Homework 4

Date: 2023-07-11 16:20
type: #Homework
Class: COM S 311

Graphs

Question 1

Given an undirected graph $G = (V, E)$, write an algorithm to determine whether there exists a cycle in the graph.

To determine whether there is a cycle in a graph we can perform a DFS search, and if there is a vertex we have already visited (in parent array) that is attempting to be searched again, then there exists a cycle (there is a back edge in the graph).

```
hasCycle(graph){
    visited = [false] * graph.numVertices
    parent = [-1] * graph.numVertices

    for each vertex v in graph.vertices
        if not visited[v]
            if dfsHasCycle(graph, v, visited, parent)
                return true

    return false
}

dfsHasCycle(graph, vertex, visited[], parent){
    visited[vertex] = true

    for each neighbor in graph.adjacentVertices(vertex)
        if not visited[neighbor]
            parent[neighbor] = vertex
            if dfsHasCycle(graph, neighbor, visited, parent)
                return true
        else if neighbor != parent[vertex]
            return true

    return false
}
```

Question 2

Let $G = (V, E)$ be a directed graph representing the road network of the village Totorum. The roads are one-way represented by directed edges and are of equal length L; each house in the village is represented by the nodes in the graph. Every day, the village doctor visists every house in the graph. For this, the doctor does the following:

```
For every house v in the village
       the doctor takes the shortest path from her residence to v
       the doctor takes the shortest path from v to her residence
```

Determine the total distance traveled by the doctor every day. (*Assume every house can be reached from the doctor's residence*)

We can find the total distance traveled by the doctor by calculating the sum of the shortest paths; if we use a DFS algorithm with a shortest path calculation function we have the following:

```
Dijkstra(graph, source){
   distance[source] = 0
   parent[source] = null

   priorityQueue.add(source, 0)

   while priorityQueue is not empty {
       current = priorityQueue.removeMin()

       for each neighbor in graph.adjacentNodes(current) {
           newDistance = distance[current] + L

           if newDistance < distance[neighbor]
               distance[neighbor] = newDistance
               parent[neighbor] = current
               priorityQueue.update(neighbor, newDistance)
               }
       }

   return distance
}

calculateTotalDistance(graph, doctorResidence){
   totalDistance = 0

   distance = Dijkstra(graph, doctorResidence)

   for each house in graph.nodes {
       distanceToHouse = Dijkstra(graph, house)
       totalDistance += distance[house] + distanceToHouse[doctorResidence]
       }
```

```
    return totalDistance
}
```

Question 3

Give an algorithm that, given an undirected graph G and a node s, creates an array `ShortestCount` in which `ShortestCount[i]` is the *number* of shortest paths from s to vertex `i`. Derive its runtime.

(Start with the BFS algorithm as given in the lecture, in which nodes are organized into layer l_i based on distance from s, and update the counts as you build the tree)

```
calculateShortestCount(graph G, starting_node s){
    ShortestCount = array of size graph.numVertices
    visited = array of size graph.numVertices
    queue = empty queue
    layer = array of size graph.numVertices

    ShortestCount[s] = 1
    visited[s] = true
    layer[s] = 0

    enqueue(queue, s)

    while queue is not empt     {
        u = dequeue(queue)

        for each neighbor v of u in graph.adjacentVertices(u) {
            if not visited[v]
                visited[v] = true
                layer[v] = layer[u] + 1
                ShortestCount[v] = ShortestCount[u]
                enqueue(queue, v)

            else if layer[v] == layer[u] + 1
                ShortestCount[v] += ShortestCount[u]
                }
        }
    return ShortestCount
}
```

- `ShortestCount` represents the number of shortest paths, initialized with all 0
- `visited` represents visited nodes, initialized with all false
- `layer` represents distance of each vertex, initialized with infinity (unknown)

Because the algorithm performs a BFS search, which checks every node V and it's neighbors E *once*, the runtime will be $O(V+E)$.

Question 4

We are given an information network where the edges represent sender-receiver relationship. I.e., if there is an edge from x to y, then x sends information to y. Any information received or originated by a node x will be received by all y such that there is an edge from x to y. For exmaple if there is an edge from x to y, edge from y to z. Any information originated at x is received by y, which in turn is received by z. Given a network of n entities and m edge relationships, give an algorithm that checks whether there exists an entity, which is capable of sending information to all other entities.

Use DFS and an array `reachable` to keep track of the entities that are able to be receiver within the graph, if reachable is n-1 in size then it should be able to see every other node:

```
 dfs(entity, graph, visited, reachable){
    visited.add(entity)
    reachable.add(entity)

    for each neighbor in graph[entity]{
        if neighbor not in visited
            dfs(neighbor, graph, visited, reachable)
        }
}

find_entity(graph){
    entities = keys(network)


    for each entity in entities
        visited = empty set
        reachable = empty set

        dfs(entity, graph, visited, reachable)

        if size(reachable) == size(entities) - 1
            return entity

    return null
}
```