Graph Optimization with NetworkX in Python

This NetworkX tutorial will show you how to do graph optimization in Python by solving the Chinese Postman Problem in Python.

Sep 2017 · 37 min read

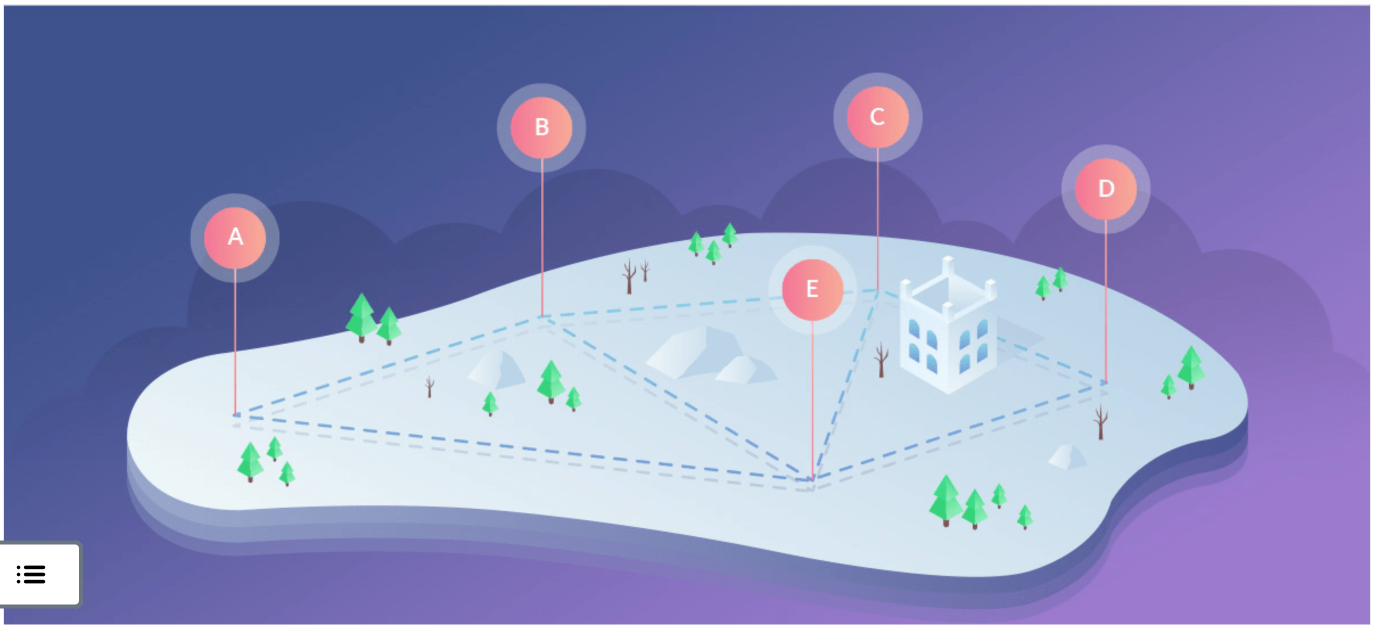
Andrew Brooks

TOPICS

Python

Data Science

Data Visualization



With this tutorial, you'll tackle an established problem in graph theory called the Chinese Postman Problem. There are some components of the algorithm that while conceptually simple, turn out to be computationally rigorous. However, for this tutorial, only some prior knowledge of Python is required: no rigorous math, computer science or graph theory background is needed.

This tutorial will first go over the basic building blocks of graphs (nodes, edges, paths, etc) and solve the problem on a real graph (trail network of a state park) using the NetworkX library in Python. You'll focus on the core concepts and implementation. For the interested reader, further reading on the guts of the optimization are provided.

Motivating Graph Optimization

The Problem

You've probably heard of the Travelling Salesman Problem which amounts to finding the shortest route (say, roads) that connects a set of nodes (say, cities). Although lesser known, the Chinese Postman Problem (CPP), also referred to as the Route Inspection or Arc Routing problem, is quite similar. 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.

Personal Motivation

I had a real-life application for solving this problem: attaining the rank of Giantmaster Marathoner.

What is a Giantmaster? A Giantmaster is one (canine or human) who has hiked every trail of Sleeping Giant State Park in Hamden CT (neighbor to my hometown of Wallingford)... in their lifetime. A Giantmaster Marathoner is one who has hiked all these trails in a single day.

Thanks to the fastidious record keeping of the Sleeping Giant Park Association, the full roster of Giantmasters and their level of Giantmastering can be found here. I have to
t this motivated me quite a bit to kick-start this side-project and get out there to run

the trails. While I myself achieved Giantmaster status in the winter of 2006 when I was a budding young volunteer of the Sleeping Giant Trail Crew (which I was pleased to see recorded in the SG archive), new challenges have since arisen. While the 12-month and 4-season Giantmaster categories are impressive and enticing, they'd also require more travel from my current home (DC) to my formative home (CT) than I could reasonably manage... and they're not as interesting for graph optimization, so Giantmaster Marathon it is!

For another reference, the Sleeping Giant trail map is provided below:

Introducing Graphs

The nice thing about graphs is that the concepts and terminology are generally intuitive. Nonetheless, here's some of the basic lingo:

Graphs are structures that map relations between objects. The objects are referred to as **nodes** and the connections between them as **edges** in this tutorial. Note that edges and nodes are commonly referred to by several names that generally mean exactly the same thing:

```
node == vertex == point
edge == arc == link
```

Hide code explanation

OpenAI

- This code snippet is not actually code, but rather a set of definitions or equivalences.
- It is stating that in the context of graph theory, the terms "node", "vertex", and "point" can be used interchangeably to refer to the same thing.
- Similarly, the terms "edge", "arc", and "link" can also be used interchangeably to refer to the same thing.
- This can be helpful to know when reading or writing code that deals with graphs, as different sources may use different terminology.

