Long Project - 0

PROJECT REPORT

Group – G13
Diksha Sharma (dxs134530) and Sharol Clerit Pereira (scp140130)
Implementation of Advanced Algorithms and Data Structures (CS6301) – Spring 2016
February 21, 2016

Instructor: Dr. Balaji Raghavachari

Abstract

A graph G is called Eulerian if it is connected and the degree of every vertex is an even number.

It is known that such graphs always have a tour (a cycle that may not be simple) that goes through every edge of the graph exactly once. Such a tour (sometimes called a circuit) is called an Euler tour.

If the graph is connected, and it has exactly 2 nodes of odd degree, then it has an Euler Path connecting these two nodes that includes all the edges of the graph exactly once.

In this project, we aim to find an Euler tour or an Euler Path in a given undirected and connected graph.

Introduction

An undirected graph G is called Eulerian if it is:

- Connected and
- Degree of every vertex is an even number.

Also Eulerian circuit or Eulerian cycle or Eulerian Tour is a trail which starts and ends on the same vertex.

It is known that such graphs always have a tour (a cycle that may not be simple) that goes through every edge of the graph exactly once. Such a tour (sometimes called a circuit) is called an Euler tour.

If the graph is connected, and it has exactly 2 nodes of odd degree, then it has an Euler Path connecting these two nodes that includes all the edges of the graph exactly once.

The initial code base we have used is:

- Edge.java
- Vertex.java
- Graph.java

The algorithm we have implemented to find the Euler tour/Path in the graph is known as Hierholzer's algorithm. Its description taken from Wikipedia is as below:

- Choose a 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.
- The overall algorithm takes linear time $O(|E|)$.
- The individual operations of the algorithm such as
    - finding unused edges exiting each vertex
    - finding a new starting vertex for a tour
    - connecting two tours that share a vertex may be performed in constant time

Methodology / Program Execution Steps

Following steps explain program execution:

- Read graph details from input file
- findEulerTour()
    - Takes graph as input
    - Outputs the tour/path found in the graph
    - Start tour/path from vertex 1.
    - Check the degrees of each of the vertices if even or not:
        - If exactly 2 vertices have odd degrees – find the vertex with smaller number as its name and start path finding using that vertex as the starting vertex. We find Euler path in this case.
        - If more than 2 vertices have odd degrees – The graph is not Eulerian – Exit execution

        - If all vertices have even degrees then we already have the first vertex (with smallest number as its name) set as the starting vertex. We find Euler tour in this case.
    - The next vertex to be visited should have index = current vertex's index +1.
        - To implement the index concept – we have modified the Vertex class to contain index field. It is same as the name of the vertex assuming all vertices have names as numbers without any gaps.
    - If the adjacency list of the current vertex does not have any edge to the next higher index then increment the index by 1 and search again.
    - If the adjacency list of the current vertex has an edge to next higher index and the edge has not yet been visited then add that edge to tour/path and mark that edge as seen (Edge class has been modified to include field "seen" for this). Once this edge is added to the tour then the next vertex becomes the other end of the edge we just visited. We now look for the next edge to visit in the new vertex's adjacency list. The new index we search will have a value equal to new vertex's index + 1.
    - If the adjacency list of the current vertex has an edge to the next higher index and the edge has already been visited then increment the index and look in the list again for another edge that has not been yet visited and whose name matches the index.
    - If the index is smaller than the total number of vertices – we keep incrementing the index to look for next possible vertex we can visit. However, once we reach the max value of the vertex, we restart index from 1.
    - We keep repeating the process till we have visited all edges in the graph. We verify the tour using the method verifyTour() which takes the graph and the tour as its input.
- verifyTour()
    - This method takes the graph and the tour/path as its input.
    - It goes through all the edges in the graph for their "seen" field. If there is any edge not yet visited (seen = false) then the function returns false.
    - If all edges have been visited then the method checks if each edge of the tour has a common vertex with its adjacent edge visited next in the tour. This is required since we are trying to find a path/tour in the graph and it should not be disconnected.
    - If the tour has disconnected adjacent edges then the method returns false else returns true – which indicates that the current tour/path is an Eulerian tour/path.

Experimental Setup

There are 4 source files in the submission folder:

- Vertex.java – contains the vertex implementation
- Edge.java – contains the edge implementation
- Graph.java – contains the graph implementation. We use only the undirected graph components

- EulerianTraversal.java – contains the driver program which evaluates the graph and finds the Euler tour or path and outputs it on console.

Input arguments:

The driver program takes the full path of the file containing the input graph's details. The input file should have the format as below:

#of vertices #of Edges

<Edge details from line 2 onwards>

The program does not perform any exception handling to determine if the input graph is correct and the details provided in the file are accurate. A correct graph is assumed by the program.

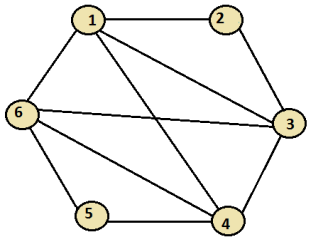
Test Results and Discussion

The program logic is not dependent on the order in which the edges of the input graph are read. The program will always output the same path/tour as we use the index to determine the next edge to be visited.

To test the program, we used the sample graph provided in the project description as below:

Input graph:

```
6 10
1 2 1
1 3 1
1 4 1
1 6 1
2 3 1
3 6 1
3 4 1
4 5 1
4 6 1
5 6 1
```

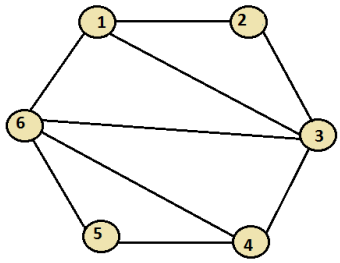


Euler Tour: (Formatted Output from program)

(1, 2)
(2, 3)
(3, 4)
(4, 5)
(5, 6)
(1, 6)
(1, 3)
(3, 6)
(4, 6)
(1, 4)

Tour: 1➪ 2 ➪ 3 ➪ 4 ➪ 5 ➪ 6 ➪ 1 ➪ 3 ➪ 6 ➪ 4 ➪ 1

If we remove the edge (1, 4) from the graph we get the Euler path for the same graph.
Input graph:
6 9
1 2 1
1 3 1
1 6 1
2 3 1
3 6 1
3 4 1
4 5 1
4 6 1
5 6 1



Euler Path: (Formatted Output from program)

(1, 2)
(2, 3)
(3, 4)
(4, 5)
(5, 6)
(1, 6)
(1, 3)
(3, 6)
(4, 6)

Path: 1➪ 2 ➪ 3 ➪ 4 ➪ 5 ➪ 6 ➪ 1 ➪ 3 ➪ 6 ➪ 4

Conclusion

Average Time taken statistics collected:

Test input file (mentioned above):

```
Time: 0 msec.
Memory: 2 MB / 128 MB.
```

Tour/Path (5K vertices):

```
Time: 78554 msec.
Memory: 17 MB / 115 MB.
```

None:

```
Time: 0 msec.
Memory: 9 MB / 128 MB.
```