A Mini Project Report

on

Euler & Hamiltonian Path/Cycle

In Subject:

ADUA21202: Data Structure

ES21201AD: Discreate Mathematics

by

Shardul Khandebharad (272026)
Neemeesh Khanzode (272027)
Jyotirmay Khavasi (272028)
Suraj Raskar (272046)

Department of Artificial Intelligence and Data Science

VIIT

2020-2021

ABSTRACT

This project aims in finding the Euler's Path, Euler's Circuit, Hamiltonian Path and Hamiltonian Cycle in a given graph. We have used the various features available in C++ and henceforth have found the required things in a given graph. Data structure concepts like the linked lists and 2-D arrays have been utilized to their utmost and the Euler's Path, Euler's Circuit, Hamiltonian Path and Hamiltonian Cycle.

CONTENTS

ACKNOWLEDGEMENT

We place on record and warmly acknowledge the continuous encouragement, invaluable supervision, timely suggestions, and inspired guidance offered by our guide **Dr. Varsha Jadhav,** mam, Department of Artificial Intelligence and Data Science, at VIIT, Pune in bringing this report to a successful completion. If not for our mam, then we would have not been able to understand the concept of the principle of duality and hence we could not have done this project.

We are also grateful to **HOD Dr. Parikshit Mahalle**, sir, Department of Artificial Intelligence and Data Science, at VIIT, Pune for helping us understand the data structure concepts using which it has been possible for us to implement the binary tree in our project.

Last but not the least we express our sincere thanks to all our friends and our parents who have patiently extended all sorts of help for accomplishing this undertaking.

INTRODUCTION

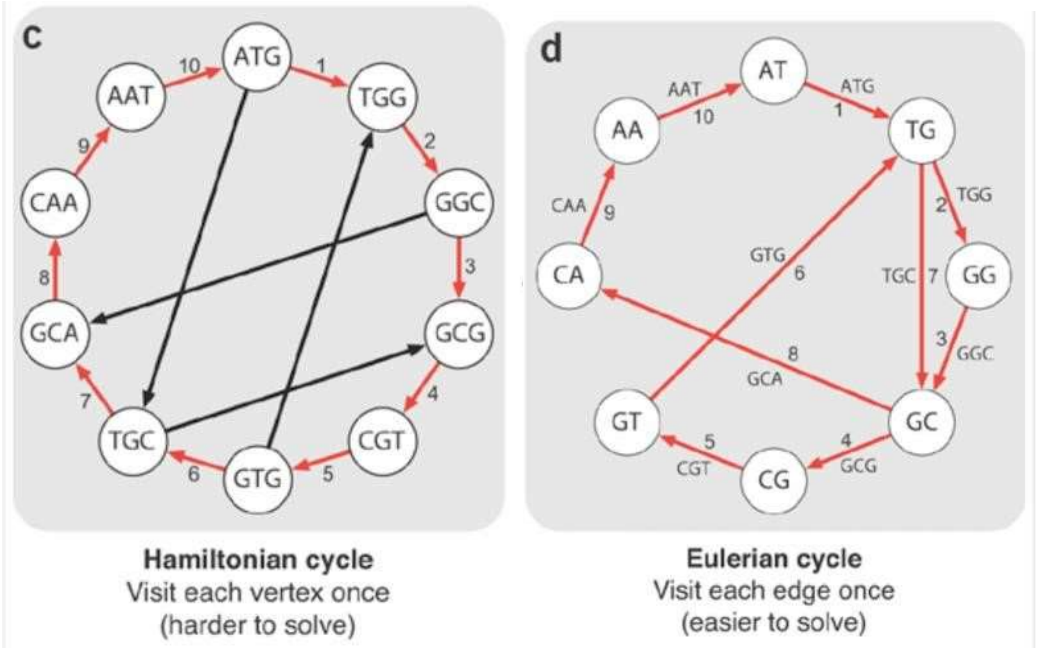
The program aims to obtain Euler's Path, Euler's Circuit, Hamiltonian Path and Hamiltonian Cycle.

An **Euler path** is a path that crosses every edge exactly once without repeating, if it ends at the initial vertex then it is a Euler cycle.

A **Hamiltonian path** passes through each vertex (note not each edge), exactly once, if it ends at the initial vertex then it is a Hamiltonian cycle.

In a Euler path you might pass through a vertex more than once.

