DAA Project:

Eulerian Cycle

Made By:
Shourojit Dutt

AIM : To make a program that takes a graph as input and outputs a Eulerian Path or Eulerian Cycle or state that is impossible.

Theory:

Eulerian Cycle : A Eulerian path that starts and ends at the same vertex.
An undirected graph has Eulerian cycle if the following two conditions are true :

(i) All vertices with non-zero degree are connected.
(ii) All vertices have even degree.

Eulerian Path : A path in the graph that visits every edge exactly once.
An undirected graph has Eulerian path if the following two conditions are true :

(i) All vertices with non-zero degree are connected.
(ii) If zero or two vertices have odd degree and all other vertices have even degree.

Eulerian Graph : A graph is called Eulerian if it has a Eulerian cycle and called semi-eulerian if it has a Eulerian path. We can find out whether a graph is Eulerian or not in polynomial time of O(V+E).

ALGORITHM USED:

The algorithm used in this project is as follows :

```
IMPOSSIBLE_EUL(Graph)
{
    current_vertex_index=0;
    start_node = Graph.vertices[current_vertex_index];
    num_vertices = Graph.no_of_vertices;
    counter = 1;  // Counter is 1 as vertex 0 is start_node.
    while(counter < num_vertices)
        {
            if(counter == current_vertex_index)
            {
                counter += 1;      // Avoiding self-traversal.
            }
            next_vertex = Graph.vertices[counter];
            if start_node.isConnectedTo(next_vertex)
            {
                // If it's connected, then we're
                // Travelling to and back from next_vertex.
                traverse_to_vertex(next_vertex);
                traverse_to_vertex(start_node);
            }
            else
            {
                // If it isn't connected, then we check the
                // Connected vertices to find a path.
                for vertex in start_node.connected_vertices
                {
                    traverse_to_vertex(vertex);
                    if(vertex.isConnectedTo(next_vertex)
                    {
                        traverse_to_vertex(next_vertex);
                        traverse_to_vertex(vertex);
                        break;
                    }
                    else
                    {
                        traverse_to_vertex(start_node);
                    }
                }
                // Heading back to start_node once path is found.
                traverse_to_vertex(start_node);
```

```
                    }

                    // If current start_node has visited all other nodes
                    if(counter == num_vertices-1 and
current_vertex_index != num_vertices - 1) // Moving to next vertex
                    {
                            current_vertex_index += 1;
                            start_node =
Graph.vertices[current_vertex_index];
                            counter = 0;
                    }
                    else // Remaining at same vertex
                    {
                            counter = counter + 1;
                    }
            }

        // Now we will travel back to starting node from last node.
        // Keep in mind that at this point, start_node is pointing to
        // the last node of the Graph.vertices array.
        if(start_node.isConnectedTo(Graph.vertices[0]))
        {
                // If it is directly connected, we travel back no problem.
                traverse_to_vertex(Graph.vertices[0]);
        }
        else
        {
                // If it isn't then we scan connected vertices to find a path.
                for vertex in start_node.connected_vertices
                {
                        traverse_to_vertex(vertex);
                        if(vertex.isConnectedTo(Graph.vertices[0]))
                        {
                                // Found path, then traverse to 1st vertex & break
                                traverse_to_vertex(Graph.vertices[0]);
                                break;
                        }
                        else
                        {
                                // Traverse back to original node.
                                traverse_to_vertex(start_node);
                        }
```

```
            }
        }

    }
```