RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT

Course Code	19CSC311A
Course Name	Graph Theory and Optimization
Programme	B. Tech.
Department	Computer Science and Engineering
Faculty	FET

Name of the Student	Deepak R ;
Reg. No	18ETCS0020041
Semester/Year	6th /2018
Course Leader/s	Mr. Narasimha Murthy K. R.

Declaration Sheet				
Student Name	Deepak R			
Reg. No	18ETCS002041			
Programme	B. Tech.	Semester/Year	6th /2018	
Course Code	19CSC311A			
Course Title	Graph Theory and Optimization			
Course Date		to		
Course Leader	Mr. Narasimha Murthy K. R.			

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Faculty of Engineering and Technology			
Ramaiah University of Applied Sciences			
Department	Computer Science and Engineering	Programme	B. Tech. in CSE
Semester/Batch	06/2018		
Course Code	19CSC311A	Course Title	Graph Theory and Optimization
Course Leader	Ms. Pallavi R. Kumar and Mr. Narasimha Murthy K. R.		

Assignment-1			
Reg. No.	18ETCS002041	**Name of Student**	Deepak R

Sections	Marking Scheme		Marks		
			Max Marks	First Examiner Marks	Moderator
Part A					
	A.1.1	Answers and Justification	06		
	A.1.2	Algorithm	04		
	A.1.3	Code and results	06		
		Part-A Max Marks	16		
Part B					
	B.1.1	Detailed Explanation of Approach	04		
	B.1.2	Algorithm	05		
		Part-B Max Marks	09		
Total Assignment Marks			25		

Course Marks Tabulation				
Component-1 (B) Assignment	**First Examiner**	**Remarks**	**Moderator**	**Remarks**
A				
B				
Marks (out of 25)				

Signature of First Examiner **Signature of Moderator**

Please note:

1. Documental evidence for all the components/parts of the assessment such as the reports, photographs, laboratory exam / tool tests are required to be attached to the assignment report in a proper order.
2. The First Examiner is required to mark the comments in RED ink and the Second Examiner's comments should be in GREEN ink.
3. The marks for all the questions of the assignment have to be written only in the **Component – CET B: Assignment** table.
4. If the variation between the marks awarded by the first examiner and the second examiner lies within +/- 3 marks, then the marks allotted by the first examiner is considered to be final. If the variation is more than +/- 3 marks then both the examiners should resolve the issue in consultation with the Chairman BoE.

Assignment 1

Term - 1

Instructions to students:

1. The assignment consists of **2** questions: Part A-**1** Question, Part B-**1** Question.
2. Maximum marks is **25**.
3. The assignment has to be neatly word processed as per the prescribed format.
4. The maximum number of pages should be restricted to **10**.
5. The printed assignment must be submitted to the course leader.
6. **Submission Date: 2nd June 2021**
7. **Submission after the due date is not permitted.**
8. **IMPORTANT**: It is essential that all the sources used in preparation of the assignment must be suitably referenced in the text.

Part A

Solution for A1.1

Question 1: Can the people of Konigsberg successfully walk over all the bridges once and get back to where they started?

Answer: The answer is **we can't** because the Landmasses are connected by more than one Bridges and due to this there would be at least 1 bridge which gets repeated during the walk, so it is not possible to walk over all the bridges exactly once and get back to the start point.

Question 2: Can the bridge walk be achieved if the people were happy not returning to their starting point?

Answer: Even if people were happy not returning to their starting point, even then the walk is **not Possible** there would be at least 1 bridge that would be repeated.

Question 3: If one or more of the bridges were removed, can the round trip walk around the bridges of Konigsberg be achieved?

Answer: Yes, it is possible, if the seven Bridges reduced to five.

If the current position of the seven bridges shown below, then the walk can be from

A-B, B-D, D-C, C-B, B-A

Question 4: If the city of Konigsberg had seven bridges arranged in some other way, will it be possible to make the round trip walk successfully?

Answer: Yes, put two land masses M and N b/w C&D and then

A, B, C, D, X, Y, B, A will get us fulfill our requirement. then the walk would be possible.

Question 5: If the seven bridges of Konigsberg are replaced with eight, nine and ten bridges, what can be commented about the land masses in each of the cases?

Answer: We can't say something sure; it depends on how we have **arranged our bridges**, sometimes there **is a walk and** sometimes there isn't a walk.

Question 6: To cover n number of bridges is there a generalized result? Is there any relevance of knowing in what way the bridges are connected to the land masses?

Answer: The key point is the **even-ness of the number of bridges at each land masses.** So, if we can somehow prove that each landmass has even no. of Bridges, then there will be guarantee round trip walk.

