Eulerian Trails and Tours

Joseph Willard

December 29, 2016

1 Introduction

This post discusses the idea behind Eulerian trails and tours. The first section provides some historical insight into the topics as well as a description of the algorithm and related theorems complete with proof. The second section gives python code that simulates finding trails or tours and an explanation behind the process.

2 Eulerian Trails and Tours

2.1 Definitions

Definition 2.1. *Given a graph G the* ***degree*** *of a vertex is the number of edges that touch the vertex. In figure 1 vertex E on the left graph has a degree of 4, this is often simplified as* $\deg(E) = 4$.

Definition 2.2. *Given a graph G, a* ***walk*** *is an alternating sequence of vertices and edges $< v_0, e_1, v_1, \cdots, e_n, v_n >$. A walk is considered* ***closed*** *if $v_0 = v_n$.*

Definition 2.3. *A graph G is* ***connected*** *if there exists a walk from any two vertices in G.*

Definition 2.4. *A walk is called a* ***trail*** *if no edges are repeated.*

Definition 2.5. *A graph G has an* ***Eulerian Trail*** *if there exists a walk using every edge (see figure 1).*

Definition 2.6. *A graph G has an* ***Eulerian Tour*** *if there exists a closed walk using every edge (see figure 1).*

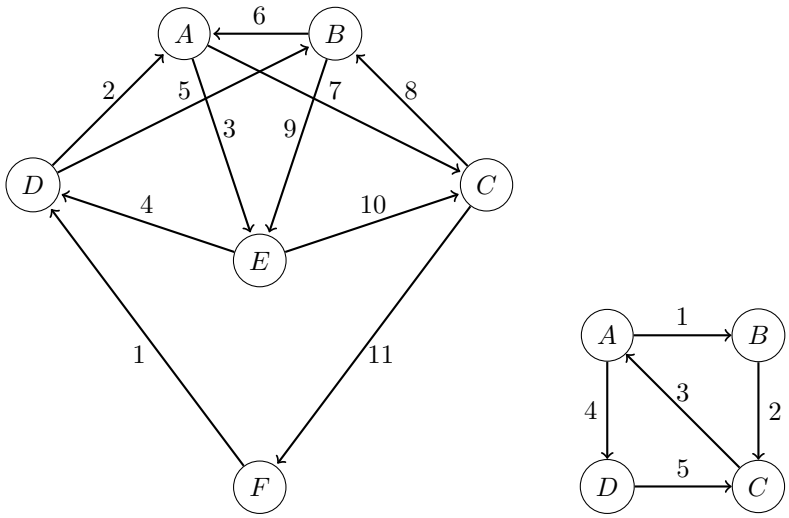

Figure 1: Eulerian Tour (left) and Eulerian Trail (right)

2.2 History

Historically the idea of of an Eulerian Tour/Trail can be linked back to the Seven Bridges of Königsberg (see figure 2), which was at the time a city in Prussia but now located in Russia. The question involves finding a way to cross each bridge only once. Euler reformulated the question such that it relied on the number of bridge crossings, labelling the land masses as nodes. This allowed him to make generalize that to use every bridge in a graph there must either exist all even degree vertices or exactly two (see below for proof).

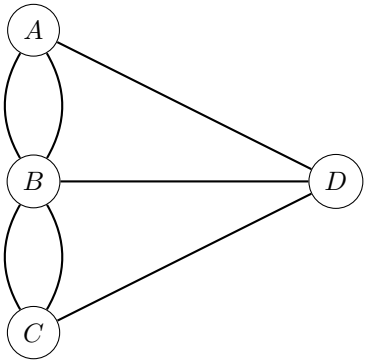

Figure 2: Bridges of Königsberg, the nodes are islands and the edges are bridges.

2.3 Theorems

Theorem 2.1. *If G is a connected finite graph with every vertex having even degree, then G has an Eulerian Tour.*

Proof. Before addressing this it is important to address the case when a graph G has no edges. In this case then there trivially exists an Eulerian Tour since there exists no edges to use. In the case where G is connected arbitrarily choose a starting vertex v_0. since it is given that each vertex of G is even the when moving from v_0 to an adjacent vertex v_1 both have an odd number of unused edges incident to them respectively. Continue this process and every time an unused edge is chosen the incident vertices will have an even number of unused edges except the v_0. continue doing this until the starting vertex is reached, which is possible because G is connected and every vertex except v_0 must have an unused edge to leave with.

If all edges are used in this process then the resulting path gives an Eulerian Tour. If there still exists unused edges then starting from an incident vertex repeat the same process which is again possible since G is connected and in the from the first iteration of this process we used an even number of edges hence each vertex should still have an even degree. □

Theorem 2.2. *If G is a connected finite graph with exactly two vertices of odd degree v and u, then there is an Euleria Trail starting from v and ending at u (same can be said by starting at u and ending at v).*

Proof. Without loss of generality starting at vertex v start creating a path always choosing an unused edge. Continue this process until a vertex is reached that has no unsused edges to leave by. Since every vertex except u and v must have even degree when we approach any of these vertices there is an edge to leave with, furthermore, by starting at v only one edge was used to leave hence it has an even amount of unused edges. Putting all these observations together

it is clear that the only possible vertex to get stuck at must be one with an odd degree which is u.

