Euler Circuit in a Directed Graph

1. All vertices with nonzero degree belong to a single [strongly connected component](http://www.geeksforgeeks.org/strongly-connected-components/).  
   2) In degree and out degree of every vertex is same.

O(V + E)

#include<iostream>

#include <list>

#define CHARS 26

using namespace std;

class Graph

{

    int V;    // No. of vertices

    list<int> \*adj;    // A dynamic array of adjacency lists

    int \*in;

public:

    // Constructor and destructor

    Graph(int V);

    ~Graph()   { delete [] adj; delete [] in; }

    void addEdge(int v, int w) { adj[v].push\_back(w);  (in[w])++; }

    bool isEulerianCycle();

    bool isSC();

    void DFSUtil(int v, bool visited[]);

    Graph getTranspose();

};

Graph::Graph(int V)

{

    this->V = V;

    adj = new list<int>[V];

    in = new int[V];

    for (int i = 0; i < V; i++)

       in[i] = 0;

}

bool Graph::isEulerianCycle()

{

    if (isSC() == false)

        return false;

    for (int i = 0; i < V; i++)

        if (adj[i].size() != in[i])

            return false;

    return true;

}

void Graph::DFSUtil(int v, bool visited[])

{

    visited[v] = true;

    list<int>::iterator i;

    for (i = adj[v].begin(); i != adj[v].end(); ++i)

        if (!visited[\*i])

            DFSUtil(\*i, visited);

}

Graph Graph::getTranspose()

{

    Graph g(V);

    for (int v = 0; v < V; v++)

    {

        list<int>::iterator i;

        for(i = adj[v].begin(); i != adj[v].end(); ++i)

        {

            g.adj[\*i].push\_back(v);

            (g.in[v])++;

        }

    }

    return g;

}

bool Graph::isSC()

{

    bool visited[V];

    for (int i = 0; i < V; i++)

        visited[i] = false;

    int n;

    for (n = 0; n < V; n++)

        if (adj[n].size() > 0)

          break;

    DFSUtil(n, visited);

    for (int i = 0; i < V; i++)

        if (adj[i].size() > 0 && visited[i] == false)

              return false;

    Graph gr = getTranspose();

    for (int i = 0; i < V; i++)

        visited[i] = false;

    gr.DFSUtil(n, visited);

    for (int i = 0; i < V; i++)

        if (adj[i].size() > 0 && visited[i] == false)

             return false;

    return true;

}

int main()

{

    // Create a graph given in the above diagram

    Graph g(5);

    g.addEdge(1, 0);

    g.addEdge(0, 2);

    g.addEdge(2, 1);

    g.addEdge(0, 3);

    g.addEdge(3, 4);

    g.addEdge(4, 0);

    if (g.isEulerianCycle())

       cout << "Given directed graph is eulerian n";

    else

       cout << "Given directed graph is NOT eulerian n";

    return 0;

}

------------------------------------------------------------

# Eulerian path and circuit for undirected graph

**Eulerian Cycle**

….a) All vertices with non-zero degree are connected. We don’t care about vertices with zero degree because they don’t belong to Eulerian Cycle or Path (we only consider all edges).  
….b) All vertices have even degree.

**Eulerian Path**

….a) Same as condition (a) for Eulerian Cycle  
….b) If zero or two vertices have odd degree and all other vertices have even degree.

