Assignment 4

Jasraj Singh Johal | 400434346 | johalj11

Question 1: Unweighted, Undirected Graphs

```
def findCenters(g: Graph):
    # Initialize an empty list to store the center vertices
    centers = []

    # Function to calculate eccentricity using BFS from a given start vertex
    def calculateEccentricity(start):
        # Initialize arrays to keep track of visited vertices and distances from
the start
        visited = [False] * num_vertices(g)
        distances = [-1] * num_vertices(g)

        # Initialize a queue for BFS and enqueue the start vertex
        open = Queue()
        open.put(start)
        visited[start] = True
        distances[start] = 0

        # Perform BFS to calculate distances to all other vertices
        while not open.empty():
            v = open.get(True)  # Fix: use True for blocking get
            # Explore neighbors of the current vertex
            for n in adj(g, v):
                if not visited[n]:
                    open.put(n)
                    visited[n] = True
                    distances[n] = distances[v] + 1

        # Return the maximum distance, representing the eccentricity
        return max(distances)

    # Initialize the radius to positive infinity
    radius = float('inf')

    # Iterate over all vertices to find the minimum eccentricity (radius)
    for v in range(num_vertices(g)):
        eccentricity = calculateEccentricity(v)
        if eccentricity < radius:
            radius = eccentricity

    # Iterate over all vertices again to find vertices with eccentricity equal to
the radius (centers)
    for v in range(num_vertices(g)):
```

```
        eccentricity = calculateEccentricity(v)
        if eccentricity == radius:
            centers.append(v)

    # Return the list of center vertices
    return centers
```

Assumptions:

Graph ADT:

- The graph is represented using the `Graph` class, which includes methods such as `add_edge`, `adj`, `num_vertices`, and `num_edges`.
- The graph is assumed to be unweighted and undirected, and it is represented as an adjacency list.

Queue ADT:

- The Breadth-First Search (BFS) algorithm uses the `Queue` class from the `queue` module for implementing the queue data structure.
- The `Queue` class provides the `put` and `get` methods for enqueueing and dequeueing elements, respectively.

Other: The implementation utilizes basic Python data structures such as lists to represent arrays and to store the list of center vertices. The `float('inf')` is used to initialize the radius to positive infinity. The code assumes a connected graph. If the graph is disconnected, each vertex has infinite eccentricity, and the concept of a "center" may not be relevant.

Justification for Correctness:

- BFS computes the shortest path from the start vertex `s` to every other vertex that is connected to `s`.
- The algorithm finds all vertices connected to `s` and examines them in order of their distance from `s`.
- The queue in BFS always consists of vertices at distance `k` followed by vertices at distance `k + 1`.
- The function `calculateEccentricity` utilizes BFS to find the maximum distance from the start vertex to any other vertex, effectively computing the eccentricity.

Worst-Case Time Complexity:

- BFS runs in $O(V + E)$ time, where V is the number of vertices and E is the number of edges.
- Creating the `marked` and `distances` arrays in `calculateEccentricity` is $O(V)$.

- Each connected edge is considered twice in BFS, which is $O(E)$.
- Since the function calls `calculateEccentricity` for each of the V vertices, the overall worst-case time complexity is $O(V * (V + E))$.
- In the case of a connected graph, the worst-case can be further simplified to $O(V^2)$, as E is in the order of V.

Question 2:

```
def reversePostorder(graph):
    visited <- array of size V initialized to false  # Mark visited vertices
    reversePost <- empty stack  # Stack to store reverse postorder

    function dfs(vertex):
        visited[vertex] <- true  # Mark current vertex as visited
        for each neighbor in graph.adjacencyList[vertex]:
            if not visited[neighbor]:
                dfs(neighbor)  # Recursive DFS on unvisited neighbors
        reversePost.push(vertex)  # Push vertex onto stack after visiting
neighbors

    for each vertex in graph.vertices:
        if not visited[vertex]:
            dfs(vertex)  # Perform DFS on unvisited vertices

    return reversePost  # Stack contains reverse postorder

# Function to check if a given permutation is a topological order of a DAG
def isTopologicalOrder(permutation, digraph):
    # Get the number of vertices in the permutation
    n <- length(permutation)

    # Array to mark visited vertices
    marked <- array of size n initialized to false

    # Array to store the position of each vertex in the permutation
    position <- array of size n

    # Assign positions to vertices in the permutation
    for i from 0 to n-1:
        vertex <- permutation[i]
        position[vertex] <- i

    # Check topological order conditions
```

```
    for i from 0 to n-1:
        currentVertex <- permutation[i]
        marked[currentVertex] <- true

        # Check neighbors of the current vertex
        for neighbor in digraph.adj(currentVertex):
            if not marked[neighbor]:
                # If a neighbor is not visited yet, it violates topological order
                return false
            elif position[currentVertex] < position[neighbor]:
                # If the neighbor appears after the current vertex, it violates
topological order
                return false

    # All vertices have been visited, and the order is valid
    return true
```

Assumptions:

- The pseudocode assumes the availability of standard ADTs like Queue, Stack, and Digraph (directed graph) along with the provided **DepthFirstOrder** and **Topological** classes.
- It assumes that the input DAG is represented by an adjacency list, as indicated by the **G.adj(v)** operation in the DFS.
- The pseudocode assumes that the input permutation is a valid ordering of vertices (i.e., it doesn't contain duplicate vertices and includes all vertices of the graph).
- The time complexity analysis assumes that the operations on the ADTs (e.g., enqueue, dequeue, push, pop) take constant time.

