CMPE 343
Programming Homework 1

Halil Mert Güler

Section 2

Assignment 1

Question 1

1. Problem Statement and Code Design

The task is to manage undirected flight routes and create a program that finds the lexicographically smallest path which will take the minimum amount of time (in minutes) required to move from city X to city Y. For this assignment graph data structure is used. Also computing the total time passed by flights and airport states which are loading and running.

Sub tasks

- Understanding the task: According to the problem, we should create a java code with using undirected graph structure that takes input from user and finds the smallest path between given two vertices. After that the program should compute the time passed by flights and airport states.

- Choose the best algorithm for the task: We are using an undirected graph, and we want to find the shortest path. According to these requirements, the proper algorithm is Breadth First Search to find the shortest path.

- Create the graph data structure: When creating the graph class, it is important to implement it as an undirected graph because we have undirected routes in the problem. So, when adding the edge between two vertices, we mut add them bidirectional.

- Generating a code part to take input accordingly: The code part that takes input from user should be in String form. After the input line is taken, this string input must split to get digits. Finally, digits will be used in code accordingly.

- Implementation: After we create our road map, we should start the implementation with creating “Main.Java” class that includes runnable function, the graph class and necessary algorithms.

- Testing: At this part we will test our program with the according input and check the result.

2. Implementation and Functionality

For the implementation part, “Main.java” class with runnable function created at first. After that, the undirected graph class was implemented.

```
class Graph {
    private int V;
    private LinkedList<Integer>[] adj;

    public Graph(int V) {
        this.V = V;
        adj = new LinkedList[V];
        for (int i = 0; i < V; i++)
            adj[i] = new LinkedList<Integer>();
    }

    public void addEdge(int u, int v) {
        adj[u].add(v);
        adj[v].add(u);
    }
}
```

The graph class has an integer value V that corresponds to number of cities (vertices), adjacency list to hold adjacency values in form of integer linked list array, constructor and method for adding edges between vertices. This “addEdge” method is adding edges bidirectionally because our graph is undirected.

After that, the code part that takes input and initializes the graph is created.

```
    scan the first line from user and split it with space character.

    n = 1. split value // number of vertices
    m = 2. split value // number of edges
    t = 3. split value // airport state change time
    c = 4. split value // time to travel between two cities

    initialize the graph with vertices count n
    g is new graph with n vertices

    for 0 to m
         scan the line.
         u = split and take first
         v = split and take second
         add a new edge to graph as u-v

    scan the last line.
    start = split and take first
    end = split and take second
```

After input management is done, a modified version of BFS algorithm to find shortest path between two vertices is added to the graph class.

```java
public List<Integer> shortestPath(int start, int end) {

    boolean[] visited = new boolean[V];
    int[] distance = new int[V];
    int[] parent = new int[V];

    Queue<Integer> queue = new LinkedList<Integer>();
    queue.add(start);
    visited[start] = true;

    while (!queue.isEmpty()) {
        int curr = queue.poll();
        if (curr == end)
            break;
        for (int next : adj[curr]) {
            if (!visited[next]) {
                visited[next] = true;
                distance[next] = distance[curr] + 1;
                parent[next] = curr;
                queue.add(next);
            }
        }
    }

    List<Integer> path = new ArrayList<Integer>();
    int curr = end;
    while (curr != start) {
        path.add(curr);
        curr = parent[curr];
    }
    path.add(start);
    Collections.reverse(path);

    return path;
}
```

This function calculates the shortest path between two nodes in a graph using BFS algorithm. It maintains an array of visited nodes, distances from the start node, and parent nodes for each node. It uses a queue to explore neighboring nodes in breadth-first manner. Once the end node is reached, it traces back the path by following parent nodes from end to start. The resulting path is returned as a list of integers in reverse order.

The last part is printing the output after finding the shortest path.

```
        List<Integer> path = graph.shortestPath(start, end);

        System.out.println(path.size());

        for (int node : path) {
            System.out.print((node + 1) + " ");
        }

        System.out.println();
        int totalFlight = (path.size() - 1) * c;
        int totalStateTime = (totalFlight % t) * (path.size() - 2);
        System.out.println(totalFlight + totalStateTime);
```

The “shortesPath” method with parameters “start” and “end” vertices returns a integer list.
First, the size of the path which corresponds to number of cities traveled and after that the cities with using a for-of loop. In the new line program prints the calculated result for total time passed with travel and airport states.
According to the sample input:
t = 3 and c = 5, airports change their state in every 3 minute and travel between two cities takes 5 minutes. So total flight time is c * (city count – 1) because plane won’t travel in start city. After that, total state time can be found by the remainder of the division total flight and t. It’s because when travel is started airport is open state and when the plane reaches the first city the time that plane needs wait is 1 minute, and multiplication is with (city count – 2) because first and last cities is not included this state time. Finally, addition of total flight and state time gives the total time of the travel.

3. Final Assessments

- <u>What were the trouble points in completing this assignment?</u>
  The trouble points were to modify BFS algorithm to include end vertex and finding total time.

- <u>Which parts were the most challenging for you?</u>
  The most challenging part was finding the total time according to flight and state times.

- <u>What did you like about the assignment? What did you learn from it?</u>
  The part that I liked at this assignment was adapting algorithm programming to a real-life problem and finding solutions. Because these kinds of algorithms are used in real life while handling similar situations.