In a Hamiltonian path you may not pass through all edges.



Hamiltonian cycle
Visit each vertex once
(harder to solve)

Eulerian cycle
Visit each edge once
(easier to solve)

A **Hamiltonian cycle** is a **cycle** that contains every vertex of the graph hence you may not use all the edges of the graph.

Euler path is a graph using every edge (NOTE) of the graph exactly once.

Euler circuit is a **Euler path** that returns to it starting point after covering all edges

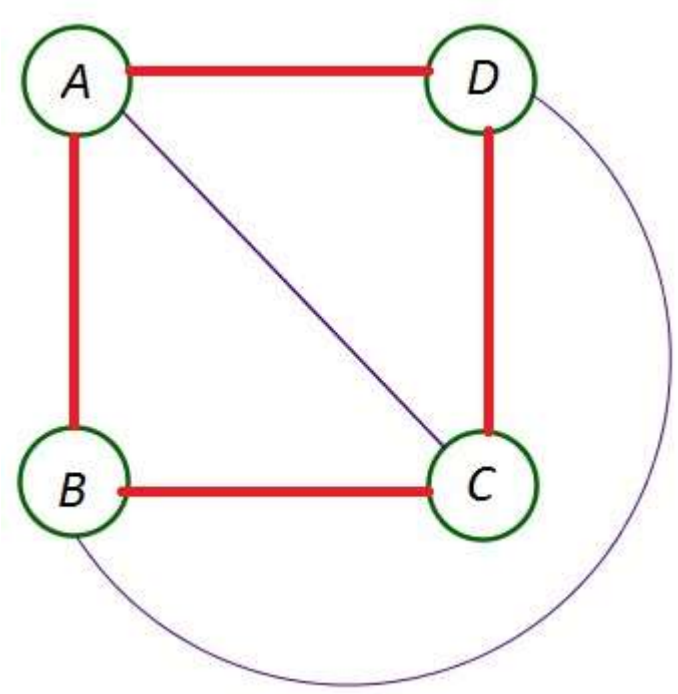
PROBLEM DEFINITION

Hamiltonian Paths and Circuits

Consider visiting each of the vertices in the below given graphs just once. When we do this, we leave the world of Euler and enter the world of Hamilton. Sir William Hamilton. He was an Irish mathematician, physicist, and astronomer.

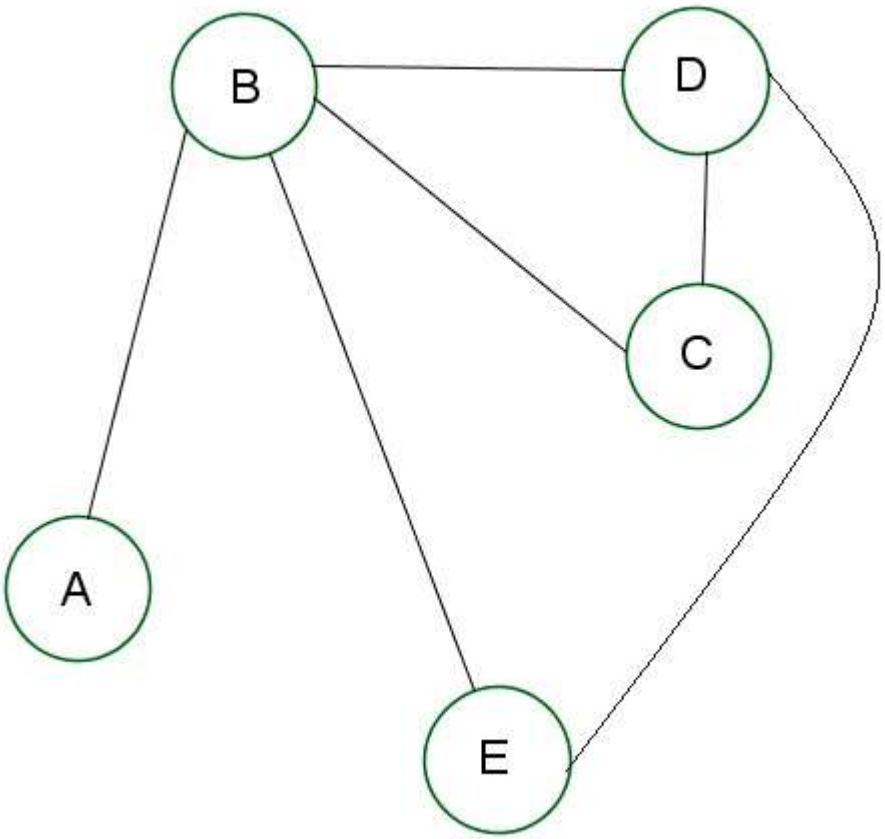
Named after Sir Hamilton, we now look at Hamiltonian paths and Hamiltonian circuits.