Was this helpful? ✓ Yes ✕ No

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). An Euler Tour is also known by several names:

```
Eulerian tour == Eulerian circuit == Eulerian cycle
```

Hide code explanation

OpenAI

• This code snippet is not actually code, but rather a statement in plain English.
• It explains that the terms "Eulerian tour," "Eulerian circuit," and "Eulerian cycle" all refer to the same concept in graph theory.
• Specifically, an Eulerian tour/circuit/cycle is a path in a graph that visits every edge exactly once and returns to the starting vertex.

Was this helpful? ✓ Yes ✗ No

A **matching** is a subset of edges in which no node occurs more than once. A **minimum weight matching** finds the **matching** with the lowest possible summed edge weight.

NetworkX: Graph Manipulation and Analysis

NetworkX is the most popular Python package for manipulating and analyzing graphs. Several packages offer the same basic level of graph manipulation, notably igraph which also has bindings for R and C++. However, I found that NetworkX had the strongest graph algorithms that I needed to solve the CPP.

Installing Packages

If you've done any sort of data analysis in Python or have the Anaconda distribution, my guess is you probably have `pandas` and `matplotlib`. However, you might not have `networkx`. These should be the only dependencies outside the Python Standard Library that you'll need to run through this tutorial. They are easy to install with `pip`:

```
pip install pandas
pip install networkx
pip install matplotlib
```

Hide code explanation

OpenAI

• This code installs three Python packages: pandas, networkx, and matplotlib.
• The **pip** command is a package installer for Python.
• It allows you to easily install and manage Python packages from the command line.
• The **install** option is used to specify that we want to install a package.
• Each package name is specified after the **install** option, separated by a space.
• When this code is executed, the **pip** command will download and install the specified packages and their dependencies.

Was this helpful? ✓ Yes ✕ No

These should be all the packages you'll need for now. `imageio` and `numpy` are imported at the very end to create the GIF animation of the CPP solution. The animation is embedded within this post, so these packages are optional.

```python
import itertools
import copy
import networkx as nx
import pandas as pd
import matplotlib.pyplot as plt
```

Hide code explanation

OpenAI

• This code imports several Python libraries that will be used in the tutorial: **itertools**: provides functions for creating iterators for efficient looping **copy**: provides functions for creating shallow and deep copies of objects **networkx**: provides functions for creating and manipulating graphs and networks **pandas**: provides functions for working with data in tabular form **matplotlib.pyplot**: provides functions for creating visualizations, such as plots and charts.

Was this helpful? ✓ Yes ✕ No

Load Data

Edge List

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

- **node1** & **node2:** names of the nodes connected.
- **trail:** edge attribute indicating the abbreviated name of the trail for each edge. For example: *rs = red square*
- **distance:** edge attribute indicating trail length in miles.
- **color**: trail color used for plotting.
- **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.

```
# Grab edge list data hosted on Gist
edgelist = pd.read_csv('https://gist.githubusercontent.com/brooksandrew/e570c38b
```

Hide code explanation

OpenAI

• This code reads a CSV file from a URL using the pandas library in Python.
• Specifically, it reads the CSV file hosted on the Gist platform at the given URL and assigns the resulting DataFrame to the variable **edgelist**.
• The CSV file contains data related to an edge list for a graph representing the Sleeping Giant mountain range.

Was this helpful? ✓ Yes ✕ No

```
# Preview edgelist
edgelist.head(10)
```

Hide code explanation

OpenAI

• This code is written in Python.
• The code is using the **head()** method to preview the first 10 rows of the **edgelist** dataframe.
• The **edgelist** dataframe is assumed to be previously defined in the code.
• The **head()** method is a pandas method that returns the first n rows of a dataframe.
• In this case, **n** is set to 10.
• This is a useful method to quickly preview the data in a dataframe.

Was this helpful? ✓ Yes ✕ No

	node1	node2	trail	distance	color	estimate
0	rs_end_north	v_rs	rs	0.30	red	0
1	v_rs	b_rs	rs	0.21	red	0
2	b_rs	g_rs	rs	0.11	red	0
3	g_rs	w_rs	rs	0.18	red	0
4	w_rs	o_rs	rs	0.21	red	0
5	o_rs	y_rs	rs	0.12	red	0
6	y_rs	rs_end_south	rs	0.39	red	0
7	rc_end_north	v_rc	rc	0.70	red	0
8	v_rc	b_rc	rc	0.04	red	0
9	b_rc	g_rc	rc	0.15	red	0

Node List

Node lists are usually optional in `networkx` and other graph libraries when edge lists are provided because the node names are provided in the edge list's first two columns. However, in this case, there are some node attributes that we'd like to add: X, Y coordinates of the nodes (trail intersections) so that you can plot your graph with the same layout as the trail map.

I spent an afternoon annotating these manually by tracing over the image with GIMP:

- **id:** name of the node corresponding to **node1** and **node2** in the edge list.
- **X:** horizontal position/coordinate of the node relative to the topleft.
- **Y** vertical position/coordinate of the node relative to the topleft.

Note on Generating the Node & Edge Lists

Creating the node names also took some manual effort. Each node represents an intersection of two or more trails. Where possible, the node is named by *trail1_trail2* where *trail1* precedes *trail2* in alphabetical order.

Things got a little more difficult when the same trails intersected each other more than once. For example, the Orange and White trail. In these cases, I appended a *_2* or *_3* to the node name. For example, you have two distinct node names for the two distinct intersections of Orange and White: *o_w* and *o_w_2*.