O(V+E)

#include<iostream>

#include <list>

using namespace std;

class Graph

{

    int V;    // No. of vertices

    list<int> \*adj;    // A dynamic array of adjacency lists

public:

    Graph(int V)   {this->V = V; adj = new list<int>[V]; }

    ~Graph() { delete [] adj; } // To avoid memory leak

    void addEdge(int v, int w);

    int isEulerian();

    bool isConnected();

    void DFSUtil(int v, bool visited[]);

};

void Graph::addEdge(int v, int w)

{

    adj[v].push\_back(w);

    adj[w].push\_back(v);  // Note: the graph is undirected

}

void Graph::DFSUtil(int v, bool visited[])

{

    visited[v] = true;

    list<int>::iterator i;

    for (i = adj[v].begin(); i != adj[v].end(); ++i)

        if (!visited[\*i])

            DFSUtil(\*i, visited);

}

bool Graph::isConnected()

{

    bool visited[V];

    int i;

    for (i = 0; i < V; i++)

        visited[i] = false;

    for (i = 0; i < V; i++)

        if (adj[i].size() != 0)

            break;

    if (i == V)

        return true;

    DFSUtil(i, visited);

    for (i = 0; i < V; i++)

       if (visited[i] == false && adj[i].size() > 0)

            return false;

    return true;

}

int Graph::isEulerian()

{

    if (isConnected() == false)

        return 0;

    int odd = 0;

    for (int i = 0; i < V; i++)

        if (adj[i].size() & 1)

            odd++;

    if (odd > 2)

        return 0;

    return (odd)? 1 : 2;

}

void test(Graph &g)

{

    int res = g.isEulerian();

    if (res == 0)

        cout << "graph is not Eulerian\n";

    else if (res == 1)

        cout << "graph has a Euler path\n";

    else

        cout << "graph has a Euler cycle\n";

}

int main()

{

    Graph g1(5);

    g1.addEdge(1, 0);

    g1.addEdge(0, 2);

    g1.addEdge(2, 1);

    g1.addEdge(0, 3);

    g1.addEdge(3, 4);

    test(g1);

    Graph g2(5);

    g2.addEdge(1, 0);

    g2.addEdge(0, 2);

    g2.addEdge(2, 1);

    g2.addEdge(0, 3);

    g2.addEdge(3, 4);

    g2.addEdge(4, 0);

    test(g2);

    Graph g3(5);

    g3.addEdge(1, 0);

    g3.addEdge(0, 2);

    g3.addEdge(2, 1);

    g3.addEdge(0, 3);

    g3.addEdge(3, 4);

    g3.addEdge(1, 3);

    test(g3);

    Graph g4(3);

    g4.addEdge(0, 1);

    g4.addEdge(1, 2);

    g4.addEdge(2, 0);

    test(g4);

    Graph g5(3);

    test(g5);

    return 0;

}

---------------------------------------------------------------

Hierholzer’s Algorithm for directed graph print

http://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/

O(V+E).

void printCircuit(vector< vector<int> > adj)

{

    unordered\_map<int,int> edge\_count;

    for (int i=0; i<adj.size(); i++)

    {

        //find the count of edges to keep track

        //of unused edges

        edge\_count[i] = adj[i].size();

    }

    if (!adj.size())

        return; //empty graph

    stack<int> curr\_path;

    vector<int> circuit;

    curr\_path.push(0);

    int curr\_v = 0; // Current vertex

    while (!curr\_path.empty())

    {

        // If there's remaining edge

        if (edge\_count[curr\_v])

        {

            curr\_path.push(curr\_v);

            int next\_v = adj[curr\_v].back();

            edge\_count[curr\_v]--;

            adj[curr\_v].pop\_back();

            curr\_v = next\_v;

        }

        else

        {

            circuit.push\_back(curr\_v);

            curr\_v = curr\_path.top();

            curr\_path.pop();

        }

    }

    for (int i=circuit.size()-1; i>=0; i--)

    {

        cout << circuit[i];

        if (i)

           cout<<" -> ";

    }

}

int main()

{

    vector< vector<int> > adj1, adj2;

    // Input Graph 1

    adj1.resize(3);

    // Build the edges

    adj1[0].push\_back(1);

    adj1[1].push\_back(2);

    adj1[2].push\_back(0);

    printCircuit(adj1);

    cout << endl;

