A General implementation of Eulerian Path

Edwardo S. Rivera Hazim
edwardo.rivera@upr.edu

Advisor:
Humberto Ortiz Zuazaga
humberto@hpcf.upr.edu
Computer Science Department
University of Puerto Rico - Río Piedras

May 22, 2015

Abstract

NetworkX is an open source project for complex networks for the Python programming language. Although it has a huge amount of complex algorithms, other algorithms like finding an Eulerian Path in a Graph are not implemented. Motivated by the need of this algorithm in a bio-informatics problem, we implement an efficient Python function for networkX to find an Eulerian Path in any type of Graph: directed, undirected or Multigraph. The long-term goal is to contribute our function to the open source project.

1 Introduction

The Konigsberg Bridge Problem, solved by the famous mathematician Euler, raised the new branch (at that time) of Graph Theory. Nowadays, the Eulerian path algorithm -a path that uses every edge of a graph exactly once- is well understood in the Computer Science community. Applications range from air traffic, mail carriers to bio-informatics. Nonetheless, built-in libraries for any type of Graphs in the Python programming language are not provided.

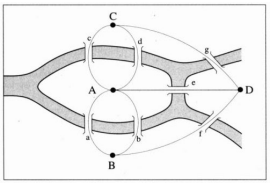

Figure 1: The Konigsberg Bridge Problem

In this work, we present an efficient and general implementation in Python of this algorithm for any type of graph using the open source library NetworkX, which is a third party package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks [1].

2 Motivation and NetworkX

The motivation of this work started when some lab partners where doing research in bio-informatics, specifically in sequence assembly using the NetworkX Python third party library. Our advisor found that we needed the Eulerian Path algorithm. For more information on how an Euler path is used in sequence assembly, see [2].

NetworkX, an open source project that support Python 2.7, provides data structures for graphs (or networks) along with graph algorithms, generators, and drawing tools. The basic data structure of NetworkX is represented as a "dictionary of dictionaries of dictionaries" [3]. Although it provides a range of graph algorithms, the Eulerian Path was not implemented. Hence, we decided to implement it for the aforementioned reasons and to contribute to the open source project.

3 Methodology

To implement the algorithm, a pseudo-code was followed, as in [4]. First, the code for Graphs was studied and implemented. Then, the Digraph code, the one that was needed for the bio-informatics research was done. Finally, a general code that works for Graphs, Digraphs and Multigraphs were merged.

To accomplish this, the power of NetworkX and Python was used to derive a general code for (undirected) Graphs and Digraphs. NetworkX functions and algorithms such as Eulerian circuit [5] were used and Python generators for efficient code was also used. The resulting code is showed here, some comments were leave out for readability:

```
import networkx as nx

"""
Find an Eulerian Path
"""
#    Copyright (C) 2015 by
#    Edwardo S. Rivera <edwardo.rivera@upr.edu>
#    Humberto Ortiz <humberto@hpcf.upr.edu>
#    BSD license.
#    All rights reserved.
```

```
def find_eulerian_path(G):
    from operator import itemgetter

    # Verify that graph is connected, short circuit
    if G.is_directed() and not nx.is_weakly_connected(G):
        raise nx.NetworkXError("G is not connected.")

    # is undirected
    if not G.is_directed() and not nx.is_connected(G):
        raise nx.NetworkXError("G is not connected.")

    # Now verify if has an eulerian circuit:
    # even condition of all nodes is satified.
    if nx.is_eulerian(G):
        x = nx.eulerian_circuit(G) # generator of edges
        for i in x:
            yield i

    # Not all vertex have even degree, check if exactly two vertex
    # have odd degrees. If yes, then there is an Euler path. If not,
    # raise an error (no euler path can be found)
    else:
        g = G.__class__(G)  # copy graph structure (not attributes)

        # list to check the odd degree condition, and a flag
        check_odd = []
        directed = False

        if g.is_directed():
            degree = g.in_degree
            out_degree = g.out_degree
            edges = g.in_edges_iter
            get_vertex = itemgetter(0)
            directed = True
        else:
            degree = g.degree
            edges = g.edges_iter
            get_vertex = itemgetter(1)

        # Verify if an euler path can be found. Complexity O(n)
        for vertex in g.nodes():
            deg = degree(vertex)
            # directed case
            if directed:
                outdeg = out_degree(vertex)
                if deg != outdeg:
```

```
            # if we have more than 2 odd nodes,we do a raise: no euler path
                if len(check_odd) > 2:
                    raise nx.NetworkXError("G doesn't have an Euler Path.")
            # is odd and we append it.
                else:
                    check_odd.append(vertex)
        # undirected case
        else:
            if deg % 2 != 0:
            # if we have more than 2 odd nodes,we do a raise: no euler path
                if len(check_odd) > 2:
                    raise nx.NetworkXError("G doesn't have an Euler Path.")
            # is odd and we append it.
                else:
                    check_odd.append(vertex)

    if directed:
        def verify_odd_cond(g,check_odd):
            first = check_odd[0]
            second = check_odd[1]
            if (g.out_degree(first) == g.in_degree(first) + 1 and
                g.in_degree(second) == g.out_degree(second) + 1):
                return second
            elif (g.out_degree(second) == g.in_degree(second) + 1
                and g.in_degree(first)  == g.out_degree(first) + 1):
                return first
            else:
                return None
        start = verify_odd_cond(g,check_odd)
    else:
        start = check_odd[0]

    # if the odd condition is not meet, raise an error.
    if not start:
        raise nx.NetworkXError("G doesn't have an Euler Path")

    # Begin algorithm:
    vertex_stack = [start]
    last_vertex = None

    while vertex_stack:

        current_vertex = vertex_stack[-1] #(4)
        # if no neighbors:
        if degree(current_vertex) == 0:
            # Special case, we cannot add a None vertex to the path.
```

```
            if last_vertex is not None:
                yield (last_vertex, current_vertex)
            last_vertex = current_vertex
            vertex_stack.pop()
        # we have neighbors, so add the vertex to the stack (2), take
        # any of its neighbors (1) remove the edge between selected
        # neighbor and that vertex, and set that neighbor as the
        # current vertex (4).
        else:
            random_edge = next(edges(current_vertex)) #(1)
            vertex_stack.append(get_vertex(random_edge)) #(2)
            g.remove_edge(*random_edge) #(3)
```