A.1.2 Algorithm/Pseudocode

Step 1: Start

Step 2: Create A class that represents an undirected graph

```
static class Graph {
```

Step 2: Initialize No. of vertices int V;

Step 3: Create A dynamic array of adjacency lists

```
ArrayList<ArrayList<Integer>> adj;
```

Step 4: Create Constructor

```
Graph(int V) {
    this.V = V;
    adj = new ArrayList<ArrayList<Integer>>();
    for (int i = 0; i < V; i++) {
        adj.add(new ArrayList<Integer>());}}
```

Step 5: Create functions to add and remove edge

```
void addEdge(int u, int v) {
    adj.get(u).add(v);
    adj.get(v).add(u);}
```

Step 6: Create a function to remove edge u-v from graph.it removes the edge by replacing adjacent vertex value with -1.

```
void rmvEdge(int u, int v) {
```

Step 7: Find v in adjacency list of u and replace it with -1

```
        int iv = find(adj.get(u), v);
        adj.get(u).set(iv, -1);
```

Step 8: Find u in adjacency list of v and replace it with -1

```
        int iu = find(adj.get(v), u);
        adj.get(v).set(iu, -1);}
    int find(ArrayList<Integer> al, int v) {
        for (int i = 0; i < al.size(); i++) {
            if (al.get(i) == v) {
                return i; } }
        return -1; }
```

Step 9: Create Methods to print Eulerian tour .The main function that print Eulerian Trail.

It first finds an odd degree vertex (if there is any) and then calls printEulerUtil() to print the path

```
    printEulerTour() {
```

Step 10: Finds a vertex with odd degree

```
        Intitialize  u = 0;
        for (int i = 0; i < V; i++) {
            if (adj.get(i).size() % 2 == 1) {
                u = i;
                break; }}
```

Step 11: Print tour starting from oddv

```
        printEulerUtil(u);}
```

Step 12: Print Euler tour starting from vertex u

```
    printEulerUtil(int u) {
```

Step 13: Recurs for all the vertices adjacent to this vertex

```
        for (int i = 0; i < adj.get(u).size(); ++i) {
```

```
int v = adj.get(u).get(i);
```

Step 14: If edge u-v is not removed and it's a avalid next edge

```
if (v != -1 && isValidNextEdge(u, v)) {
   Display(u + "-" + v + );
   rmvEdge(u, v);
   printEulerUtil(v);}} }
```

Step 15: This function returns count of vertices reachable from v. It does DFS A DFS based function to count reachable vertices from v

```
DFSCount(int v, boolean visited[]) {
```

Step 16: Marks the current node as visited

```
visited[v] = true;
intialize  count = 1;
```

Step 16: Recur for all vertices adjacent to this vertex

```
for (int i = 0; i < adj.get(v).size(); ++i) {
   int u = adj.get(v).get(i);
   if (u != -1 && !visited[u]) {
    Assign   count += DFSCount(u, visited); }}
return count;}
```

Step 17: Create Utility function to check if edge u-v is a valid next edge in Eulerian trail or circuit .The function to check if edge u-v can be considered as next edge in Euler Tout

```
boolean isValidNextEdge(int u, int v) {
```

Step 18:The edge u-v is valid in one of the following two cases:

1) If v is the only adjacent vertex of u

```
int count = 0; // To store count of adjacent vertices
for (int i = 0; i < adj.get(u).size(); ++i) {
   if (adj.get(u).get(i) != -1) {
```

count++;}}

if (count == 1) {

return true; }

2) If there are multiple adjacents, then u-v is not a bridge

Step 19: Do following steps to check if u-v is a bridge

2.a) count of vertices reachable from u

boolean visited[] = new boolean[V];

Arrays.fill(visited, false);

int count1 = DFSCount(u, visited);

2.b) Remove edge (u, v) and after removing the edge, count vertices reachable from u

rmvEdge(u, v);

Arrays.fill(visited, false);

int count2 = DFSCount(u, visited);

2.c) Add the edge back to the graph

addEdge(u, v);

2.d) If count1 is greater, then edge (u, v) is a bridge

return (count1 > count2) ? false : true}}

Step 20: Input the graph to check whether it follows above prescribe rules.

