Network Analysis for Route Inspection

1 Problem Statement

Using Network Analysis to solve the Chinese Postman Problem.

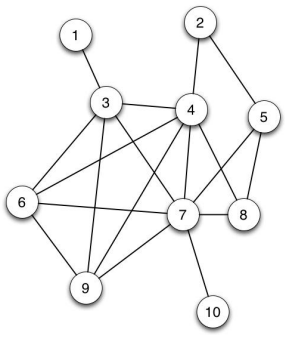
2 Explanation

The Chinese Postman Problem (CPP), also referred to as the Route Inspection or Arc Routing problem, is quite similar to the Travelling Salesman Problem which amounts to finding the shortest route (say, roads) that connects a set of nodes (say, cities). The objective of the CPP is to find the shortest path that covers all the links (roads) on a graph at least once. If this is possible without doubling back on the same road twice, great; That's the ideal scenario and the problem is quite simple. However, if some roads must be traversed more than once, you need some math to find the shortest route that hits every road at least once with the lowest total mileage.

The solution to this CPP problem will be a Eulerian tour: a graph where a cycle that passes through every edge exactly once can be made from a starting node back to itself (without backtracking).

3 Literature Review

A graph consists of a set of objects V called vertices and a set of edges E connecting pairs of vertices.



In the above picture, the circles represent the vertices and lines connecting the circles are edges. Graphs are structures that map relations between objects. The objects are referred to as nodes and the connections between them as edges in. Note that edges and nodes are commonly referred to by several names that generally mean exactly the same thing.

The starting graph is undirected. That is, your edges have no orientation: they are **bi-**directional. For example:

A<--->B==B<--->A.

By contrast, the graph you might create to specify the shortest path to hike every trail could be a directed graph, where the order and direction of edges matters. For example:

A--->B != B--->A.

The graph is also an edge-weighted graph where the distance (in miles) between each pair of adjacent nodes represents the weight of an edge. This is handled as an edge attribute named "distance".

Degree refers to the number of edges incident to (touching) a node. Nodes are referred to as odd-degree nodes when this number is odd and even-degree when even.

The solution to this CPP problem will be a Eulerian tour: a graph where a cycle that passes through every edge exactly once can be made from a starting node back to itself (without backtracking).

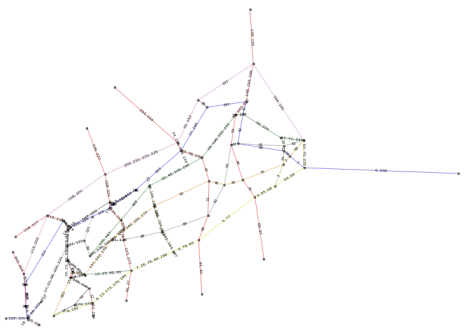
4 Data

The edge list is a simple data structure that we'll use to create the graph. Each row represents a single edge of the graph with some edge attributes.

1. node1 & node2: names of the nodes connected.
2. trail: edge attribute indicating the abbreviated name of the trail for each edge. For example: rs = red square
3. distance: edge attribute indicating trail length in miles.
4. color: trail color used for plotting.
5. estimate: edge attribute indicating whether the edge distance is estimated from eyeballing the trailmap (1=yes, 0=no) as some distances are not provided. This is solely for reference; it is not used for analysis.

5 Deliverable

The graph shows the most optimized route.



6 Evaluation

Not Applicable

7 Data Ingestion

The data was ingested as a CSV.

8 Data Analysis

Edges

Your graph edges are represented by a list of tuples of length 3. The first two elements are the node names linked by the edge. The third is the dictionary of edge attributes.

Nodes

Similarly, your nodes are represented by a list of tuples of length 2. The first element is the node ID, followed by the dictionary of node attributes.

9 Data Munging

Not Required

10 Data Exploration

Not Required

11 Feature Engineering

Not Applicable

12 Modeling

Not Applicable

13 Optimization

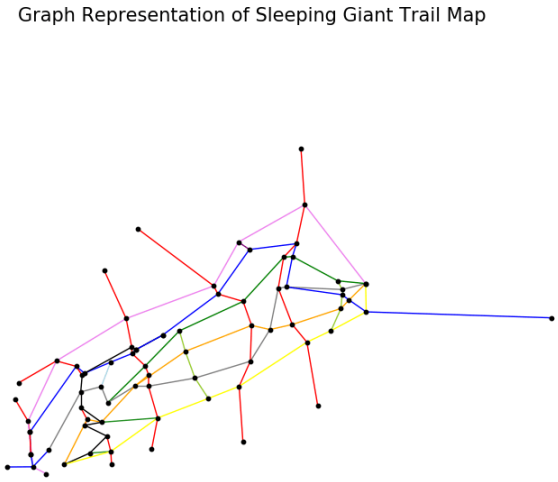
Not Applicable

14 Prediction

Not Applicable

15 Visual Analysis

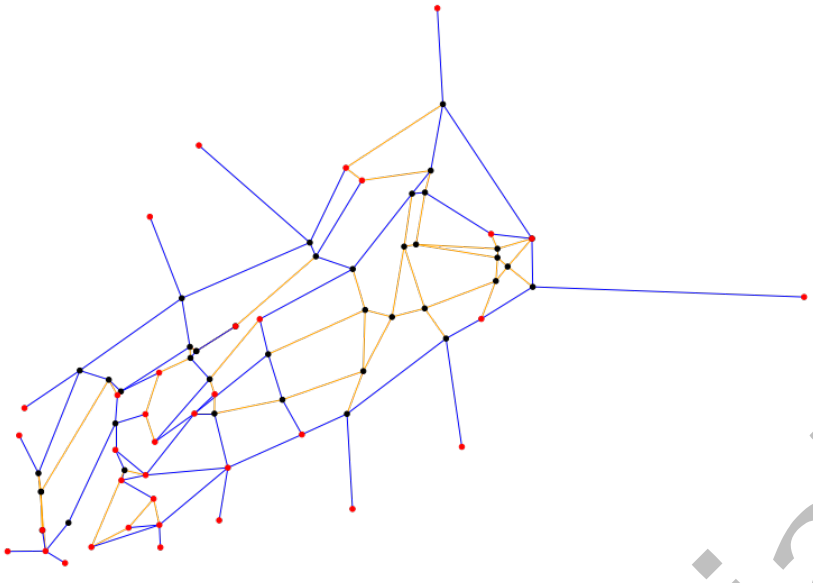
Positions: First you need to manipulate the node positions from the graph into a dictionary. This will allow you to recreate the graph using the same layout as the actual trail map. `Y` is negated to transform the Y-axis origin from the topleft to the bottomleft.
Colors: Now you manipulate the edge colors from the graph into a simple list so that you can visualize the trails by their color.



16 Results

Here you illustrate which edges are walked once (orange) and more than once (blue). This is the "correct" version of the visualization created in **2.4** which showed the naive (as the crow flies) connections between the odd node pairs (red). That is corrected here by tracing the shortest path through edges that actually exist for each pair of odd degree nodes.

If the optimization is any good, these blue lines should represent the least distance possible. Specifically, the minimum distance needed to generate a matching of the odd degree nodes.



Here we plot the original graph (trail map) annotated with the sequence numbers in which we walk the trails per the CPP solution. Multiple numbers indicate trails we must double back on.

You start on the blue trail in the bottom right (0th and the 157th direction).