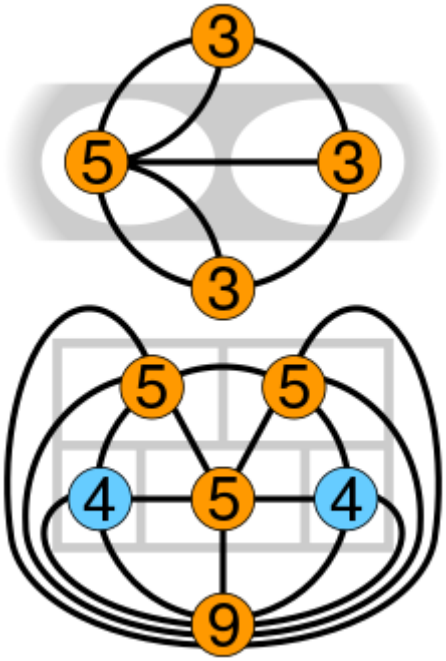
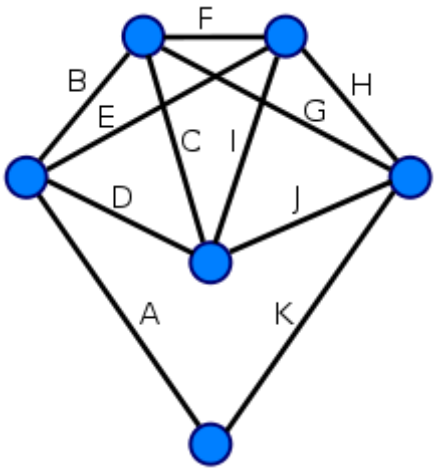
Eulerian path

In graph theory, an **Eulerian trail** (or **Eulerian path**) is a trail in a finite graph that visits every edge exactly once (allowing for revisiting vertices). Similarly, an **Eulerian circuit** or **Eulerian cycle** is an Eulerian trail that starts and ends on the same vertex. They were first discussed by Leonhard Euler while solving the famous Seven Bridges of Königsberg problem in 1736. The problem can be stated mathematically like this:



Multigraphs of both Königsberg Bridges and Five room puzzles have more than two odd vertices (in orange), thus are not Eulerian and hence the puzzles have no solutions.



Every vertex of this graph has an even degree. Therefore, this is an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

> Given the graph in the image, is it possible to construct a path (or a cycle; i.e., a path starting and ending on the same vertex) that visits each edge exactly once?

Euler proved that a necessary condition for the existence of Eulerian circuits is that all vertices in the graph have an even degree, and stated without proof that connected graphs with all vertices of even degree have an Eulerian circuit. The first complete proof of this latter claim was published posthumously in 1873 by Carl Hierholzer.[1] This is known as **Euler's Theorem:**

> A connected graph has an Euler cycle if and only if every vertex has even degree.

The term **Eulerian graph** has two common meanings in graph theory. One meaning is a graph with an Eulerian circuit, and the other is a graph with every vertex of even degree. These definitions coincide for connected graphs.[2]

For the existence of Eulerian trails it is necessary that zero or two vertices have an odd degree; this means the Königsberg graph is *not* Eulerian. If there are no vertices of odd degree, all Eulerian trails are circuits. If there are exactly two vertices of odd degree, all Eulerian trails start at one of them and end at the other. A graph that has an Eulerian trail but not an Eulerian circuit is called **semi-Eulerian**.

Definition

An **Eulerian trail**,[3] or **Euler walk**, in an undirected graph is a walk that uses each edge exactly once. If such a walk exists, the graph is called **traversable** or **semi-eulerian**.[4]

An **Eulerian cycle**,[3] also called an **Eulerian circuit** or **Euler tour**, in an undirected graph is a cycle that uses each edge exactly once. If such a cycle exists, the graph is called **Eulerian** or **unicursal**.[5] The term "Eulerian graph" is also sometimes used in a weaker sense to denote a graph where every vertex has even degree. For finite connected graphs the two definitions are equivalent, while a possibly unconnected graph is Eulerian in the weaker sense if and only if each connected component has an Eulerian cycle.

For directed graphs, "path" has to be replaced with *directed path* and "cycle" with *directed cycle*.