3.1 Digraph implementation

Given that the Graph and Multigraph case are the same, and are more easy to understand that the Digraph one, a discussion of the later must be mentioned. Also, the motivation for finding an Eulerian Path was for the Digraph case, so a general overview on how the code is organized for this case will be given here.

1. Verify that the Digraph is connected. If it is not, then it does not have an Eulerian Path, and an error is raised.
2. If it has an Eulerian circuit, (the case where all vertices's has the same out and in-degree) just call the eulerian_cicuit() function, and we are done.
3. If it does not have an Eulerian circuit, then we have to check if exactly only two nodes of the Digraph does not have the same out and in degree.
    3.1. If from these two nodes, one of them has out-degree with one greater than its in-degree, and the other has in-degree with one greater than its out-degree, then choose the vertex that has its in-degree with one greater than its out-degree. Choosing this vertex as the starting vertex enforce that when we actually give the path, it will be in the right order.
    3.2. If not, then the Digraph does not have an Eulerian Path, and an error is raised.
4. If conditions from point (3) are satisfied, run the main algorithm to find the Eulerian Path.

4 Results

The Eulerian Path algorithm has been implemented (See Section 2) in which it works for Graphs, Digraphs and Multigraphs. Right now, it is used in our lab to do bio-informatics research in sequence assemby by (at least) Ricardo Lopez and Omar Rosado.

5 Future Work

A pull request of the algorithm has been done to the networkX repository at github [6]. A script for testing the algorithm is been developed using nosetools, a package to do testing in Python.

In the immediate future, the script for testing would be finish. The pull request would be edited to meet networkX requirements. After this, we believe that our algorithm would be part of the networkX library.

Acknowledgements

Thanks to Humberto Ortiz-Zuazaga for his support as a mentor and as a research scientist along the way.

References

[1] NetworkX. (n.d.). Retrieved May 22, 2015, from https://networkx.github.io/

[2] Rosado, O., & Ortiz, H. (2015). Rethinking Sequence Assembly. University of Puerto Rico, Rio Piedras Campus.

[3] NetworkX Documentation. (n.d.). Retrieved May 22, 2015, from https://networkx.github.io/documentation/latest/reference/introduction.html

[4] Graph Magics. (n.d.). Retrieved May 23, 2015, from http://www.graph-magics.com/articles/euler.php

[5] Networkx eulerian circuit code. (n.d.). Retrieved May 23, 2015, from https://github.com/networkx/networkx/blob/master/networkx/algorithms/euler.py

[6] Added euler path for Graphs, Digraphs and Multigraphs by edwardo2013 · Pull Request #1488 · networkx/networkx. (n.d.). Retrieved May 23, 2015, from https://github.com/networkx/networkx/pull/1488