a BFS traversal, visiting each vertex and each edge once. The enqueue and dequeue operations take O(1) time, and each vertex and edge are processed once, resulting in a linear runtime complexity.

4. **Pseudocode:**

```
function hasEntityToSendToAll(graph, n):
    visited = array of size n, initialized with false
    sender = null

    for i from 0 to n-1:
        if not visited[i]:
            if dfs(graph, i, visited):
                sender = i
            else:
                return false

    return sender is not null

function dfs(graph, vertex, visited):
    visited[vertex] = true

    for each neighbor in graph.adjacentVertices(vertex):
        if not visited[neighbor]:
            if not dfs(graph, neighbor, visited):
                return false

    return true
```

Algorithm:

In this algorithm, **graph.adjacentVertices(vertex)** returns a list of neighbors of the current vertex.

The algorithm starts by initializing an array **visited** to keep track of visited vertices and a variable **sender** as **null**.

Then, for each entity (vertex) in the network, it checks if it hasn't been visited yet. If so, it calls the **dfs** subroutine to perform a DFS traversal starting from that vertex. If the **dfs** subroutine returns true, it means that all entities can be reached from the current vertex. In this case, it updates the **sender** variable to the current vertex. If the **dfs** subroutine returns false, it means that not all entities can be reached from the current vertex, so the algorithm immediately returns false.

Finally, after the traversal is complete, it checks whether a sender has been found (**sender** is not **null**). If so, it means there exists an entity capable of sending information to all other entities, and it returns true. Otherwise, it returns false.

The runtime complexity of this algorithm is O(V + E), where V is the number of vertices (entities) and E is the number of edges (edge relationships) in the graph. This is because the algorithm performs a DFS traversal, visiting each vertex and each edge once, resulting in a linear runtime complexity.