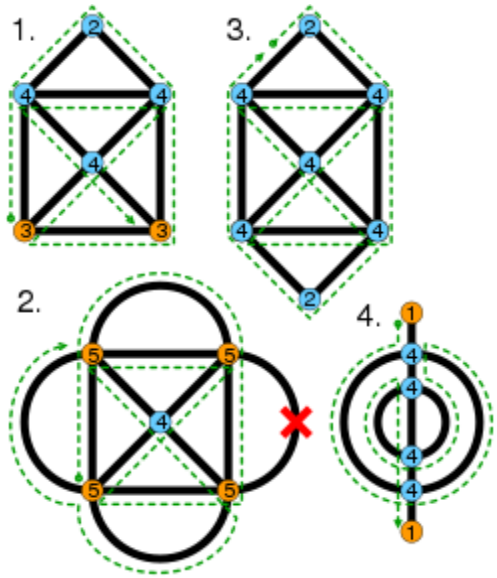
The definition and properties of Eulerian trails, cycles and graphs are valid for multigraphs as well.

An **Eulerian orientation** of an undirected graph G is an assignment of a direction to each edge of G such that, at each vertex v, the indegree of v equals the outdegree of v. Such an orientation exists for any undirected graph in which every vertex has even degree, and may be found by constructing an Euler tour in each connected component of G and then orienting the edges according to the tour.[6] Every Eulerian orientation of a connected graph is a strong orientation, an orientation that makes the resulting directed graph strongly connected.

Properties

- An undirected graph has an Eulerian cycle if and only if every vertex has even degree, and all of its vertices with nonzero degree belong to a single connected component.
- An undirected graph can be decomposed into edge-disjoint cycles if and only if all of its vertices have even degree. So, a graph has an Eulerian cycle if and only if it can be decomposed into edge-disjoint cycles and its nonzero-degree vertices belong to a single connected component.
- An undirected graph has an Eulerian trail if and only if exactly zero or two vertices have odd degree, and all of its vertices with nonzero degree belong to a single connected component
- A directed graph has an Eulerian cycle if and only if every vertex has equal in degree and out degree, and all of its vertices with nonzero degree belong to a single strongly connected component. Equivalently, a directed graph has an Eulerian cycle if and only if it can be decomposed into edge-disjoint directed cycles and all of its vertices with nonzero degree belong to a single strongly connected component.
- A directed graph has an Eulerian trail if and only if at most one vertex has (out-degree) − (in-degree) = 1, at most one vertex has (in-degree) − (out-degree) = 1, every other vertex has equal in-degree and out-degree, and all of its vertices with nonzero degree belong to a single connected component of the underlying undirected graph.[*citation needed*]

Constructing Eulerian trails and circuits



Using Eulerian trails to solve puzzles involving drawing a shape with a continuous stroke:

1. As the Haus vom Nikolaus puzzle has two odd vertices (orange), the trail must start at one and end at the other.
2. Annie Pope's with four odd vertices has no solution.
3. If there are no odd vertices, the trail can start anywhere and forms an Eulerian cycle.
4. Loose ends are considered vertices of degree 1.

Fleury's algorithm

Fleury's algorithm is an elegant but inefficient algorithm that dates to 1883.[7] Consider a graph known to have all edges in the same component and at most two vertices of odd

degree. The algorithm starts at a vertex of odd degree, or, if the graph has none, it starts with an arbitrarily chosen vertex. At each step it chooses the next edge in the path to be one whose deletion would not disconnect the graph, unless there is no such edge, in which case it picks the remaining edge left at the current vertex. It then moves to the other endpoint of that edge and deletes the edge. At the end of the algorithm there are no edges left, and the sequence from which the edges were chosen forms an Eulerian cycle if the graph has no vertices of odd degree, or an Eulerian trail if there are exactly two vertices of odd degree.

While the *graph traversal* in Fleury's algorithm is linear in the number of edges, i.e. , we also need to factor in the complexity of detecting bridges. If we are to re-run Tarjan's linear time bridge-finding algorithm[8] after the removal of every edge, Fleury's algorithm will have a time complexity of . A dynamic bridge-finding algorithm of Thorup (2000) allows this to be improved to , but this is still significantly slower than alternative algorithms.