Question 2

1. Problem Statement and Code Design

The task is to organize an island tour from the start island and include desired island and return the beginning point at the end of the tour. But when organizing this tour, we should use the smallest cycle and we should avoid going back from the path that we came on.

Sub tasks

- Understanding the task: According to the problem, we should create a java code using an undirected graph structure that takes input from user and finds the smallest cycle that starts from vertex x and includes the vertex y.

- Choose the best algorithm for the task: We are using an undirected graph, and we want to find the smallest cycle. According to these requirements, the proper algorithm is Depth First Search to find the smallest cycle.

- Create the graph data structure: When creating the graph class, it is important to implement it as an undirected graph because we have undirected routes in the problem. So, when adding the edge between two vertices, we mut add them bidirectional.

- Generating a code part to take input accordingly: The code part that takes input from user should be in String form. After the input line is taken, this string input must split to get digits. Finally, digits will be used in code accordingly.

- Implementation: After we create our road map, we should start the implementation with creating “Main.Java” class that includes runnable function, the graph class and necessary algorithms.

- Testing: At this part we will test our program with the according input and check the result.

2. Implementation and Functionality

For the implementation part, “Main.java” class with runnable function created at first. After that, the undirected graph class was implemented.

```
class Graph {
    private int V;
    private LinkedList<Integer>[] adj;
    private LinkedList<Integer> smallestCycle;

    public Graph(int V) {
        this.V = V;
        adj = new LinkedList[V];
        for (int i = 0; i < V; i++)
            adj[i] = new LinkedList<Integer>();
    }

    public void addEdge(int u, int v) {
        adj[u].add(v);
        adj[v].add(u);
    }
}
```

The graph class has an integer value V that corresponds to number of cities (vertices), adjacency list to hold adjacency values in form of integer linked list array, constructor and method for adding edges between vertices. This “addEdge” method is adding edges bidirectionally because our graph is undirected. Also, the variable “smallestCycle” holds the vertices of the found smallest cycle.

After that, the code part that takes input and initializes the graph is created.

```
    scan the first line from user and split it with space character.

    n = 1. split value // number of vertices
    m = 2. split value // number of edges

    initialize the graph with vertices count n + 1
    g is new graph with n + 1 vertices

    for 0 to m
         scan the line.
         u = split and take first
         v = split and take second
         add a new edge to graph as u-v

    scan the last line.
    x = split and take first
    y = split and take second
```

After input management is done, a modified version of DFS algorithm to find smallest cycle that starts from vertex x and includes vertex y is added to the graph class.

```
    public LinkedList<Integer> findCycles(int x, int y) {
        smallestCycle = null;
        boolean[] visited = new boolean[V];
        LinkedList<Integer> path = new LinkedList<>();
        path.add(x);
        findCyclesUtil(x, y, x, visited, path, 0);
        if (smallestCycle != null) {
            return smallestCycle;
        }

        return null;
    }

    private void findCyclesUtil(int start, int y, int current, boolean[] visited,
LinkedList<Integer> path,
            int cycleSize) {
        visited[current] = true;

        for (Integer v : adj[current]) {
            if (v == start && path.size() >= 3) {
                path.add(v);
                if (path.contains(y)) {
                    if (smallestCycle == null || cycleSize + 1 < smallestCycle.size()) {
                        smallestCycle = new LinkedList<>(path);
                    }
                }
                path.removeLast();
            } else if (!visited[v]) {
                path.add(v);
                findCyclesUtil(start, y, v, visited, path, cycleSize + 1);
                path.removeLast();
            }
        }
        visited[current] = false;
    }
```

The “findCycles” method finds cycles in a graph represented as an adjacency list, starting from node x and ending at node y, and returns the smallest cycle found as linked list of integers. It uses a recursive depth-first search approach and keeps track of visited nodes using a Boolean array. If no cycles are found, it returns null.

“findCyclesUtil” is a recursive utility method used to find cycles in a graph. It starts from the 'start' node and explores adjacent nodes using an adjacency list. It keeps track of visited nodes using a Boolean array. If a cycle of size greater than or equal to 3 is found that ends at node 'y' (other ways, cycle with size less than 3 will use path between only

two vertices because of bidirectional ways of undirected graph structure ex: 1-2-1), it updates the smallest cycle found so far.

The last part is ordering the cycle vertices (islands) in ascending order and printing them.

```
LinkedList<Integer> path = graph.findCycles(x, y);
path.removeLast();

int[] finalPath = new int[path.size()];
for (int i = 0; i < path.size(); i++) {
    finalPath[i] = path.get(i);
}

Arrays.sort(finalPath);

for (int vertex : finalPath) {
    System.out.print(vertex + " ");
}
```

The linked list path holds the value of resulted path list after cycle is found.
The last vertex from the path is removed because algorithm includes start vertex at the end of the path (1-2-4-3-1). After that an integer array is used to copy the vertex values and this array is sorted in ascending order. Lastly, output is printed.

3. Final Assessments

- What were the trouble points in completing this assignment?
  The trouble points were to modify DFS algorithm to include desired vertex, finding smallest cycle and not using path that are already used.

- Which parts were the most challenging for you?
  The most challenging parts were again to modify DFS algorithm to include desired vertex, finding smallest cycle and not using path that are already used.

- What did you like about the assignment? What did you learn from it?
  The part that I liked at this assignment was adapting algorithm programming to a real-life problem and finding solutions. Because these kinds of algorithms are used in real life while handling similar situations.