This took a lot of trial and error and comparing the plots generated with X,Y coordinates to the real trail map.

```
# Grab node list data hosted on Gist
nodelist = pd.read_csv('https://gist.githubusercontent.com/brooksandrew/f98
```

Hide code explanation

OpenAI

• This code reads data from a CSV file hosted on GitHub's Gist platform and stores it in a pandas DataFrame called **nodelist**.
• The CSV file contains data related to nodes in a network, specifically the Sleeping Giant trail network.
• The **pd.read_csv()** function is used to read the CSV file.
• The function takes the URL of the CSV file as an argument and returns a pandas DataFrame containing the data from the CSV file.
• The **nodelist** variable is then assigned to this DataFrame.
• Overall, this code is useful for accessing and analyzing data stored in a CSV file hosted on GitHub's Gist platform.

Was this helpful? ✓ Yes ✕ No

```
# Preview nodelist
nodelist.head(5)
```

Hide code explanation

OpenAI

• This code is written in Python.
• The code is calling the **head()** method on the **nodelist** object, which is likely a pandas DataFrame or Series.
• The **head()** method returns the first n rows of the DataFrame or Series, where n is the argument passed to the method (in this case, 5).
• So, this code is previewing the first 5 rows of the **nodelist** object.

Was this helpful? ✓ Yes ✕ No

	id	X	Y
0	b_bv	1486	732
1	b_bw	716	1357
2	b_end_east	3164	1111
3	b_end_west	141	1938
4	b_g	1725	771

Create Graph

Now you use the edge list and the node list to create a graph object in `networkx`.

```python
# Create empty graph
g = nx.Graph()
```

Hide code explanation

OpenAI

• This code creates an empty undirected graph using the NetworkX library in Python.
• The **nx.Graph()** function creates a new instance of a graph object, which can be used to add nodes and edges.
• The **g** variable is assigned to this new graph object, allowing the user to manipulate and analyze the graph using various NetworkX functions.

Was this helpful? ✓ Yes ✕ No

Loop through the rows of the edge list and add each edge and its corresponding attributes to graph `g`.

```python
# Add edges and edge attributes
for i, elrow in edgelist.iterrows():
    g.add_edge(elrow[0], elrow[1], attr_dict=elrow[2:].to_dict())
```

Hide code explanation

OpenAI

• This code adds edges and their attributes to a graph object **g**.
• The **iterrows()** method is used to iterate over each row of the **edgelist** DataFrame.
• For each row, the **add_edge()** method is called on the graph object **g** with the two nodes specified in the first two columns of the row as arguments.
• The **attr_dict** parameter is used to add the remaining columns of the row as attributes to the edge.
• The **to_dict()** method is called on a slice of the row (**elrow[2:]**) to convert the remaining columns to a dictionary of attribute-value pairs.
• This dictionary is then passed as the value of the **attr_dict** parameter.
• Overall, this code is a way to add edges and their attributes to a graph object in a loop over a DataFrame.

Was this helpful? ✓ Yes ✕ No

To illustrate what's happening here, let's print the values from the last row in the edge list that got added to graph `g`:

```
# Edge list example
print(elrow[0]) # node1
print(elrow[1]) # node2
print(elrow[2:].to_dict()) # edge attribute dict
```

Hide code explanation

OpenAI

• This code snippet is written in Python and is used to print information about an edge list.
• The first line of code is a comment that indicates that this is an example of an edge list.
• The second line of code prints the first element of the edge list, which represents the first node in the edge.
• The variable **elrow** is assumed to be a Pandas Series object that contains information about an edge.
• The third line of code prints the second element of the edge list, which represents the second node in the edge.
• The fourth line of code uses Pandas' **to_dict()** method to convert the remaining elements of the edge list into a dictionary that represents the attributes of the edge.
• This dictionary is then printed to the console.
• Overall, this code snippet is useful for understanding how to extract information from an edge list and how to work with Pandas Series objects.

Was this helpful? ✓ Yes ✗ No

```
o_gy2
y_gy2
{'color': 'yellowgreen', 'estimate': 0, 'trail': 'gy2', 'distance': 0.12}
```

Hide code explanation

OpenAI

• Unfortunately, the code snippet provided is incomplete and lacks context.
• It appears to be three separate lines of code, but without additional information, it is impossible to determine their purpose or functionality.
• Please provide more context or additional code for me to assist you better.

Was this helpful? ✓ Yes ✗ No

Similarly, you loop through the rows in the node list and add these node attributes.

```python
# Add node attributes
for i, nlrow in nodelist.iterrows():
    g.node[nlrow['id']] = nlrow[1:].to_dict()
```

Hide code explanation

OpenAI

- This code adds attributes to nodes in a graph.
- The **for** loop iterates over each row in the **nodelist** DataFrame using the **iterrows()** method.
- For each row, the code accesses the **id** column and uses it as the key to access the corresponding node in the graph **g**.
- The **to_dict()** method is used to convert the remaining columns in the row (excluding the **id** column) into a dictionary of node attributes.
- These attributes are then assigned to the node in the graph using the **node** attribute of the graph object.
- Overall, this code is adding node attributes to a graph based on the values in a DataFrame.

Was this helpful? ✓ Yes ✕ No

Here's an example from the last row of the node list:

```python
# Node list example
print(nlrow)
```

Hide code explanation

OpenAI

- This code snippet is printing the value of a variable named "nlrow".
- It is assumed that "nlrow" is a list of nodes, which are likely objects that represent elements in a graph or network.
- The code is using the built-in Python function "print()" to display the contents of the "nlrow" list.