Hierholzer's algorithm

Hierholzer's 1873 paper provides a different method for finding Euler cycles that is more efficient than Fleury's algorithm:

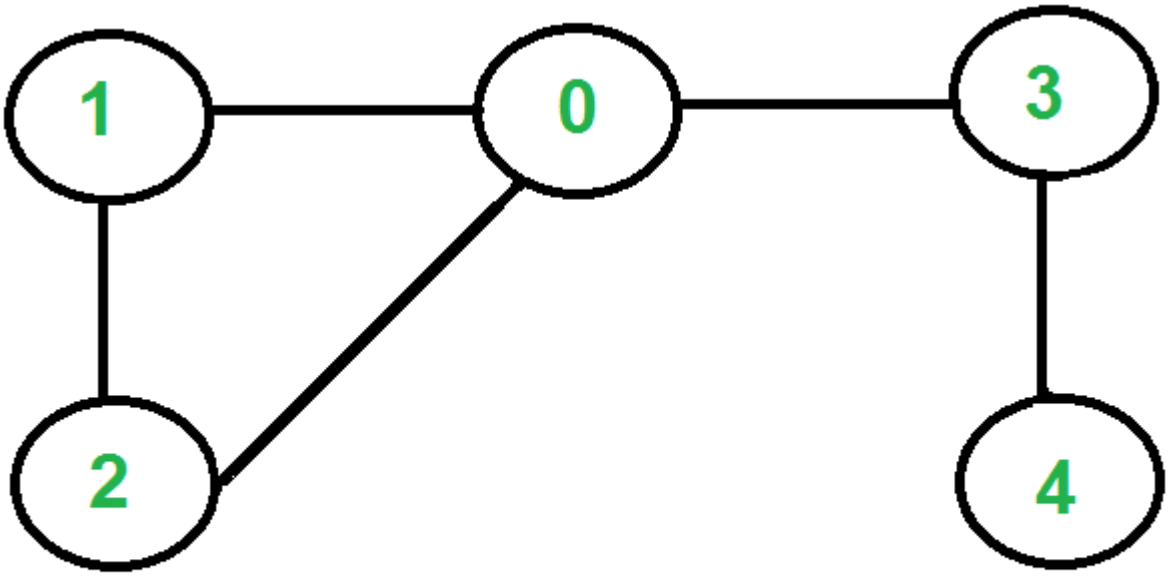
- Choose any starting vertex *v*, and follow a trail of edges from that vertex until returning to *v*. It is not possible to get stuck at any vertex other than *v*, because the even degree of all vertices ensures that, when the trail enters another vertex *w* there must be an unused edge leaving *w*. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.
- As long as there exists a vertex *u* that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from *u*, following unused edges until returning to *u*, and join the tour formed in this way to the previous tour.
- Since we assume the original graph is connected, repeating the previous step will exhaust all edges of the graph.

By using a data structure such as a doubly linked list to maintain the set of unused edges incident to each vertex, to maintain the list of vertices on the current tour that have unused edges, and to maintain the tour itself, the individual operations of the algorithm (finding unused edges exiting each vertex, finding a new starting vertex for a tour, and connecting two tours that share a vertex) may be performed in constant time each, so the overall algorithm takes linear time, .[9]

