Student Name: Ganesh Nathan
Student ID: 2021BEC00104

ISC 225 Datastrucutres-III
Assignment 2

1 Problem statement

1. Detect all cycles in an undirected graph using BFS and DFS.
2. Program to find the length of shortest cycle and longest cycle in an undirected graph.
3. Program to find the number of cycles in an undirected graph.
4. Given an undirected graph represented as an adjacency matrix, write a function to determine if the graph contains any cycles. Your function should return the edges that form the cycle.

2 Solution Description

1. Detect all cycles in an undirected graph using BFS and DFS.

Solution:

We begin by selecting any vertex in the graph and adding it to the visited list and queue. Following that, we conduct a typical BFS traversal with an additional step to look for cycles. We add the unvisited neighbours of each visited vertex to the queue and mark them as visited. A cycle has been discovered if a visited vertex that we come across is not the present vertex's parent. To find the cycle, we use parent pointers to go backwards from the current vertex to the visited vertex. Lastly, we repeat the procedure for any graph vertex that has not yet been visited. A list of each cycle in the graph is the output.

2. Program to find the length of shortest cycle and longest cycle in an undirected graph.

Solution:

We may use BFS to determine the size of the shortest cycle in an undirected graph. We begin by selecting any vertex in the graph and adding it to the visited list and queue. Following that, we conduct a typical BFS traversal with an additional step to look for cycles. We add the unvisited neighbours of each visited vertex to the queue and mark them as visited. A cycle has been discovered if a visited vertex that we come across is not the present vertex's parent. To find the cycle, we use parent pointers to go backwards from the current vertex to the visited vertex. The shortest cycle's length is equal to the minimum length of all cycles discovered by the BFS traversal.

3. Program to find the number of cycles in an undirected graph.

Solution:

We can use DFS to determine the number of cycles in an undirected graph. Starting with any vertex in the graph, we traverse it using a DFS algorithm. We add the unvisited neighbours of each visited vertex to the DFS stack and mark them as visited. A cycle has been discovered if a visited vertex that we come across is not the present vertex's parent. To find the cycle, we use parent pointers to go backwards from the current vertex to the visited vertex. The overall count of all cycles discovered during the DFS traversal is represented by the number of cycles in the graph.

4. Given an undirected graph represented as an adjacency matrix, write a function to determine if the graph contains any cycles. Your function should return the edges that form the cycle.

Solution:

We can use DFS to find any cycles in an undirected graph represented as an adjacency matrix and return the edges that make up the cycle. Starting with any vertex in the graph, we traverse it using a DFS algorithm. We add the unvisited neighbours of each visited vertex to the DFS stack and mark them as visited. A cycle has been discovered if a visited vertex that we come

3 Pseudocode

```
Problem 1 To detect all cycles in an undirected graph using BFS:
1. Initialize an empty queue and an empty visited list.
2. Pick any vertex v in the graph and add it to the queue and the visited list.
3. While the queue is not empty, do the following:
1.Dequeue a vertex u from the queue.
2.For each neighbor w of u, do the following:
2.1 If w is not in the visited list, add it to the queue and the visited list and set the parent of w to u.
2.2 If w is already in the visited list and w is not the parent of u, a cycle is found. Store the cycle in a list by backtracking from u to w through the parent pointers.
4. Repeat step 2 for any unvisited vertex in the graph.
```

4 Implementation

Problem 1

```python
def find_cycles(graph):
    cycles = []
    visited = set()
    for v in graph.keys():
        if v not in visited:
            queue = [(v, None)]
            cycle = []
            while queue:
                u, parent = queue.pop(0)
                if u in visited:
                    if parent is not None:
                        cycle = backtrack(u, parent)
                        cycles.append(cycle)
                else:
                    visited.add(u)
                    for w in graph[u]:
                        queue.append((w, u))
            cycles.append(cycle)
    return cycles

def backtrack(u, parent):
    cycle = [u, parent]
    while parent != u:
        u, parent = parent, cycle[-1]
        cycle.append(parent)
    return cycle

#main code
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D'],
    'D': ['B', 'C', 'E'],
    'E': ['D', 'F'],
    'F': ['E']
}
print(find_cycles(graph))
```

Problem 2

```python
def find_shortest_cycle(graph):
    shortest_cycle = float('inf')
    visited = set()
    for v in graph.keys():
        if v not in visited:
            queue = [(v, None, 0)]
            while queue:
                u, parent, length = queue.pop(0)
                if u in visited:
                    if parent is not None:
                        shortest_cycle = min(shortest_cycle, length)
                else:
                    visited.add(u)
                    for w in graph[u]:
                        queue.append((w, u, length + 1))
    return shortest_cycle

def find_longest_cycle(graph):
    longest_cycle = 0
    visited = set()
    for v in graph.keys():
        if v not in visited:
            stack = [(v, None, 0)]
            while stack:
                u, parent, length = stack.pop()
                if u in visited:
                    if parent is not None:
                        longest_cycle = max(longest_cycle, length)
                else:
                    visited.add(u)
                    for w in graph[u]:
                        stack.append((w, u, length + 1))
    return longest_cycle

#main code
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D'],
    'D': ['B', 'C', 'E'],
    'E': ['D', 'F'],
    'F': ['E']
}

print(find_longest_cycle(graph))
print(find_shortest_cycle(graph))
```

Problem 3

```
def find_cycle_count(graph):
    cycle_count = 0
    visited = set()
    for v in graph.keys():
        if v not in visited:
            stack = [(v, None)]
            while stack:
                u, parent = stack.pop()
                if u in visited:
                    if parent is not None and parent != u:
                        cycle_count += 1
                else:
                    visited.add(u)
                    for w in graph[u]:
                        stack.append((w, u))
    return cycle_count
#main code
n=int(input("Enter the number od nodes;"))
m=[[1,0,0],[0,0,1]]
print(find_cycle_count(m))#pass the adjacecy matrix into this funtion
```

Problem 4

```
def find_cycle(graph):
    cycle = []
    visited = set()
    parent = {0: None}  # assuming vertex 0 is the root
    stack = [(0, None)]
    while stack:
        u, p = stack.pop()
        if u in visited:
            if parent[u] is not p:
                # found a cycle
                v = u
                while v != p:
                    cycle.append((v, parent[v]))
                    v = parent[v]
                cycle.append((u, p))
                return cycle
        else:
            visited.add(u)
            parent[u] = p
            for v in range(len(graph)):
                if graph[u][v] == 1:
                    stack.append((v, u))
    return None

#main code
n=int(input("Enter the number od nodes;"))
m=[[1,0,0],[0,0,1]]
find_cycle(m)#pass the adjacecy matrix into this funtion
```

5 Algorithm Analysis

1.Time Complexity: if an undirected graph represented as an adjacency matrix contains any cycles and return the edges that form the cycle is O(V+E), where V is the number of vertices in the graph and E is the number of edges in the graph. This is because we perform a DFS traversal on the graph, and in the worst case, we visit every vertex and edge once.

2.Space complexity: O(V+E)
where V is the number of vertices in the graph and E is the number of edges in the graph. This is because we maintain a set of visited vertices, a dictionary of parent vertices, and a stack of vertices and their parents during the DFS traversal. In the worst case, the stack can contain all vertices and their parents, which results in a space complexity of O(V+E).

6 Reference

LeetCode: https://leetcode.com/