If all edges were used in this process then the output will be an Eulerian Trail. However, if there remains unused edges then they must have even degree, since u is the only possible vertex to get stuck at. From these unused vertices create another path and append it to the originall creating an Eulerian Trail. □

3 Code

```
'''
This takes input in the form of tuples (see the bottom of program
    for example) representing edges of a graph and determines
    whether they form either an Eulerian trail/tour. If they do it
    returns the edges in the respective order.
'''


import networkx as nx
import matplotlib.pyplot as plt
import ast as a



edges_input = input("Please enter edges as tuples followed by a
    comma (e.x, (1,2), (2,3), ... ): ")
edges = a.literal_eval(edges_input)
vertices = int(input("How many vertices in total? "))


def degrees():
    """
    This determines the degree of each respective vertex.
    """
    a = edges
    b = []
    odd = []
    zeroes = []

    for i in range(vertices):
        c = 0
        for j in range(len(edges)):
            if i+1 == a[j][0]:
                c += 1
            if i+1 == a[j][1]:
                c += 1
        b.append(c)
    for i in range(vertices):
        if b[i]%2 != 0:
            odd.append(i+1)
    for i in range(vertices):
        if b[i] == 0:
            zeroes.append(i+1)

    return b, odd, zeroes
```

```
def connected():
    """
    This determines whether the graph is connected by performing a
    breadth first search.
    """


    b = list(edges)
    path = []
    vertices_travelled = [b[0][0]]
    f = 0

    while f < vertices + 1:
        for i in range(len(b)):
            if b[i][0] not in vertices_travelled and b[i][1] in
    vertices_travelled:
                vertices_travelled.append(b[i][0])
                path.append(b[i])
            elif b[i][1] not in vertices_travelled and b[i][0] in
    vertices_travelled:
                vertices_travelled.append(b[i][1])
                path.append(b[i])
            elif b[i][1] in vertices_travelled and b[i][0] in
    vertices_travelled:
                path.append(b[i])
            elif b[i][1] not in vertices_travelled and b[i][0] not
    in vertices_travelled:
                pass
        f += 1
        b = list(set(b) - set(path))

    return  vertices_travelled, b



def euler_trail(x):
    """
    This chooses one of the odd degree vertices and builds a path
    starting from there.
    """
    b = list(x)
    a = degrees()[1][0]
    path = []

    for i in range(len(x)):
        for i in range(len(b)):
            if b[i][0] == a:
                path.append(b[i])
                a = b[i][1]

            elif b[i][1] == a:
                path.append(b[i])
                a = b[i][0]

        for i in range(len(path)):
```

```
            if path[i] in b:
                del b[b.index(path[i])]



    return path, b

def euler_tour(x):
    """
    This chooses the first available vertex and builds a path
    starting from there by always choosing an unused edge.
    """
    b = list(x)
    a = b[0][0]
    path = []
    y = 0

    for y in range(len(x)):
        for i in range(len(b)):
            if b[i][0] == a:
                path.append(b[i])
                a = b[i][1]

            elif b[i][1] == a:
                path.append(b[i])
                a = b[i][0]

        for j in range(len(path)):
            if path[j] in b:
                del b[b.index(path[j])]


    return path, b




def cpath(x):
    """
    This function takes the unused edges from either euler_trail or
     euler_tour and makes a new path.
    """
    opath = x[0]
    b = x[1]
    a = b[0][0]
    path = []
    y = vertices
    i = 0


    while True:
        if len(b) > 0:
            if i < len(b):
                while True:
                    if b[i][0] == a:
                        path.append(b[i])
```

```
                        a = b[i][1]
                        del b[i]
                        i = 0
                        break

                    elif b[i][1] == a:
                        path.append(b[i])
                        a = b[i][0]
                        del b[i]
                        i = 0
                        break

                    else:
                        y = y - 1
                        i += 1
                        break
            else:
                break


        else:
            break







    return opath, path, b



def add(x):
    """
    This takes trails from the cpath and finds a place to append
    them in the trail created by either euler_tour or euler_trail.
    """
    fpath = []
    npath = x[0]
    path = x[1]
    z = x[2]
    c = 0

    while True:
        if len(z) > 0:
            fpath = []

            while True: #c < len(x[0]) + 1:


                if path[0][0] in npath[c] and path[-1][0] in npath[
    c+1]:
                    npath.insert(c+1, path)
                    break
```

```
            elif path[0][0] in npath[c] and path[-1][1] in
npath[c+1]:
                npath.insert(c+1, path)
                break
            elif path[0][1] in npath[c] and path[-1][0] in
npath[c+1]:
                npath.insert(c+1, path)
                break
            elif path[0][1] in npath[c] and path[-1][1] in
npath[c+1]:
                npath.insert(c+1, path)
                break
            c += 1

        for i in range(len(npath)):
            if type(npath[i]) == list:
                for j in range(len(npath[i])):
                    fpath.append(npath[i][j])
            else:
                fpath.append(npath[i])

        npath = fpath
        new_path = cpath((npath, z))
        path = new_path[1]
        z = new_path[2]
        c = 0


    elif len(z) == 0 and len(path) > 0:
        fpath = []

        while True:


            if path[0][0] in npath[c] and path[-1][0] in npath[
c+1]:
                npath.insert(c+1, path)
                break
            elif path[0][0] in npath[c] and path[-1][1] in
npath[c+1]:
                npath.insert(c+1, path)
                break
            elif path[0][1] in npath[c] and path[-1][0] in
npath[c+1]:
                npath.insert(c+1, path)
                break
            elif path[0][1] in npath[c] and path[-1][1] in
npath[c+1]:
                npath.insert(c+1, path)
                break
            c += 1

        for i in range(len(npath)):
            if type(npath[i]) == list:
                for j in range(len(npath[i])):
                    fpath.append(npath[i][j])
            else:
```

```
                    fpath.append(npath[i])

            npath = fpath


            c = 0
            break
        else:
            break


    return npath, z


def clean(x):
    """
    This takes the input and arranges the tuples so that the final
    output gives a walk to follow (i.e (1,2), (2,3), ...etc).
    """
    path = x[0]
    clean = [path[0]]

    for i in range(len(path) - 1):
        if clean[i][1] == path[i+1][0]:
            clean.append(path[i+1])
        else:
            clean.append((path[i+1][1], path[i+1][0]))

    return clean



def trail(x):
    """
    This fucntions purpose is to give the correct output of the
    program taking into account whether the input is connected and
    has the correct number of odd degree vertices.
    """

    if len(connected()[0]) != vertices:
        return "This graph is not connected!"

    elif len(degrees()[1]) == 2 and len(euler_trail(x)[1]) == 0:
        return  "The Eulerian Trail is: ", clean(euler_trail(x))

    elif len(degrees()[1]) == 2 and len(euler_trail(x)[1]) != 0:
        return "The Eulerian Trail is: ", clean(add(cpath(
    euler_trail(x))))

    elif len(degrees()[1]) == 0 and len(euler_tour(x)[1]) != 0:
        return "The Eulerian Tour is: ", clean(add(cpath(euler_tour
    (x))))

    elif len(degrees()[1]) == 0:
        return "The Eulerian Tour is: ", clean(euler_tour(x))
```

```
    else:
        return "there exists neither an Eulerian Tour or Trail due
    to the number of odd degree vertices."

G = nx.Graph(list(edges))
nx.draw(G)
plt.show()
print(trail(edges))
```