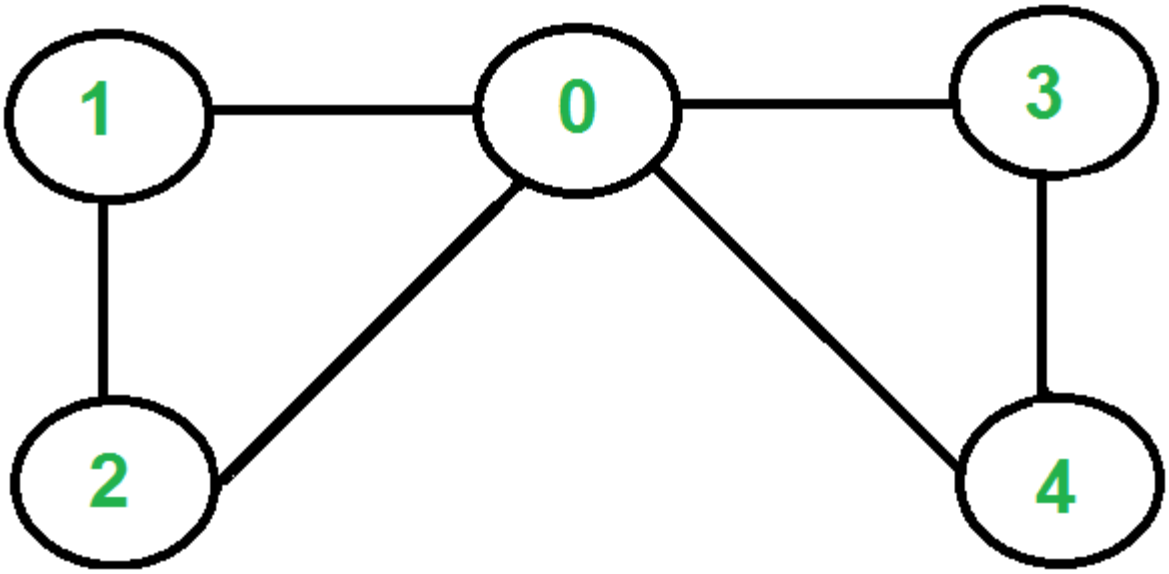
This algorithm may also be implemented with a deque. Because it is only possible to get stuck when the deque represents a closed tour, one should rotate the deque by removing edges from the tail and adding them to the head until unstuck, and then continue until all edges are accounted for. This also takes linear time, as the number of rotations performed is never larger than (intuitively, any "bad" edges are moved to the head, while fresh edges are added to the tail)

Eulerian path and circuit for undirected graph

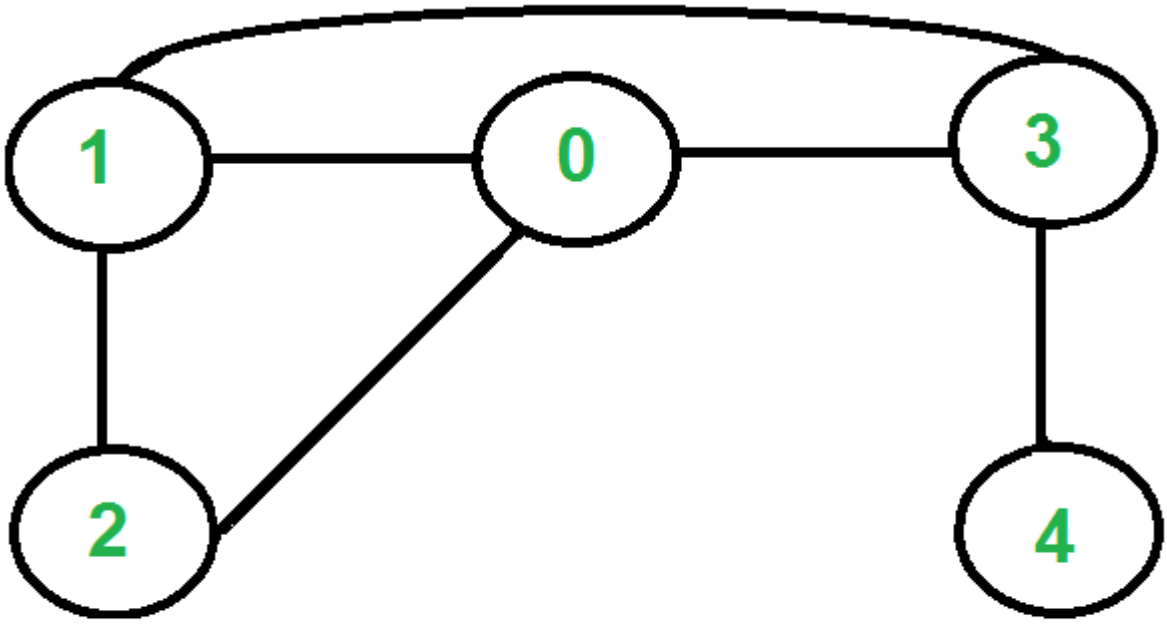
Eulerian Path is a path in a graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path that starts and ends on the same vertex.