    // Input Graph 2

    adj2.resize(7);

    adj2[0].push\_back(1);

    adj2[0].push\_back(6);

    adj2[1].push\_back(2);

    adj2[2].push\_back(0);

    adj2[2].push\_back(3);

    adj2[3].push\_back(4);

    adj2[4].push\_back(2);

    adj2[4].push\_back(5);

    adj2[5].push\_back(0);

    adj2[6].push\_back(4);

    printCircuit(adj2);

    return 0;

}

-----------------------------------------------------------------------------------------------------------

# Fleury’s Algorithm for printing Eulerian Path or Circuit

O((V+E)\*(V+E)) which can be written as O(E2) for a connected graph.

**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***](http://www.geeksforgeeks.org/bridge-in-a-graph/)**“**

#include <iostream>

#include <string.h>

#include <algorithm>

#include <list>

using namespace std;

class Graph

{

  int V;    // No. of vertices

  list<int> \*adj;    // A dynamic array of adjacency lists

public:

  Graph(int V)  { this->V = V;  adj = new list<int>[V];  }

  ~Graph()      { delete [] adj;  }

  void addEdge(int u, int v) {  adj[u].push\_back(v); adj[v].push\_back(u); }

  void rmvEdge(int u, int v);

  void printEulerTour();

  void printEulerUtil(int s);

  int DFSCount(int v, bool visited[]);

  bool isValidNextEdge(int u, int v);

};

void Graph::printEulerTour()

{

  int u = 0;

  for (int i = 0; i < V; i++)

      if (adj[i].size() & 1)

        {   u = i; break;  }

  printEulerUtil(u);

  cout << endl;

}

void Graph::printEulerUtil(int u)

{

  list<int>::iterator i;

  for (i = adj[u].begin(); i != adj[u].end(); ++i)

  {

      int v = \*i;

      if (v != -1 && isValidNextEdge(u, v))

      {

          cout << u << "-" << v << "  ";

          rmvEdge(u, v);

          printEulerUtil(v);

      }

  }

}

bool Graph::isValidNextEdge(int u, int v)

{

  int count = 0;  // To store count of adjacent vertices

  list<int>::iterator i;

  for (i = adj[u].begin(); i != adj[u].end(); ++i)

     if (\*i != -1)

        count++;

  if (count == 1)

    return true;

  bool visited[V];

  memset(visited, false, V);

  int count1 = DFSCount(u, visited);

  rmvEdge(u, v);

  memset(visited, false, V);

  int count2 = DFSCount(u, visited);

  addEdge(u, v);

  return (count1 > count2)? false: true;

}

void Graph::rmvEdge(int u, int v)

{

  list<int>::iterator iv = find(adj[u].begin(), adj[u].end(), v);

  \*iv = -1;

  list<int>::iterator iu = find(adj[v].begin(), adj[v].end(), u);

  \*iu = -1;

}

int Graph::DFSCount(int v, bool visited[])

{

  visited[v] = true;

  int count = 1;

  list<int>::iterator i;

  for (i = adj[v].begin(); i != adj[v].end(); ++i)

      if (\*i != -1 && !visited[\*i])

          count += DFSCount(\*i, visited);

  return count;

}

int main()

{

  Graph g1(4);

  g1.addEdge(0, 1);

  g1.addEdge(0, 2);

  g1.addEdge(1, 2);

  g1.addEdge(2, 3);

  g1.printEulerTour();

  Graph g2(3);

  g2.addEdge(0, 1);

  g2.addEdge(1, 2);

  g2.addEdge(2, 0);

  g2.printEulerTour();

  Graph g3(5);

  g3.addEdge(1, 0);

  g3.addEdge(0, 2);

  g3.addEdge(2, 1);

  g3.addEdge(0, 3);

  g3.addEdge(3, 4);

  g3.addEdge(3, 2);

  g3.addEdge(3, 1);

  g3.addEdge(2, 4);

  g3.printEulerTour();

  return 0;

}