Was this helpful? ✓ Yes ✕ No

```
id     y_rt
X       977
Y      1666
Name: 76, dtype: object
```

Hide code explanation

OpenAI

• This code snippet is written in Python and it creates a Pandas DataFrame with two columns: "id" and "y_rt".
• The DataFrame has one row with the values "X" and 977 in the "id" and "y_rt" columns respectively.
• The second row has the values "Y" and 1666 in the "id" and "y_rt" columns respectively.
• The "Name: 76, dtype: object" is just a label for the row and the data type of the row.

Was this helpful? ✓ Yes ✕ No

Inspect Graph

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.

```
# Preview first 5 edges
g.edges(data=True)[0:5]
```

Hide code explanation

OpenAI

• This code is written in Python and is used to preview the first 5 edges of a graph.
• The **g.edges(data=True)** function returns a list of all the edges in the graph **g**, along with any associated data.
• The **data=True** argument ensures that any data associated with the edges is also returned.
• The **[0:5]** at the end of the function call is used to slice the list and return only the first 5 edges.
• This allows the user to preview a small portion of the graph's edges and their associated data.

Was this helpful? ✓ Yes ✕ No

```
[('rs_end_south',
  'y_rs',
  {'color': 'red', 'distance': 0.39, 'estimate': 0, 'trail': 'rs'}),
 ('w_gy2',
  'park_east',
  {'color': 'gray', 'distance': 0.12, 'estimate': 0, 'trail': 'w'}),
 ('w_gy2',
  'g_gy2',
  {'color': 'yellowgreen', 'distance': 0.05, 'estimate': 0, 'trail': 'gy2'}),
 ('w_gy2',
  'b_w',
  {'color': 'gray', 'distance': 0.42, 'estimate': 0, 'trail': 'w'}),
 ('w_gy2',
  'b_gy2',
  {'color': 'yellowgreen', 'distance': 0.03, 'estimate': 1, 'trail': 'gy2'})]
```

Hide code explanation

OpenAI

- This code snippet is a list of tuples, where each tuple contains three elements.
- The first element is a string representing the starting point of a trail, the second element is a string representing the ending point of the trail, and the third element is a dictionary containing information about the trail.
- The information about the trail is stored in the dictionary as key-value pairs.
- The 'color' key represents the color of the trail, the 'distance' key represents the length of the trail, the 'estimate' key represents an estimate of the time it takes to complete the trail, and the 'trail' key represents the name of the trail.
- For example, the first tuple in the list represents a trail starting at 'rs_end_south' and ending at 'y_rs'.
- The trail is colored red, has a length of 0.39, has an estimate of 0, and is named 'rs'.
- This code could be used to represent a map of hiking trails, with each tuple representing a different trail and its associated information.

Was this helpful? ✓ Yes ✕ No

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.

```python
# Preview first 10 nodes
g.nodes(data=True)[0:10]
```

Hide code explanation

OpenAI

- This code is written in Python and it is used to preview the first 10 nodes of a graph.
- The **g.nodes()** function returns a list of all nodes in the graph **g**.
- The **data=True** parameter includes any node attributes in the output.
- The **[0:10]** slice notation is used to select the first 10 nodes from the list.
- Overall, this code snippet returns a preview of the first 10 nodes in the graph **g** along with any associated node attributes.

Was this helpful? ✓ Yes ✕ No

```
[('rs_end_south', {'X': 1865, 'Y': 1598}),
 ('w_gy2', {'X': 2000, 'Y': 954}),
 ('rd_end_south_dupe', {'X': 273, 'Y': 1869}),
 ('w_gy1', {'X': 1184, 'Y': 1445}),
 ('g_rt', {'X': 908, 'Y': 1378}),
 ('v_rd', {'X': 258, 'Y': 1684}),
 ('g_rs', {'X': 1676, 'Y': 775}),
 ('rc_end_north', {'X': 867, 'Y': 618}),
 ('v_end_east', {'X': 2131, 'Y': 921}),
 ('rh_end_south', {'X': 721, 'Y': 1925})]
```

Hide code explanation

OpenAI

- This code snippet is a list of tuples, where each tuple contains a string and a dictionary.
- The string represents a location or point on a map, and the dictionary contains the X and Y coordinates of that location.
- For example, the first tuple in the list is ('rs_end_south', {'X': 1865, 'Y': 1598}), which means that the location named 'rs_end_south' has X coordinate of 1865 and Y coordinate of 1598.
- This code can be used to store and access location data in a program, such as in a mapping or navigation application.

Was this helpful? ✓ Yes ✕ No

Summary Stats

Print out some summary statistics before visualizing the graph.

```
print('# of edges: {}'.format(g.number_of_edges()))
print('# of nodes: {}'.format(g.number_of_nodes()))
```

Hide code explanation

OpenAI

• This code snippet is written in Python and it is using the **networkx** library.
• The first line of code prints the number of edges in the graph **g** using the **number_of_edges()** method of the **networkx** library.
• The **format()** method is used to insert the number of edges into the string.
• The second line of code prints the number of nodes in the graph **g** using the **number_of_nodes()** method of the **networkx** library.
• The **format()** method is used to insert the number of nodes into the string.
• Overall, this code snippet is used to print the number of edges and nodes in a graph.

Was this helpful? ✓ Yes ✕ No

```
# of edges: 123
# of nodes: 77
```

Hide code explanation

OpenAI

• This code snippet is not actually code, but rather comments in the code.
• Comments are used to provide information about the code and are not executed by the computer.
• In this case, the comments indicate the number of edges and nodes in a graph or network.