A Hamiltonian circuit is a circuit that visits each of the vertices once and only once and ends on the same vertex as it began. For example, in this network the Hamiltonian circuit is marked in red. As it is a circuit, we cannot have repeated edges. We do not need to use all the edges, just visit each vertex once.



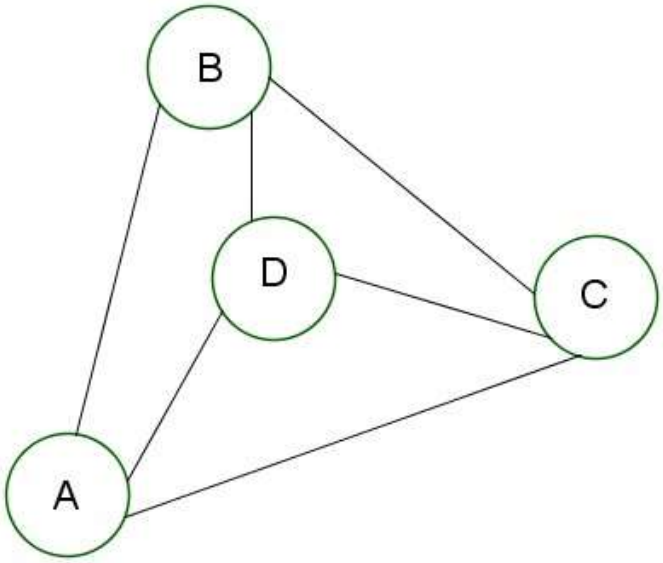
Other Hamiltonian circuits here include ABDCA, $ABDCA$, DBACD, $DBACD$ and many others. If a graph has a Hamiltonian circuit, then it automatically has a Hamiltonian path. (By just dropping off the last vertex in the circuit we create a path, for example, ABDC and DBAC from above)
A Hamiltonian path is a path that visits each of the vertices once and only once but can begin and end on different vertices. For example, the Hamiltonian path in the network here could

be $ABCDE$$ABCDE$. In a Hamiltonian path we do not need to use all the edges, we just have to visit all the vertices once.



Unlike with the Euler paths and Euler circuits, there is no single rule or theorem to help us identify if a Hamiltonian path or Hamiltonian circuit exists. The only thing we can do is look for them.

The following network has both Hamiltonian path and Hamiltonian circuit - can you find them both?



Remember, the Hamiltonian path is a path where we visit each of the vertices only once. There are many Hamiltonian Paths here,

ABCD*ABCD*, ABDC*ABDC*, ADCB*ADCB*, ADBC*ADBC*, ACBD*ACBD*, ACDB*ACDB*, BACD*BACD*, BADC*BADC*, BCDA*BCDA*, BCAD*BCAD*, BDAC*BDAC*, BDCA*BDCA*, CABD*CABD*, CADB*CADB*, CBDA*CBDA*, CBAD*CBAD*, CDBA*CDBA*, CDAB*CDAB*, DABC*DABC*, DACB *DACB*, DBAC*DBAC*, DBCA*DBCA*, DCAB*DCAB* and DCBA*DCBA*. In fact, because each vertex here is connected to every other, then this list is all combinations of the four vertices A, B, C and D.

The Hamiltonian circuit is a circuit where we visit each of the vertices once but return to the beginning node. All the above paths can, in this case, be turned into a circuit by returning to the original node.

If we start at a vertex and trace along edges to get to other vertices, we create a *walk* through the graph. More precisely, a ***walk*** in a graph is a sequence of vertices such that every vertex in the sequence is adjacent to the vertices before and after it in the sequence. If the walk travels along every edge exactly once, then the walk is called an ***Euler path*** (or ***Euler walk***). If, in addition, the starting and ending vertices are the same (so you trace along every edge exactly once and end up where you started), then the walk is called an ***Euler circuit*** (or ***Euler tour***). Of course, if a graph is not connected, there is no hope of finding such a path or circuit. For the rest of this section, assume all the graphs discussed are connected.

