Istanbul Technical University
Faculty of Computer and Informatics
Computer Engineering Department

BLG 336E
Assignment 1 Report

Ömer Malik Kalembaşı - 150180112

March 28th, 2023

1 Explain your code and your solution.

1.1 Write your pseudo-code

The bfs function performs a breadth-first search on the graph to find the shortest path from a source node to a target node. It takes in the input graph, the number of children, the source node, and the target node as arguments. The function uses a queue to visit each node and stores the distance of each node from the source node in a vector. It also stores the previous node in the shortest path in another vector. The function returns a string representation of the shortest path from the source to the target node.

Algorithm 1 BFS

```
1: function BFS(graph, numOfKids, source, target)
2:     dist ← vector of length numOfKids filled with INT_MAX
3:     prev ← vector of length numOfKids filled with −1
4:     q ← empty queue
5:     dist[source] ← 0
6:     q.push(source)
7:     while q is not empty do
8:         currKid ← q.front()
9:         q.pop()
10:        for neighbor ← 0 to numOfKids − 1 do
11:            if graph[currKid][neighbor] = 1 and dist[neighbor] = INT_MAX then
12:                dist[neighbor] ← dist[currKid] + 1
13:                prev[neighbor] ← currKid
14:                q.push(neighbor)
15:            else if graph[currKid][neighbor] = 1 and dist[neighbor] > dist[currKid] + 1
   then
16:                dist[neighbor] ← dist[currKid] + 1
17:                prev[neighbor] ← currKid
18:            end if
19:        end for
20:    end while
21:    if prev[target] = −1 then
22:        return "-1"
23:    end if
24:    path ← to_string(target)    currKid ← target
25:    while prev[currKid] ≠ −1 do
27:        path ← to_string(prev[currKid]) + "− > "+path        currKid ← prev[currKid]
28:    end while
30:    return path
31: end function
```

The dfs function performs a depth-first search on the graph to find a cycle. It takes in the input graph, a vector of boolean values to track visited nodes, a stack to hold the current path, the current node, the source node, and a Boolean flag indicating whether a cycle has been found. The function recursively visits each node and checks if there is a path to the source node. If it

finds a path, it sets the found flag to true and adds the source node to the path. The function also checks if there is a cycle and sets the found flag to true if one is found. It then returns the stack containing the path or an empty stack if no cycle is found.

Algorithm 2 DFS

```
1: function DFS(graph, visited, path, currKid, source, found)
2:     visited[currKid] ← true                          ▷ Mark current kid as visited
3:     path.push(currKid)                               ▷ Push current kid into the path
4:     for i ← 0 to (sizeofgraph[currKid]) − 1 do       ▷ Loop through all neighbors of the curr kid
5:         if found then
6:             return                                   ▷ If path is found than execute the function
7:         end if
8:         if graph[currKid][i] = 1 then                ▷ Check if there is an edge from curr kid to its neighbor
9:             if graph[source][currKid] = graph[currKid][source] and path.size() > 2 and graph[currKid][source] = 1 then   ▷ Check if cycle found
10:                found ← true                         ▷ Set found to true if cycle is found
11:                path.push(source)                    ▷ Add the source kid into the path
12:                return                               ▷ Exit the function
13:            end if
14:            if i = source and path.size() > 2 then   ▷ Check if cycle found and path is correct
15:                found ← true
16:                path.push(source)                    ▷ Add the source kid into the path
17:                return                               ▷ Exit the function
18:            end if
19:            if not visited[i] and not found then     ▷ Check if neighbor is visited and cycle not found
20:                DFS(graph, visited, path, i, source, found)   ▷ Recursively call the dfs function on the neighbor kid
21:            end if
22:        end if
23:    end for
24:    if not found then                                ▷ If cycle not found
25:        path.pop()                                   ▷ Remove the current kid from the path
26:    end if
27: end function
```

The findCycle function uses the dfs function to find a cycle in the graph. It takes in the input graph, the number of children, and the source node as arguments. The function initializes the visited vector, path stack, and found Boolean flag. It then calls the dfs function and returns the stack containing the path or an empty stack if no cycle is found.