Was this helpful? ✓ Yes ✕ No

Visualize Graph

Manipulate Colors and Layout

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.

```
# Define node positions data structure (dict) for plotting
node_positions = {node[0]: (node[1]['X'], -node[1]['Y']) for node in g.nodes(dat

# Preview of node_positions with a bit of hack (there is no head/slice method fo
dict(list(node_positions.items())[0:5])
```

Hide code explanation

OpenAI

• This code defines a dictionary called **node_positions** that stores the positions of nodes in a graph.
• The keys of the dictionary are the node IDs, and the values are tuples containing the X and Y coordinates of the node.
• The dictionary is created using a dictionary comprehension that iterates over the nodes in the graph **g**.
• For each node, the comprehension extracts the node ID (**node[0]**) and the X and Y coordinates (**node[1]['X']** and **node[1]['Y']**, respectively) from the node's data dictionary (**node[1]**).
• The X coordinate is stored as the first element of the tuple, and the negative of the Y coordinate is stored as the second element of the tuple.
• The negative Y coordinate is used because in most plotting libraries, the Y axis is inverted relative to the standard Cartesian coordinate system.
• The second line of code uses a bit of a hack to preview the first five items in the **node_positions** dictionary.
• It converts the dictionary to a list of key-value pairs using the **items()** method, takes the first five items using slicing (**[0:5]**), and then converts the resulting list back to a dictionary using the **dict()** constructor.

Was this helpful? ✓ Yes ✕ No

```
{'b_rd': (268, -1744),
 'g_rt': (908, -1378),
 'o_gy1': (1130, -1297),
 'rh_end_tt_2': (550, -1608),
 'rs_end_south': (1865, -1598)}
```

Hide code explanation

OpenAI

• This code is a Python dictionary that contains key-value pairs.
• The keys are strings that represent the names of certain locations, and the values are tuples that contain two integers representing the x and y coordinates of those locations.
• For example, the key 'b_rd' represents a location with x-coordinate 268 and y-coordinate -1744.
• This dictionary can be used to store and access information about different locations in a program.

Was this helpful? ✓ Yes ✕ No

Colors: Now you manipulate the edge colors from the graph into a simple list so that you can visualize the trails by their color.

```
# Define data structure (list) of edge colors for plotting
edge_colors = [e[2]['color'] for e in g.edges(data=True)]

# Preview first 10
edge_colors[0:10]
```

Hide code explanation

OpenAI

• This code is written in Python.
• The code defines a list called **edge_colors** that contains the color of each edge in a graph **g**.
• The color of each edge is obtained from the **color** attribute in the edge's data dictionary, which is accessed using the **data=True** parameter in the **edges()** method.
• The list comprehension **[e[2]['color'] for e in g.edges(data=True)]** iterates over each edge in the graph **g**, extracts the color attribute from the edge's data dictionary, and adds it to the **edge_colors** list.
• The code then previews the first 10 elements of the **edge_colors** list using slicing notation **[0:10]**.

Was this helpful? ✓ Yes ✕ No

```
['red',
 'gray',
 'yellowgreen',
 'gray',
 'yellowgreen',
 'blue',
 'black',
 'yellowgreen',
 'gray',
 'gray']
```

• This code is a simple list of strings.
• The list contains 10 elements, each of which is a string representing a color.
• The colors in the list are 'red', 'gray', 'yellowgreen', 'blue', and 'black'.
• The colors 'gray' and 'yellowgreen' are repeated multiple times in the list.
• There is no specific programming language specified in this code snippet, as it is simply a list of strings that could be used in any programming language that supports lists.

Plot

Now you can make a nice plot that lines up nicely with the Sleeping Giant trail map:

```
plt.figure(figsize=(8, 6))
nx.draw(g, pos=node_positions, edge_color=edge_colors, node_size=10, node_color=
plt.title('Graph Representation of Sleeping Giant Trail Map', size=15)
plt.show()
```

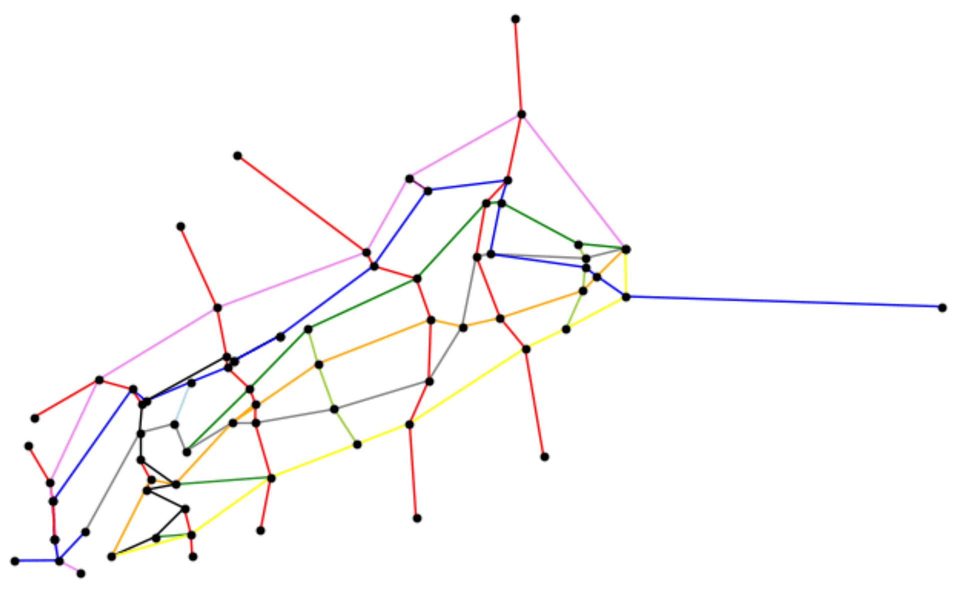
Hide code explanation

OpenAI

- This code uses the **matplotlib** library to create a visualization of a graph using the **nx.draw()** function from the **networkx** library.
- First, a figure is created with a size of 8 inches by 6 inches using **plt.figure(figsize=(8, 6))**.
- Then, the **nx.draw()** function is called with the graph **g** as the first argument, and the positions of the nodes on the graph as the **pos** argument.
- The **edge_color** argument specifies the color of the edges, and the **node_size** argument sets the size of the nodes.
- The **node_color** argument sets the color of the nodes to black.
- After that, a title is added to the plot using **plt.title('Graph Representation of Sleeping Giant Trail Map', size=15)**.
- Finally, the plot is displayed using **plt.show()**.

