Using OSM to Optimize Walking Trails

Abinav Chari, Pranav Koushik, Sathvik Vangavolu

December 7, 2021

Contents

Introduction

1.1 Abstract

The purpose of this research is to find trails or paths that one who goes for a walk outside could take without having to enter the same area again. This can help prevent people from getting lost or trying to guess and check where a route leads. In this paper, we will first go over the code that creates the graph from OpenStreetMap (OSM) data. Then, we will compare different algorithms to find Euler circuits and paths, and show one implemented in code. Finally, we will discuss limitations of the program.

1.2 Introduction

Graph theory is often used on real paths and roads to minimize a cost. For example, Google Maps utilizes graph theory to find the quickest paths from two locations, or nodes. The goal of this project, on the other hand, is to find an order that would allow one to travel every path between several locations. The most useful application of this would be to plan hiking trails: each landmark or intersection of paths would be a node, and each trail an edge.

Creating the Graph

- Nodes (OSM): A point with a specific longitude and latitude, which usually describes a specific landmark.
- Ways (OSM): A path consisting of several nodes. Ways are ordered by which nodes one passes through to get from node A to B.

2.1 What is OSM?

OSM is a community-built map system, somewhat similar to Google Maps. The reason we used OSM over Google Maps is because of its availability: OSM data is very easily accessible. Given left, right, upper, and lower latitudinal and longitudinal boundaries, OSM will create an XML file with nodes and ways. However, we cannot directly use this data to find paths. The Python script in Appendix A reads the XML file, breaks up the Ways into several edges using a recursive function (lines 20-43), and creates a traditional graph of the given map data.

2.2 Finding Paths

- Euler path - Given two nodes A and B, an Euler path is a sequence of edges that start with A and end with B, where every edge is visited exactly once.
- Euler circuit - An Euler path where A is the same node as B.

Ideally, when creating a hiking path, we will want to walk through every path possible. To do this, we will find an Euler path or circuit in the graph. Fortunately, the NetworkX library in Python makes it very simple to find both Euler paths and circuits in the graph. However, many of the trail graphs will not have an Euler path or circuit starting from a specified node. Instead of informing the user that visiting all edges exactly once is impossible, we will simply search for the next best options. That is, either a path that visits every edge at least once, or the biggest path which does not repeat any edges. We will refer to these trails as exhaustive trails and maximized trails.

2.3 Algorithms for Finding Exhaustive Circuits

Terms:

- Complete graph: A complete graph is a graph with n vertices where each vertex has degree $n-1$: all vertices are neighbors of each other.

Finding exhaustive trails is a very famous question in graph theory. Commonly known as the Chinese Postman Problem, we must find the smallest trail in an undirected graph that visits every edge at least once. To solve this problem, we must define the concept of a T-join.

- Given T is a subset of the graph's vertex set, we call an edge set J a T-join if the odd-degree vertices in J are exactly T. For this problem, we will define T as the set of all odd-degree vertices.

With this concept, we can find an exhaustive circuit by duplicating the edges found in J in the original graph, creating an Eulerian multigraph. However, to create an optimal solution, we must minimize the edge cost of the circuit. Therefore, we will have to find a minimum T-join, using the algorithm defined below.

1. For each pair of vertices u and $v \in T$, let $w(uv)$ be the shortest path distance from u to v in the original graph. P_{uv} will be the shortest path from u to v. Use the Djikstra's algorithm to determine the shortest path (defined in Appendix B).
2. Define H to be the complete graph, with vertices T and edge weights w(uv).
3. Use the Blossom algorithm [2] to find a minimum weight perfect matching M in H.
4. Output the set of edges that occur in an odd number of paths P_{uv} where $uv \in M$.

Now that we have a minimum T-join, we simply duplicate these edges in the original graph, and compute an Euler circuit. Both the Hungarian algorithm and Floyd-Warshall algorithm run in $O(n^3)$, where n is the number of vertices in the graph. Thus, the entire algorithm for finding an exhaustive circuit runs in polynomial time.

2.4 Algorithms for Finding Maximized Trails or Circuits

As we explained earlier, a maximized trail is a path that travels the most edges possible without repeating an edge. In other words, we must find the subgraph S with the largest number of edges which contains an Euler trail. The algorithm for this is actually quite simple: by Euler's theorem, a graph G with an Euler trail has at most 2 vertices of odd degree. G is not Eulerian if and only if there are more than two vertices of odd degree. So, to find the Eulerian subgraph of G with the largest number of edges, we simply remove edges from those with odd degree until there are only two left. Then, we call the NetworkX function to compute an Euler trail. We will prioritize removing edges from vertices with the highest degree to avoid forming loose points as much as possible. To find an Euler circuit, a similar process is used: by Euler's theorem, a graph G with an Euler circuit has only even degree nodes. So, we simply remove 1 edge from every vertex with an odd degree. For both of these methods, we will prioritize a maximum number of edges, where we start by removing edges between two vertices which both have odd degrees. This algorithm also runs in polynomial time.