Correctness:

reversePostorder Function:

The reversePostorder function uses Depth-First Search (DFS) to traverse the graph and construct a reverse postorder list. The correctness of this function is established by Proposition F mentioned in the initial explanation, which states that the reverse postorder in a DAG is a topological sort. The function constructs the reverse postorder correctly.

isTopologicalOrder Function:

The isTopologicalOrder function checks whether a given permutation is a valid topological order. It correctly verifies that each edge in the permutation goes from a vertex earlier in the order to a vertex later in the order, ensuring that it satisfies the conditions of a topological order.

Time Complexity:

reversePostorder Function:

The time complexity of the reversePostorder function is $O(V + E)$, where V is the number of vertices and E is the number of edges. This is because the function performs a DFS, visiting each vertex and edge once.

isTopologicalOrder Function:

The time complexity of the isTopologicalOrder function is $O(V + E)$, where V is the number of vertices and E is the number of edges. The function iterates through the given permutation and checks each edge in the adjacency list, and both operations are proportional to the size of the graph.

Reference: (Book pg 581) This DepthFirstOrder and DirectedCycle client returns a topological order for a DAG. The test client solves the precedence-constrained scheduling problem for a SymbolDigraph. The instance method order() returns null if the given digraph is not a DAG and an iterator giving the vertices in topological order otherwise.

Question 3:

```
def minimum_spanning_tree_with_edges(G, S):
    # Initialize an empty minimum spanning tree T
    T = set()

    # Initialize a disjoint set data structure to keep track of connected
components
    parent = {}  # Dictionary to represent the parent of each vertex

    def make_set(u):
        # Create a set for each vertex if not already created
        if u not in parent:
            parent[u] = u

    def find(u):
        # Find the representative of the set containing vertex u
        if parent[u] != u:
            parent[u] = find(parent[u])  # Path compression
        return parent[u]

    def union(u, v):
        # Merge the sets of u and v
        parent[find(u)] = find(v)

    # Add all edges from S to T
```

```
    for (u, v) in S:
        make_set(u)
        make_set(v)
        if find(u) != find(v):
            T.add((u, v))
            union(u, v)

    # Sort the remaining edges of G in increasing order of weights (optional if S
already forms a connected graph)

    # Process the remaining edges
    for (u, v) in sorted(G.edges(), key=lambda e: G.weight(e)):
        if (u, v) not in T:
            if find(u) != find(v):
                T.add((u, v))
                union(u, v)

    return T
```

Assumptions:

- The input graph `G` is assumed to have an `edges()` method that returns a list of edges and a `weight(e)` method that returns the weight of edge `e`.
- Edges in the graph are assumed to be represented as pairs `(u, v)` where `u` and `v` are vertices.
- The graph `G` is assumed to be connected, meaning there is a path between any pair of vertices. If the graph is not connected, the algorithm may not find a spanning tree containing all vertices.
- The pseudocode assumes the existence of a disjoint set data structure with `MakeSet`, `Find`, and `Union` operations.

Correctness:

Adding Edges from S: The algorithm starts by adding all edges from the specified set `S` to the minimum spanning tree `T`. This guarantees that the resulting tree is connected and acyclic because `S` is assumed to contain edges that do not form cycles.

Processing Remaining Edges: The algorithm then processes the remaining edges from `G`. It checks whether adding each edge creates a cycle using the union-find data structure. If adding an edge doesn't create a cycle, it is added to the minimum spanning tree. The correctness is ensured by the properties of minimum spanning trees and the logic of union-find operations.

Time Complexity:

- The primary operations in the algorithm are the union-find operations (MakeSet, Find, Union), which have a time complexity of (O(log n)) each. The loops that process edges from (S) and the remaining edges from (G) contribute (O(E)), where (E) is the number of edges.
- Therefore, the overall time complexity is (O(E log n)), where (E) is the number of edges and (n) is the number of vertices. This is because the union-find operations dominate the overall complexity, and (E >= V - 1) for a connected graph with (V) vertices.
- The space complexity is (O(n + E)) because of the space required for the union-find data structure (parent dictionary) and the set of edges in the minimum spanning tree.