The bridges of Königsberg problem is really a question about the existence of Euler paths. There will be a route that crosses every bridge exactly once if and only if the graph below has an Euler path:

This graph is small enough that we could actually check every possible walk that does not reuse edges, and in doing so convince ourselves that there is no Euler path (let alone an Euler circuit). On small graphs which do have an Euler path, it is usually not difficult to find one. Our goal is to find a quick way to check whether a graph has an Euler path or circuit, even if the graph is quite large.

One way to guarantee that a graph does *not* have an Euler circuit is to include a "spike," a vertex of degree 1.

The vertex aa has degree 1, and if you try to make an Euler circuit, you see that you will get stuck at the vertex. It is a dead end. That is, unless you start there. But then there is no way to return, so there is no hope of finding an Euler circuit. There is however an Euler path. It starts at the vertex a,a, then loops around the triangle. You will end at the vertex of degree 3. You run into a similar problem whenever you have a vertex of any odd degree. If you start at such a vertex, you will not be able to end there (after traversing every edge exactly once). After using one edge to leave the starting vertex, you will be left with an even number of edges emanating from the vertex. Half of these could be used for returning to the vertex, the other half for leaving. So you return, then leave. Return, then leave. The only way to use up all the edges is to use the last one by leaving the vertex. On the other hand, if you have a vertex with odd degree that you do not start a path at, then you will eventually get stuck at that vertex. The path will use pairs of edges incident to the vertex to arrive and leave again. Eventually all but one of these edges will be used up, leaving only an edge to arrive by, and none to leave again.

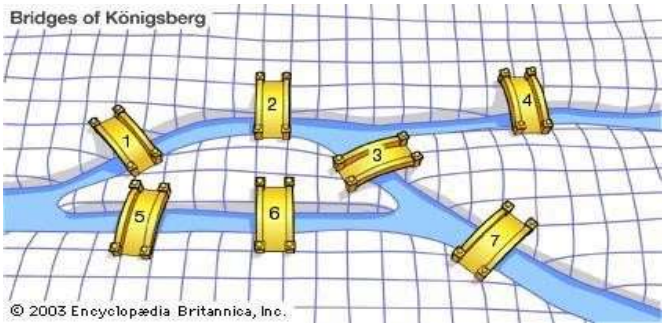
What all this says is that if a graph has an Euler path and two vertices with odd degree, then the Euler path must start at one of the odd degree vertices and end at the other. In such a situation, every other vertex *must* have an even degree since we need an equal number of edges to get to those vertices as to leave them. How could we have an Euler circuit? The graph could not have any odd degree vertex as an Euler path would have to start there or end there, but not both. Thus for a graph to have an Euler circuit, all vertices must have even degree.

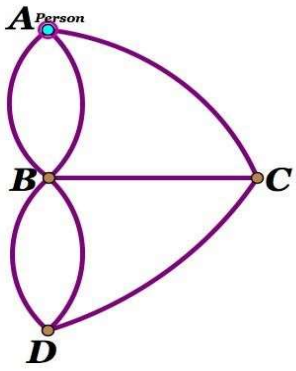
The converse is also true: if all the vertices of a graph have even degree, then the graph has an Euler circuit, and if there are exactly two vertices with odd degree, the graph has an Euler path. To prove this is a little tricky, but the basic idea is that you will never get stuck because there is an "outbound" edge for every "inbound" edge at every vertex. If you try to make an Euler path and miss some edges, you will always be able to "splice in" a circuit using the edges you previously missed.