Step 21:Stop

A.1.3 Code and results

```
//Program By Deepak R 18ETCS002041
import java.util.*;
public class GraphAssignment1 {
    static class Graph {
        int V;
        ArrayList<ArrayList<Integer>> adj;
        Graph(int V) {
            this.V = V;
            adj = new ArrayList<ArrayList<Integer>>();
            for (int i = 0; i < V; i++) {
                adj.add(new ArrayList<Integer>());}}
        void addEdge(int u, int v) {
            adj.get(u).add(v);
            adj.get(v).add(u);}
        void rmvEdge(int u, int v) {
            int iv = find(adj.get(u), v);
            adj.get(u).set(iv, -1);
            int iu = find(adj.get(v), u);
            adj.get(v).set(iu, -1);}
        int find(ArrayList<Integer> al, int v) {
            for (int i = 0; i < al.size(); i++) {
                if (al.get(i) == v) {
                    return i;}}
            return -1;}
        void printEulerTour() {
            int u = 0;
            for (int i = 0; i < V; i++) {
                if (adj.get(i).size() % 2 == 1) {
                    u = i;
                    break;}}
            printEulerUtil(u);
            System.out.println();
        }
        void printEulerUtil(int u) {
            for (int i = 0; i < adj.get(u).size(); ++i) {
                int v = adj.get(u).get(i);
                if (v != -1 && isValidNextEdge(u, v)) {
                    System.out.print(u + "-" + v + " ");
                    rmvEdge(u, v);
                    printEulerUtil(v);}}}
        int DFSCount(int v, boolean visited[]) {
            visited[v] = true;
            int count = 1;
            for (int i = 0; i < adj.get(v).size(); ++i) {
                int u = adj.get(v).get(i);
                if (u != -1 && !visited[u]) {
                    count += DFSCount(u, visited);}}
            return count;}
        boolean isValidNextEdge(int u, int v) {
            int count = 0; // To store count of adjac
            for (int i = 0; i < adj.get(u).size(); ++
                if (adj.get(u).get(i) != -1) {
                    count++;}}
            if (count == 1) {
                return true;}
            boolean visited[] = new boolean[V];
            Arrays.fill(visited, false);
            int count1 = DFSCount(u, visited);
            rmvEdge(u, v);
            Arrays.fill(visited, false);
            int count2 = DFSCount(u, visited);
            addEdge(u, v);
            return (count1 > count2) ? false : true;}
    public static void main(String args[]) {
        Graph g1 = new Graph(4);
        g1.addEdge(0, 1);
        g1.addEdge(0, 2);
        g1.addEdge(1, 2);
        g1.addEdge(2, 3);
        g1.printEulerTour();
        Graph g3 = new Graph(4);
        g3.addEdge(0, 1);
        g3.addEdge(1, 0);
        g3.addEdge(0, 2);
        g3.addEdge(2, 0);
        g3.addEdge(2, 3);
        g3.addEdge(3, 1);
        g3.printEulerTour();}}
```

Result

```
run:
2-0 0-1 1-2 2-3
1-0 0-2 2-3 3-1 1-0 0-2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Part B

Solution for B.1.1 Detailed Explanation of Approach

We use Backtracking Algorithm to Solve the Problem. In backtracking algorithms, we try to build a solution one step at a time. If at some step it becomes clear that the current path that we are on cannot lead to a solution we go back to the previous step (backtrack) and choose a different path. Briefly, once we exhaust all our options at a certain step we go back. In Short A backtracking algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output.

1. Like all other Backtracking problems, we solve Sudoku by one-by-one assigning numbers to empty cells.
2. Before assigning a number, we need to confirm that the same number is not present in current row, current column and current 3X3 subgrid.
3. If number is not present in respective row, column or subgrid, we can assign the number, and recursively check if this assignment leads to a solution or not. If the assignment doesn't lead to a solution, then we try next number for current empty cell. And if none of number (1 to 9) lead to solution, we return false.

Solution for B.1.2 Algorithm

Algorithm

Step 1: Start

Step 2 : Define a method called isPresentInCol(), this will take call and num.

Step 3:for each row r in the grid, do

if grid[r, col] = num, then return true.

Step 4:return false otherwise

Step 5: Define a method called isPresentInRow(), this will take row and num.

Step 6:for each column c in the grid, do.

if grid[row, c] = num, then return true

Step 7:return false otherwise

Step 8: Define a method called isPresentInBox() this will take boxStartRow, boxStartCol, num.

Step 9:for each row r in boxStartRow to next 3 rows, do

for each col r in boxStartCol to next 3 columns, do

if grid[r, c] = num, then return true

Step 10:return false otherwise.

Step 11: Define a method called findEmptyPlace(), this will take row and col.

Step 12:for each row r in the grid, do.

for each column c in the grid, do.

if grid[r, c] = 0, then return true.

Step 13:return false.

Step 14: Define a method called isValidPlace(), this will take row, col, num.

Step 15:if isPresentInRow(row, num) and isPresentInCol(col, num) and isPresntInBox(row – row mod 3, col – col mod 3, num) all are false, then return true.

Step 16: Define a method called solveSudoku(), this will take the grid.

Step 17:if no place in the grid is empty, then return true.

Step 18:for number 1 to 9, do.

if isValidPlace(row, col, number), then.

grid[row, col] := number.

if solveSudoku = true, then return true.

grid[row, col] := 0.

Step 19:return false.

Step 20 : Stop

References

1. Class Notes
2. Video lectures