Was this helpful? ✓ Yes ✕ No

Graph Representation of Sleeping Giant Trail Map



This graph representation obviously doesn't capture all the trails' bends and squiggles, however not to worry: these are accurately captured in the edge `distance` attribute which is used for computation. The visual does capture distance between nodes (trail intersections) as the crow flies, which appears to be a decent approximation.

Overview of CPP Algorithm

OK, so now that you've defined some terms and created the graph, how do you find the shortest path through it?

Solving the Chinese Postman Problem is quite simple conceptually:

1. Find all nodes with odd degree (very easy).
   (Find all trail intersections where the number of trails touching that intersection is an odd number)

2. Add edges to the graph such that all nodes of odd degree are made even. These added edges must be duplicates from the original graph (we'll assume no bushwhacking for this problem). The set of edges added should sum to the minimum distance possible (hard...np-hard to be precise).
   (In simpler terms, minimize the amount of double backing on a route that hits every trail)

3. Given a starting point, find the Eulerian tour over the augmented dataset (moderately easy).
   (Once we know which trails we'll be double backing on, actually calculate the route from beginning to end)

Assumptions and Simplifications

While a shorter and more precise path could be generated by relaxing the assumptions below, this would add complexity beyond the scope of this tutorial which focuses on the CPP.

Assumption 1: Required trails only

u can see from the trail map above, there are roads along the borders of the park that could be used to connect trails, particularly the red trails. There are also some trails (Horseshoe and unmarked blazes) which are not required per the Giantmaster log, but could be helpful to prevent lengthy double backing. The inclusion of optional trails is actually an established variant of the CPP called the Rural Postman Problem. We ignore optional trails in this tutorial and focus on required trails only.

Assumption 2: Uphill == downhill

The CPP assumes that the cost of walking a trail is equivalent to its distance, regardless of which direction it is walked. However, some of these trails are rather hilly and will require more energy to walk up than down. Some metric that combines both distance and elevation change over a directed graph could be incorporated into an extension of the CPP called the Windy Postman Problem.

Assumption 3: No parallel edges (trails)

While possible, the inclusion of parallel edges (multiple trails connecting the same two nodes) adds complexity to computation. Luckily this only occurs twice here (Blue <=> Red Diamond and Blue <=> Tower Trail). This is addressed by a bit of a hack to the edge list: duplicate nodes are included with a *_dupe* suffix to capture every trail while maintaining uniqueness in the edges. The CPP implementation in the postman_problems package I wrote robustly handles parallel edges in a more elegant way if you'd like to solve the CPP on your own graph with many parallel edges.

CPP Step 1: Find Nodes of Odd Degree

This is a pretty straightforward counting computation. You see that 36 of the 76 nodes have odd degree. These are mostly the dead-end trails (degree 1) and intersections of 3 trails. There are a handful of degree 5 nodes.

```python
# Calculate list of nodes with odd degree
nodes_odd_degree = [v for v, d in g.degree_iter() if d % 2 == 1]

# Preview
nodes_odd_degree[0:5]
```

Hide code explanation

OpenAI

- This code calculates a list of nodes in a graph **g** that have an odd degree.
- The **degree_iter()** method returns an iterator over the degrees of all nodes in the graph.
- The **for** loop iterates over each node and its degree, and the **if** statement checks if the degree is odd (**d % 2 == 1**).
- If the degree is odd, the node is added to the **nodes_odd_degree** list using a list comprehension.
- Finally, the first five elements of the **nodes_odd_degree** list are previewed using slicing.

Was this helpful? ✓ Yes ✕ No

```
['rs_end_south', 'rc_end_north', 'v_end_east', 'rh_end_south', 'b_end_east
```

Hide code explanation

OpenAI

- This code is a simple Python list containing five string elements.
- The elements are enclosed in square brackets and separated by commas.
- The strings represent different types of endpoints, such as 'rs_end_south' and 'rc_end_north'.
- The list can be accessed and manipulated using various list methods in Python.

Was this helpful? ✓ Yes ✕ No

```
# Counts
print('Number of nodes of odd degree: {}'.format(len(nodes_odd_degree)))
print('Number of total nodes: {}'.format(len(g.nodes())))
```

Hide code explanation

OpenAI

• This code snippet is written in Python and is used to print the number of nodes in a graph that have an odd degree and the total number of nodes in the graph.
• The first line uses the **print()** function to display a message that includes the number of nodes with an odd degree.
• The **len()** function is used to determine the length of the **nodes_odd_degree** list, which contains the nodes with an odd degree.
• The **format()** method is used to insert the length of the list into the message.
• The second line also uses the **print()** function to display a message that includes the total number of nodes in the graph.
• The **len()** function is used to determine the length of the **g.nodes()** list, which contains all the nodes in the graph.
• The **format()** method is used to insert the length of the list into the message.
• Overall, this code snippet is used to provide information about the number of nodes in a graph with an odd degree and the total number of nodes in the graph.