APPLICATIONS

1. **The Seven bridges of Königsberg**

The Problem of Seven Bridges The year 1736 when Euler solved the problem of seven bridges of Königsberg is taken to mark the birth of graph theory[4]. The seven bridges problem is a well known problem that can be stated as follows: The Pregel river in the town of Königsberg divided it four land regions with a central island. These four land regions were connected by seven bridges as shown in the diagram in Figure 2 where the land areas are denoted by the letters A, B, C, D. It is said that the people of the town used to ponder over the question of whether it is possible to start in a land area and make a walk through the bridges visiting each bridge exactly once and returning to the starting point. In solving this problem of "Seven bridges", Euler developed an abstract approach which is considered to be related to the "geometry of position". Represented in terms of a graph Eg (Figure 2), the four land areas A, B, C, D are the vertices and the seven bridges are the edges of the graph. The vertices A and B are joined by multiple edges (two edges), so that the graph Eg is a multigraph! The "Seven bridges problem" can be expressed in graph-theoretic terms as follows: Is there an Eulerian circuit in the graph Eg? The answer from Euler‟s 1736 paper to this question is NO! . This is stated as an important theorem in the study of Eulerian graphs. Theorem on Eulerian graphs: A connected graph with two or more vertices is an Eulerian graph (ie. has an Eulerian circuit) if and only if each vertex of the graph has even degree. Note that the necessary part of the theorem is based on the fact that, in an Eulerian graph, every time a circuit enters a vertex through an edge it exits the vertex through another edge, thus accounting for an even degree two at the vertex. At the starting vertex, the circuit initially exits and finally enters when the circuit is completed. In the graph Eg in the Seven bridges problem, all the four vertices have odd degree. So, it cannot have an Eulerian circuit





2. The travelling salesman problem (TSP)

The travelling salesman problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.

The travelling purchaser problem and the vehicle routing problem are both generalizations of TSP.

In the theory of computational complexity, the decision version of the TSP (where, given a length L, the task is to decide whether the graph has any tour shorter than L) belongs to the class of NP-complete problems. Thus, it is possible that the worst-case running time for any algorithm for the TSP increases superpolynomially (but no more than exponentially) with the number of cities.

The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact algorithms are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%.[1]

The TSP has several applications even in its purest formulation, such

as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept *city* represents, for example, customers, soldering points, or DNA fragments, and the concept *distance* represents travelling times or cost, or a similarity measure between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources will want to minimize the time spent moving the telescope between the sources. In many applications, additional constraints such as limited resources or time windows may be imposed.

ALGORITHM

Backtracking Algorithm for Hamiltonian cycle

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

Input:
A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
Output:
An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.
For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}. There are more Hamiltonian Cycles in the graph like {0, 3, 4, 2, 1, 0}

```
(0)--(1)--(2)
 |  / \  |
 | /   \ |
 |/     \|
(3)-------(4)
```

And the following graph doesn't contain any Hamiltonian Cycle.

```
(0)--(1)--(2)
 |  / \  |
 | /   \ |
 |/     \|
(3)     (4)
```

Backtracking Algorithm

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

Backtracking works like depth first search. It performs the first search on the problem.
Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.
The repeated nodes are killed off, bonding function is used in backtracking which will kill the similar nodes.
Once we're reached 4, check if there exists a node from 4 to 0, then if the edge is found, print the result.

```
Algorithm Hamiltonian(k)
{
    do{
        NextVertex(k);
        if(x[k]==0){
            return;
        }
        if(k==n){
            print(x[1:n]);
        }
        else{
            Hamiltonian(k+1);
        }
    while(true);
    }
}
```

0,1,2,3,4,0,1,2,3,4. . .

0	1	2	3	4
0	1	2	3	4

0	1	2	3	0
0	1	2	3	4

No Edge

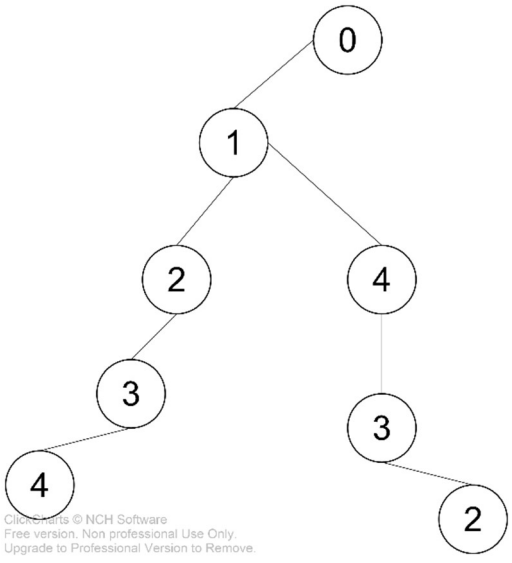
0	1	2	4	0
0	1	2	3	4

0	1	2	0	0
0	1	2	3	4

0	1	3	0	0
0	1	2	3	4

0	1	3	1	0

0	1	3	2	0
0	1	2	3	4

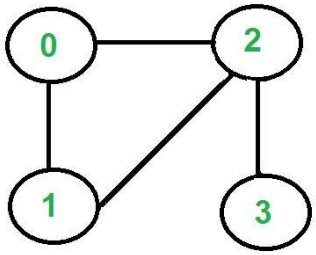


Fleury's Algorithm for printing Eulerian Path or Circuit
Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