Algorithm 3 findCycle

```
1: function FINDCYCLE(graph, numOfKids, source)
2:     visited ← vector of length numOfKids filled with false
3:     path ← empty stack
4:     found ← false
5:     dfs(graph, visited, path, source, source, found)
6:     if found then
7:         path ← reverse_stack(path)
8:         result ← string(path.top())
9:         path.pop()
10:        stepCount ← 1
11:        while path is not empty do
12:            result ← result + " → "+ tostring(path.top())
13:            path.pop()
14:            stepCount ← stepCount + 1
15:        end while
16:        return string(stepCount − 1) + result
17:    end if
18:    return "-1"
19: end function
```

In the main function, the program reads in a graph from a file and stores it in a vector of vectors. It then calls the bfs function to find the shortest path between two nodes and prints the output to the console. The program then calls the findCycle function to find a cycle in the graph and prints the output to the console. Finally, the program writes the graph into graph.txt, bfs into bfs.txt and dfs into dfs.txt files as outputs.

1.2 Show the time complexity of your algorithm

The bfs function uses a queue to perform a breadth-first search on the graph. The worst-case time complexity of a breadth-first search is O(n+m), since each vertex and edge are visited at most once. Therefore, the time complexity of the bfs function is O(n+m).

The dfs function uses a stack to perform a depth-first search on the graph. The worst-case time complexity of a depth-first search is also O(n+m), since each vertex and edge are visited at most once. However, the worst-case space complexity of a depth-first search is O(n), since the maximum depth of the recursive call stack is n. Therefore, the dfs function has a time complexity of O(n+m) and a space complexity of O(n).

The findCycle function calls the dfs function, which has a time complexity of O(n+m). Therefore, the time complexity of the findCycle function is also O(n+m).

2 Why should you maintain a list of discovered nodes? How does this affects the outcome of the algorithms?

Maintaining a list of discovered kids helps to ensure that each kid is visited only once and prevents the algorithm from getting stuck in an infinite loop by revisiting nodes multiple times.

3 How does increasing the number of the kids affects the memory complexity of the algorithms?

3.1 Show the space complexity of your algorithm on the pseudo-code

The space complexity of the algorithm can be calculated by determining how much additional memory the program needs to store the variables and data structures it uses. Here's the breakdown of the space complexity for the given code:

graph: A 2D vector that stores the adjacency matrix of the graph. The size of this vector is numOfKids x numOfKids. Therefore, the space complexity of graph is O(numOfKids^2).

dist and prev: Two 1D vectors that store the distance and predecessor information for the BFS algorithm. The size of these vectors is numOfKids. Therefore, the space complexity of dist and prev is O(numOfKids).

q: A queue data structure used in the BFS algorithm. The size of the queue can be as large as numOfKids. Therefore, the space complexity of q is O(numOfKids).

path: A stack data structure used in the DFS algorithm. The size of the stack can be as large as numOfKids. Therefore, the space complexity of path is O(numOfKids).

visited: A 1D boolean vector that keeps track of visited nodes in the DFS algorithm. The size of this vector is numOfKids. Therefore, the space complexity of visited is O(numOfKids).

stepCount: A global variable that keeps track of the number of steps taken by the ball. This variable does not depend on the input size and therefore does not affect the space complexity.

Therefore, the overall space complexity of the given code is O(numOfKids^2).

3.2 Show run-time of all cases in a graph with number of nodes in x-axis and run-time in y-axis.

Table 1: Run-time of Public Cases

Case	**Input Size**	**Time Elapsed (microseconds)**
1	7	65652
2	10	65652
3	20	50449
6	49	54827
7	191	51882
9	376	94844
11	821	218555
13	1260	456704