Was this helpful? ✓ Yes ✕ No

```
Number of nodes of odd degree: 36
Number of total nodes: 77
```

Hide code explanation

OpenAI

• This code snippet does not contain any programming language.
• It simply displays two pieces of information: the number of nodes in a graph that have an odd degree (i.e., an odd number of edges connected to them), and the total number of nodes in the graph.

Was this helpful? ✓ Yes ✕ No

CPP Step 2: Find Min Distance Pairs

This is really the meat of the problem. You'll break it down into 5 parts:

1. Compute all possible pairs of odd degree nodes.
2. Compute the shortest path between each node pair calculated in **1.**
3. Create a **complete graph** connecting every node pair in **1.** with shortest path distance attributes calculated in **2.**
4. Compute a **minimum weight matching** of the graph calculated in **3.**
   (This boils down to determining how to pair the odd nodes such that the sum of the distance between the pairs is as small as possible).
5. Augment the original graph with the shortest paths between the node pairs calculated in **4.**

Step 2.1: Compute Node Pairs

You use the `itertools combination` function to compute all possible pairs of the odd degree nodes. Your graph is undirected, so we don't care about order: For example, `(a,b) == (b,a)`.

```
# Compute all pairs of odd nodes. in a list of tuples
odd_node_pairs = list(itertools.combinations(nodes_odd_degree, 2))

# Preview pairs of odd degree nodes
odd_node_pairs[0:10]
```

Hide code explanation

OpenAI

• This code uses the **itertools** module in Python to compute all pairs of odd nodes in a list of tuples.
• First, the **itertools.combinations()** function is used to generate all possible combinations of two elements from the **nodes_odd_degree** list, which contains the nodes with odd degrees.
• The resulting pairs are stored in the **odd_node_pairs** list.
• Then, the first 10 pairs of odd degree nodes are previewed using list slicing (**odd_node_pairs[0:10]**).
• Overall, this code is useful for identifying pairs of nodes with odd degrees in a graph, which can be important for certain graph algorithms and analyses.

Was this helpful? ✓ Yes ✗ No

```
[('rs_end_south', 'rc_end_north'),
 ('rs_end_south', 'v_end_east'),
 ('rs_end_south', 'rh_end_south'),
 ('rs_end_south', 'b_end_east'),
 ('rs_end_south', 'b_bv'),
 ('rs_end_south', 'rt_end_south'),
 ('rs_end_south', 'o_rt'),
 ('rs_end_south', 'y_rt'),
 ('rs_end_south', 'g_gy2'),
 ('rs_end_south', 'b_tt_3')]
```

Hide code explanation

OpenAI

• This code is a list of tuples, where each tuple contains two strings.
• The strings represent the names of different variables or objects.
• The purpose of this code is not clear without additional context, but it appears to be a collection of related variables or objects that are being used in some way.

Was this helpful? ✓ Yes ✕ No

```
# Counts
print('Number of pairs: {}'.format(len(odd_node_pairs)))
```

Hide code explanation

OpenAI

• This code snippet is written in Python.
• The **print()** function is used to display the output on the console.
• The output is a string that contains the text "Number of pairs: " and the length of the **odd_node_pairs** list.
• The **len()** function is used to get the length of the **odd_node_pairs** list.
• The length is then passed as an argument to the **format()** method of the string.
• The curly braces {} in the string act as placeholders for the value of the length.
• So, the output of this code will be a string that displays the number of pairs in the **odd_node_pairs** list.

Was this helpful? ✓ Yes ✕ No

```
Number of pairs: 630
```

Hide code explanation

OpenAI

- This code snippet is not actually code, but rather a statement indicating that there are 630 pairs.
- It does not require any programming language to understand.

Was this helpful? ✓ Yes ✕ No

Let's confirm that this number of pairs is correct with a the combinatoric below. Luckily, you only have 630 pairs to worry about. Your computation time to solve this CPP example is trivial (a couple seconds).

However, if you had 3,600 odd node pairs instead, you'd have ~6.5 million pairs to optimize. That's a ~10,000x increase in output given a 100x increase in input size.

\begin{equation*} \#\;of\;pairs = n\;choose\;r = {n \choose r} = \frac{n!}{r!(n-r)!} = \frac{36!}{2! (36-2)!} = 630 \end{equation*}

Step 2.2: Compute Shortest Paths between Node Pairs

This is the first step that involves some real computation. Luckily `networkx` has a convenient implementation of **Dijkstra's algorithm** to compute the shortest path between two nodes. You apply this function to every pair (all 630) calculated above in `odd_node_pairs`.

```python
def get_shortest_paths_distances(graph, pairs, edge_weight_name):
    """Compute shortest distance between each pair of nodes in a graph.  Return
    distances = {}
    for pair in pairs:
        distances[pair] = nx.dijkstra_path_length(graph, pair[0], pair[1], weigh
    return distances
```

Hide code explanation

OpenAI

• This code defines a function called **get_shortest_paths_distances** that takes in three arguments: **graph**, **pairs**, and **edge_weight_name**.
• The **graph** argument is a networkx graph object, **pairs** is a list of tuples representing pairs of nodes in the graph, and **edge_weight_name** is the name of the edge attribute that represents the weight of each edge in the graph.
• The function then initializes an empty dictionary called **distances**.
• It then loops through each pair of nodes in the **pairs** list and uses the **nx.dijkstra_path_length** function from the networkx library to compute the shortest path distance between the two nodes in the graph, using the **edge_weight_name** as the weight for each edge.
• The resulting distance is then stored in the **distances** dictionary with the node pair tuple as the key.
• Finally, the function returns the **distances** dictionary, which contains the shortest path distances between each pair of nodes in the graph.