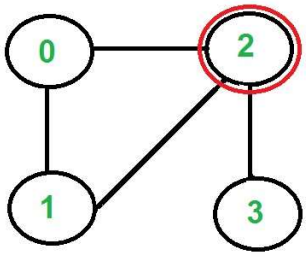
Following is Fleury's Algorithm for printing Eulerian trail or cycle 1.
Make sure the graph has either 0 or 2 odd vertices.

2. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
3. Follow edges one at a time. If you have a choice between a bridge and a non-bridge, *always choose the non-bridge*.
4. Stop when you run out of edges.

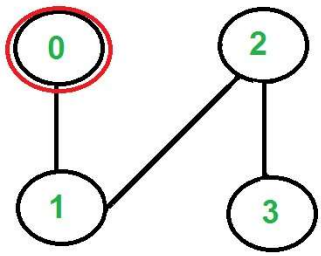
The idea is, *"don't burn bridges"* so that we can come back to a vertex and traverse remaining edges. For example let us consider the following graph.



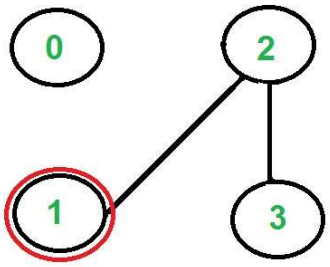
There are two vertices with odd degree, '2' and '3', we can start path from any of them. Let us start tour from vertex '2'.



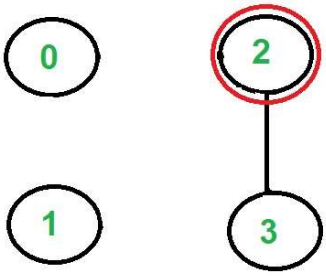
There are three edges going out from vertex '2', which one to pick? We don't pick the edge '2-3' because that is a bridge (we won't be able to come back to '3'). We can pick any of the remaining two edge. Let us say we pick '2-0'. We remove this edge and move to vertex '0'.



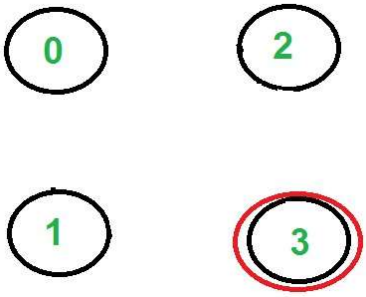
There is only one edge from vertex '0', so we pick it, remove it and move to vertex '1'. Euler tour becomes '2-0 0-1'



There is only one edge from vertex ‘1’, so we pick it, remove it and move to vertex ‘2’. Euler tour becomes ‘2-0 0-1 1-2’



Again, there is only one edge from vertex 2, so we pick it, remove it and move to vertex 3. Euler tour becomes ‘2-0 0-1 1-2 2-3’



There are no more edges left, so we stop here. Final tour is ‘2-0 0-1 1-2 2-3’.