The graph has Eulerian Paths, for example "4 3 0 1 2 0", but no Eulerian Cycle. Note that there are two vertices with odd degree (4 and 0)



The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2"
Note that all vertices have even degree



The graph is not Eulerian. Note that there are four vertices with odd degree (0, 1, 3 and 4)

How to find whether a given graph is Eulerian or not?

The problem is same as following question. “Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once”.
A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. The problem seems similar to Hamiltonian Path which is NP complete problem for a general graph. Fortunately, we can find whether a given graph has a Eulerian Path or not in polynomial time. In fact, we can find it in O(V+E) time.
Following are some interesting properties of undirected graphs with an Eulerian path and cycle. We can use these properties to find whether a graph is Eulerian or not.

Eulerian Cycle: An undirected graph has Eulerian cycle if following two conditions are true.

1. 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).
2. All vertices have even degree.

Eulerian Path: An undirected graph has Eulerian Path if following two conditions are true.

1. Same as condition (a) for Eulerian Cycle.
2. If zero or two vertices have odd degree and all other vertices have even degree. Note that only one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)

Note that a graph with no edges is considered Eulerian because there are no edges to traverse.

How does this work?
In Eulerian path, each time we visit a vertex v, we walk through two unvisited edges with one end point as v. Therefore, all middle vertices in Eulerian Path must have even degree. For Eulerian Cycle, any vertex can be middle vertex, therefore all vertices must have even degree.

Implementation:

```
// A C++ program to check if a given graph is Eulerian or not
#include<iostream>
#include <list>
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
public:
    // Constructor and destructor
    Graph(int V)   {this->V = V; adj = new list<int>[V]; }
    ~Graph() { delete [] adj; } // To avoid memory leak

     // function to add an edge to graph
    void addEdge(int v, int w);

    // Method to check if this graph is Eulerian or not
```

```
    int isEulerian();

    // Method to check if all non-zero degree vertices are connected
    bool isConnected();

    // Function to do DFS starting from v. Used in 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[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Method to check if all non-zero degree vertices are connected.
// It mainly does DFS traversal starting from
bool Graph::isConnected()
{
    // Mark all the vertices as not visited
    bool visited[V];
    int i;
    for (i = 0; i < V; i++)
        visited[i] = false;

    // Find a vertex with non-zero degree
    for (i = 0; i < V; i++)
        if (adj[i].size() != 0)
            break;

    // If there are no edges in the graph, return true
    if (i == V)
        return true;

    // Start DFS traversal from a vertex with non-zero degree
    DFSUtil(i, visited);

    // Check if all non-zero degree vertices are visited
    for (i = 0; i < V; i++)
       if (visited[i] == false && adj[i].size() > 0)
            return false;

    return true;
}

/* The function returns one of the following values
```

```
   0 --> If graph is not Eulerian
   1 --> If graph has an Euler path (Semi-Eulerian)
   2 --> If graph has an Euler Circuit (Eulerian)  */
int Graph::isEulerian()
{
    // Check if all non-zero degree vertices are connected
    if (isConnected() == false)
        return 0;

    // Count vertices with odd degree
    int odd = 0;
    for (int i = 0; i < V; i++)
        if (adj[i].size() & 1)
            odd++;

    // If count is more than 2, then graph is not Eulerian
    if (odd > 2)
        return 0;

    // If odd count is 2, then semi-eulerian.
    // If odd count is 0, then eulerian
    // Note that odd count can never be 1 for undirected graph
    return (odd)? 1 : 2;
}

// Function to run test cases
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";
}

// Driver program to test above function
int main()
{
    // Let us create and test graphs shown in above figures
    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);

    // Let us create a graph with 3 vertices
    // connected in the form of cycle
    Graph g4(3);
    g4.addEdge(0, 1);
    g4.addEdge(1, 2);
    g4.addEdge(2, 0);
    test(g4);

    // Let us create a graph with all vertices
    // with zero degree
    Graph g5(3);
    test(g5);

    return 0;
}
```

Output

```
graph has a Euler path
graph has a Euler cycle
graph is not Eulerian
graph has a Euler cycle
graph has a Euler cycle
```

Time Complexity: O(V+E)

Space Complexity: O(V+E)