#5 ,(1, 2), (2, 4), (5, 1), (2, 3), (5, 4), (2, 3), (4, 1), (1, 3), trail

#10 (1, 2), (4, 4), (2, 4), (7, 8), (5, 1), (8, 9), (2, 3), (6, 8), (5, 4), (7, 8), (2, 3), (10, 9), (4, 1), (6, 10), (1, 3), not connected

#14, (1, 2), (9, 11), (4, 4), (3, 14), (2, 4), (5, 11), (7, 8), (5, 1), (11, 13), (8, 9), (5, 13), (2, 3), (14, 11), (6, 8), (13, 2), (5, 4), (7, 8), (9, 14), (2, 3), (10, 9), (4, 1), (13, 2), (6, 10), (1, 3), not connected since 12 is never reached.

#16, (1, 2), (9, 11), (4, 4), (3, 14), (2, 4), (15, 12), (6, 9), (5, 11), (7, 8), (5, 1), (11, 13), (8, 9), (5, 13), (2, 3), (14, 11), (6, 8), (13, 2), (5, 4), (16, 1), (7, 8), (15, 6), (9, 14), (11, 16), (2, 3), (15, 2), (4, 12), (10, 9), (4, 1), (13, 2), (6, 10), (1, 3) Too many odd.

#18, (1, 2), (9, 11), (4, 4), (17, 16), (3, 14), (16, 15), (2, 4), (15, 12), (6, 9), (13, 13), (5, 11), (7, 8), (17, 11), (2, 14), (5, 1), (11, 13), (8, 9), (5, 13), (2, 3), (1, 18), (14, 11), (6, 8), (13, 2),(5, 4), (18, 9), (16, 1), (7, 8), (15, 6), (9, 14), (11, 16), (2, 3), (15, 2), (4, 12), (10, 9), (4, 1), (13, 2), (6, 10), (1, 3) tour

#15, (1, 2), (1, 3), (1, 4), (2, 3), (4, 5), (4, 6), (4, 7), (5, 8), (5, 9), (5, 10), (6, 7), (8, 9), (10, 11), (10, 12), (10, 13), (11, 14), (11, 15), (12, 13), (14, 15) trail