PROGRAM

```
#include <iostream>
#include <string.h>
#include <algorithm>
#include <vector>
#include<stdio.h>
using namespace std;
int n;

class Euler{
  int V;
  vector<int> *adj;
public:
  Euler(int V)  { this->V = V;  adj = new
vector<int>[V];  }//declaring no. of vertices and adjecency
matrix
  ~Euler()      { delete [] adj;  }

  void addEdge(int u, int v) { adj[u].push_back(v);
adj[v].push_back(u); }   //creating and updating adjecency matrix
  void rmvEdge(int u, int v);
  void printEulerTour();
  void printEulerUtil(int s);
  int DFSCount(int v, bool visited[]);
  bool isValidNextEdge(int u, int v);
  int iseuler(int[]);
};

void Euler::printEulerTour(){
  int u = 0;
  for (int i = 0; i < V; i++)
      if (adj[i].size() & 1 )   //taking vertex having odd degree
        {   u = i; break;  }
  printEulerUtil(u);
  cout << endl;
}

void Euler::printEulerUtil(int u){
  vector<int>::iterator i;
  for (i = adj[u].begin(); i != adj[u].end(); ++i){ //traversing
through adjecency matrix
      int v = *i;
      if (v != -1 && isValidNextEdge(u, v)){    // If edge u-v is
not removed and it's a a valid next edge
```

```
            cout << u << "-" << v << " ";
            rmvEdge(u, v);
            printEulerUtil(v);    //recursive function for all
vertices adjacent to this vertex 'u'
        }
    }
}

bool Euler::isValidNextEdge(int u, int v){  //function to check
whether u-v edge should be considered as next edge
    int count = 0;
    vector<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)    //1.to
check whether v is the only adjacent vertex of u
        if (*i != -1)
            count++;
    if (count == 1)
        return true;

    //2.code for checking whether u-v is a bridge
    bool visited[V];
    memset(visited, false, V);
    int count1 = DFSCount(u, visited);  //count of vertices
reachable from u
    rmvEdge(u, v);
    memset(visited, false, V);
    int count2 = DFSCount(u, visited);  //count vertices
reachable from u after removing u-v edge
    addEdge(u, v);
    return (count1 > count2)? false: true;  //if count1>2 => u-v
is a bridge if not then we can proceed with is u-v edge
}

void Euler::rmvEdge(int u, int v){
  vector<int>::iterator iv = find(adj[u].begin(), adj[u].end(),
v);
  *iv = -1;

  vector<int>::iterator iu = find(adj[v].begin(), adj[v].end(),
u);
  *iu = -1;
}

int Euler::DFSCount(int v, bool visited[]){     //finding count
of vertices reachable using DFS algo
```

```
  visited[v] = true;
  int count = 1;
  vector<int>::iterator i;
  for (i = adj[v].begin(); i != adj[v].end(); ++i)  // Recur for
all vertices adjacent to this vertex
        if (*i != -1 && !visited[*i])
            count += DFSCount(*i, visited);
  return count;
}

int Euler::iseuler(int a[]){    //function for checking whether
Euler path/circuit is possible
    int count = 0;
    for(int i=0;i<n;i++){
        if(a[i] % 2 != 0)
            count++;
    }
    if(count == 0){
        cout<<"Euler's circuit exists:\n";
        return 1;
    }
    else if(count == 2){
        cout<<"Euler's path exists:\n";
        return 1;
    }
    else{
        cout<<"No euler's path or circuit exist:'\n\n";
        return -1;
    }
}

class Hamiltonian{
    int *path,v,x;
    public:
        Hamiltonian(int n){ v = 0; x = 0; path = new int[n]; }
        bool isSafe(bool graph[][10], int pos);
        bool hamCycleUtil(bool graph[][10], int pos, int i);
        bool hamCycle(bool graph[][10]);
        void printSolution();
};

bool Hamiltonian::isSafe(bool graph[][10], int pos){
    if (graph [ path[pos-1] ][ v ] == 0)  //check whether edege
is present in between the previous vertex and new vertex to be
added in the path
```

```
        return false;

     for (int i = 0; i < pos; i++)
        if (path[i] == v)       //check whether the new vertex to
be added is already present in the path
            return false;
    return true;
}

bool Hamiltonian::hamCycleUtil(bool graph[][10], int pos, int i){
    if (pos == n){      //checks whether it's last element of
path/cycle(end of path/cycle)
        if ( graph[ path[pos-1] ][ path[0] ] == 1 ){    //check
whether last element of path is same as 1st
            cout<<"Hamiltonian cycle exists\n";
            x = 1;
        }
        else
            cout<<"Hamiltonian path exists\n";
        return true;
    }
    // Try different vertices as a next candidate in Hamiltonian
Cycle.
    for (v = 0; v < n; v++){
        if(v == i)
            continue;
        if (isSafe(graph, pos)){       //checks whether element
to be added is valid or not
            path[pos] = v;
            if (hamCycleUtil (graph, pos+1, i) == true)     //
recur to construct rest of the path
                return true;
            path[pos] = -1;
        }
    }
    return false;
}

bool Hamiltonian::hamCycle(bool graph[][10]){
    for (int i = 0; i < n; i++) //setting initial path with -1
        path[i] = -1;
    for(int i=0;i < n; i++){
        path[0] = i;    //setting 0th position of path as i
```

```
        if ( hamCycleUtil(graph,1,i) == true ){ //initial call to
hamCycleUtil with current position as 1 and check it whether it
returns true
            printSolution();
            return true;
        }
    }
    cout<<"\nNo Hamiltonian path or cycle exist";
    return false;

}

void Hamiltonian::printSolution()
{
    for (int i = 0; i < n - 1; i++)
        cout << path[i] << "-" << path[i + 1] << " ";  //print
path

    if(x == 1)
        cout << path[n - 1] << "-" << path[0];   //print cycle
    cout<<"\n";
}

int main(){
    int i,j,v1,v2,Edges;
    cout<<"\n Enter the number of vertices of graph: ";
    cin>>n;
    Euler g1(n);
    Hamiltonian g2(n);
    int a[n] = {0};         //array for degrees
    bool graph1[10][10];    //adjacency matrix
    for(i=0;i<n;i++)        //initializing adjacency matrix with
0.
        for(j=0;j<n;j++)
            graph1[i][j]=0;
    cout<<"\n Enter the total number of edges: ";
    cin>>Edges;
    for(i=1;i<=Edges;i++)   {
        cout<<"\n Enter the edge: ";
        cin>>v1>>v2;
        graph1[v1][v2]=1;   //setting adjecency matrix (used in
euler)
        graph1[v2][v1]=1;   //setting adjecency matrix (used in
euler)
```

```
        g1.addEdge(v1, v2);     //adding edge for euler path
implementation
        a[v1]++;    //incrementing degree (used for euler)
        a[v2]++;    //incrementing degree (used for euler)
    }
    cout<<"\n";
    g2.hamCycle(graph1);    //call to find hamiltonian path
    cout<<"\n";
    if(g1.iseuler(a) == 1)      //checking if euler path can be
found or not using degrees' logic
        g1.printEulerTour();
  return 0;
}
```