Analysis

3.1 Limitations

The main limitation in this program comes from the reliability - or lack thereof - of OpenStreetMap. Because the data is created by users, some areas may not have nodes or ways, which obviously makes this program ineffective.

3.2 Conclusion

Graph theory is already very popular for optimal pathfinding, as seen in Google Maps and other mapping software. This paper explores a different use of graph theory using Euler trails and circuits. The program provides a relatively simple method to mapping out hiking paths so that a user goes through as many trails - or edges - as possible. In the future, we could add to this program by using weighted graphs to search for the highest order path closest to a specific weight (i.e. a mapped trail closest to 4 miles long). This program lays out basic groundwork for the use of graph theory and OSM data to find paths not based on minimizing a cost.

Appendix

4.1 Appendix A

```python
import xml.sax
import copy
import networkx
import urllib3
from bs4 import BeautifulSoup

class Node:
   def __init__(self, id):
       self.id = id
       self.tags = {}


class Way:
   def __init__(self, osm, id):
       self.osm = osm
       self.id = id
       self.nodes = []
       self.tags = {}

   def split(self, dividers):
       # Recursive function to slice node array
       def slice_array(array,dividers):
           for i in range(1,len(array)-1):
               if dividers[array[i]] >= 2:
                   l = array[0:i+1]
                   r = array[i:]

                   rsliced = slice_array(right,dividers)

                   return [left]+rsliced
           return [array]

       slices = slice_array(self.nodes,dividers)

       ways = []
       j = 0
```

```
        for slice in slices:
            way = copy.copy(self)
            way.id += "-"+j
            way.nodes = slice
            ways.append(way)
            j += 1
        return ways


def downloadOSM(l,b,r,t):
    http = urllib3.PoolManager()
    try:
        response = http.request('GET',"http://api.openstreetmap.org/
    api/0.6/map?bbox=%f,%f,%f,%f"%(l,b,r,t))
        soup = BeautifulSoup(response.data)
        return response
    except:
        print("OSM download failed.")

def readOSM(filename):
    osm = OSM(filename)
    graph = networkx.Graph()

    for x in osm.ways.itervalues():
        if 'highway' in x.tags:
            graph.add_path(x.nds,id=x.id,data=x)
    for node_id in graph.nodes_iter():
        n = osm.nodes[node_id]
        graph.node[node_id] = dict(data=n)
        return graph


class OSM:
    def __init__(self, filename_or_stream):
        nodes = {}
        ways = {}

        superself = self

        class OSMHandler(xml.sax.ContentHandler):
            @classmethod
            def setDocumentLocator(self, loc):
                pass

            @classmethod
            def startDocument(self):
                pass

            @classmethod
            def endDocument(self):
                pass

            @classmethod
            def startElement(self, name, attrs):
                if name == 'node':
                    self.currElem = Node(attrs['id'], float(attrs['
    lon']), float(attrs['lat']))
```

```
                elif name == 'way':
                    self.currElem = Way(attrs['id'], superself)
                elif name == 'tag':
                    self.currElem.tags[attrs['k']] = attrs['v']
                elif name == 'nd':
                    self.currElem.nds.append(attrs['ref'])

            @classmethod
            def endElement(self, name):
                if name == 'node':
                    nodes[self.currElem.id] = self.currElem
                elif name == 'way':
                    ways[self.currElem.id] = self.currElem

            @classmethod
            def characters(self, chars):
                pass

        xml.sax.parse(filename_or_stream, OSMHandler)

        self.nodes = nodes
        self.ways = ways

        node_counter = dict.fromkeys(self.nodes.keys(),0)
        for w in self.ways.values():
            if len(w.nodes) <= 1:
                del self.ways[w.id]
            else:
                for node in w.nodes:
                    node_counter[node] += 1

        final_ways = {}
        for id, way in self.ways.iteritems():
            split_ways = way.split(node_counter)
            for split in split_ways:
                final_ways[split.id] = split
        self.ways = final_ways
```

4.2 Appendix B

Breadth-first search is a way to traverse a graph, sometimes repeating each vertex. This is similar to what we learned through the greedy algorithm for an Euler circuit. The difference, however, is that this is split up into layers. To find the smallest path from vertex V to U, the steps are as follows:

1) Start from V, and visit all the nodes neighboring vertex V. Check to see if any of these vertices are U. If one is, trace the path back to V, and return the list of vertices.

2) After visiting every node in this ”layer”, remove them from the graph and continue to the next layer.

3) Repeat this process.

Bibliography

[1] Keller, Mitchel T., and William T. Trotter. Applied Combinatorics. Langara College, 2019.

[2] Shoemaker, Amy, and Sagar Vare. Edmonds' Blossom Algorithm. Stanford University,6 June 2016,

[3] OpenStreetmaps. https://wiki.openstreetmap.org/wiki/Main$_Page$