OUTPUT

```
Enter the number of vertices of graph: 7

Enter the total number of edges: 11

Enter the edge: 0 1

Enter the edge: 0 3

Enter the edge: 0 5

Enter the edge: 0 6

Enter the edge: 1 2

Enter the edge: 1 3

Enter the edge: 2 3

Enter the edge: 2 4

Enter the edge: 2 5

Enter the edge: 4 5

Enter the edge: 5 6

Hamiltonian path exists
1-0 0-3 3-2 2-4 4-5 5-6
```

```
Enter the number of vertices of graph: 7

Enter the total number of edges: 12

Enter the edge: 0 1

Enter the edge: 0 2

Enter the edge: 0 3

Enter the edge: 0 6

Enter the edge: 1 6

Enter the edge: 2 3

Enter the edge: 2 5

Enter the edge: 2 6

Enter the edge: 3 4

Enter the edge: 3 5

Enter the edge: 4 5

Enter the edge: 5 6

Hamiltonian path exists
0-1 1-6 6-2 2-3 3-4 4-5

Euler's circuit exists:
0-1 1-6 6-0 0-2 2-3 3-4 4-5 5-2 2-6 6-5 5-3 3-0 5-0
```

CONCLUSION:

The Euler's Path, Euler's Circuit, Hamiltonian Path and Hamiltonian Cycle have been successfully implemented in the program.

REFERENCES:

Ralph P. Grimaldi. (2011). “Discrete and combinatorial mathematics – 5th edition”, Pearson Education (2011)

Kenneth H. Rosen. (1999). “Discrete Mathematics and Its Applications”, McGraw Hill Education; 4th Revised edition (1 January 1999)

Jean-Paul Tremblay, R. Manohar. (2017). “DISCRETE MATHEMATICAL STRUCTURES

WITH APPLICATIONS TO COMPUTER SCIENCE”. McGraw Hill Education; 1 edition (1

July 2017)

V.K. Balakrishnan. (2000). “Introductory Discrete Mathematics (Dover Books on Computer Science)”. Dover Publications Inc.; New edition edition (1 February 2000)

László Lovász, Jozsef Pelikan, Katalin Vesztergombi. (2003). “Discrete Mathematics: Elementary and Beyond (Undergraduate Texts in Mathematics)”. Springer; 2003 edition (January 27, 2003)

C.L. Liu. (2012). “Elements of Discrete Mathematics 4th Edition (English, Paperback, C. L.

Liu)”. McGraw Hill Education https://en.wikipedia.org/wiki/Discrete_mathematics