Was this helpful? ✓ Yes ✕ No

```python
# Compute shortest paths.  Return a dictionary with node pairs keys and a
odd_node_pairs_shortest_paths = get_shortest_paths_distances(g, odd_node_pairs,

# Preview with a bit of hack (there is no head/slice method for dictionaries).
dict(list(odd_node_pairs_shortest_paths.items())[0:10])
```

Hide code explanation

OpenAI

- This code computes the shortest paths between pairs of nodes in a graph.
- The **get_shortest_paths_distances** function takes three arguments: the graph **g**, a list of node pairs **odd_node_pairs**, and the attribute to use for distance calculation (**'distance'** in this case).
- It returns a dictionary with node pairs as keys and the shortest path distance as the value.
- The second line of code uses a bit of a hack to preview the first 10 items of the dictionary.
- It converts the dictionary to a list of key-value pairs, takes the first 10 items, and then converts the list back to a dictionary.
- This is necessary because dictionaries don't have a **head** or **slice** method like lists do.

Was this helpful? ✓ Yes ✕ No

```
{('b_bv', 'y_gy1'): 1.22,
 ('b_bw', 'rc_end_south'): 1.35,
 ('b_end_east', 'b_bw'): 3.0400000000000005,
 ('b_end_east', 'rd_end_north'): 3.83,
 ('g_gy1', 'nature_end_west'): 0.9900000000000001,
 ('o_rt', 'y_gy1'): 0.53,
 ('rc_end_north', 'rd_end_south'): 2.21,
 ('rc_end_north', 'rs_end_north'): 1.79,
 ('rs_end_north', 'o_tt'): 2.0999999999999996,
 ('w_bw', 'rd_end_north'): 1.02}
```

Hide code explanation

OpenAI

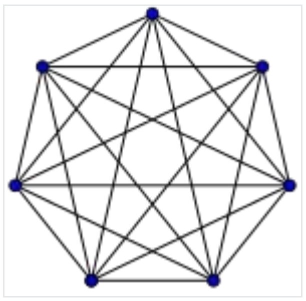
• This code snippet is a Python dictionary that contains key-value pairs.
• The keys are tuples of two strings, representing the names of two locations in a transportation network.
• The values are floating-point numbers, representing the distance or cost between the two locations.
• For example, the key **('b_bv', 'y_gy1')** represents the distance between locations "b_bv" and "y_gy1", and the value **1.22** represents the distance or cost between those two locations.
• This dictionary can be used to represent a graph or network, where the keys represent nodes and the values represent edges between those nodes.

Was this helpful? ✓ Yes ✕ No

Step 2.3: Create Complete Graph

A complete graph is simply a graph where every node is connected to every other node by a unique edge.

Here's a basic example from Wikipedia of a 7 node complete graph with 21 (7 choose 2) edges:



The graph you create below has 36 nodes and 630 edges with their corresponding edge
t (distance).

create_complete_graph is defined to calculate it. The flip_weights parameter is used to transform the distance to the weight attribute where smaller numbers reflect large distances and high numbers reflect short distances. This sounds a little counter intuitive, but is necessary for Step **2.4** where you calculate the minimum weight matching on the complete graph.

Ideally you'd calculate the minimum weight matching directly, but NetworkX only implements a max_weight_matching function which maximizes, rather than minimizes edge weight. We hack this a bit by negating (multiplying by -1) the distance attribute to get weight. This ensures that order and scale by distance are preserved, but reversed.

```
def create_complete_graph(pair_weights, flip_weights=True):
    """
    Create a completely connected graph using a list of vertex pairs and the sho
    Parameters:
        pair_weights: list[tuple] from the output of get_shortest_paths_distance
        flip_weights: Boolean. Should we negate the edge attribute in pair_weigh
    """
    g = nx.Graph()
    for k, v in pair_weights.items():
        wt_i = - v if flip_weights else v
        g.add_edge(k[0], k[1], attr_dict={'distance': v, 'weight': wt_i})
    return g
```

Hide code explanation

OpenAI

• This code defines a function called **create_complete_graph** that takes in a list of vertex pairs and their shortest path distances as **pair_weights** and a boolean **flip_weights** that determines whether the edge attribute in **pair_weights** should be negated or not.
• The function creates an empty graph using the **nx.Graph()** method from the NetworkX library.
• It then iterates through each key-value pair in **pair_weights** using a for loop.
• For each pair, it calculates the weight of the edge by negating the value of the distance if **flip_weights** is True, or using the original value if **flip_weights** is False.
• Finally, the function adds an edge to the graph using the **add_edge()** method from NetworkX, with the two vertices as the first two arguments, and an attribute dictionary containing the distance and weight as the third argument.
• The function returns the completed graph.

Was this helpful? ✓ Yes ✕ No

```
# Generate the complete graph
g_odd_complete = create_complete_graph(odd_node_pairs_shortest_paths, flip_weigh

# Counts
print('Number of nodes: {}'.format(len(g_odd_complete.nodes())))
print('Number of edges: {}'.format(len(g_odd_complete.edges())))
```

Hide code explanation

OpenAI

• This code snippet generates a complete graph using the function **create_complete_graph()** and then prints the number of nodes and edges in the graph using the **len()** function and the **nodes()** and **edges()** methods of the graph object.
• The **create_complete_graph()** function takes in a dictionary of shortest paths between pairs of nodes and creates a graph where each node is connected to every other node.
• The **flip_weights** parameter is set to **True**, which means that the weights of the edges in the graph will be the inverse of the shortest path distances.
• The **len()** function is used to get the number of nodes and edges in the graph, and the **nodes()** and **edges()** methods are used to get the list of nodes and edges in the graph, respectively.
• The output of this code will be the number of nodes and edges in the complete graph.

Was this helpful? ✓ Yes ✕ No

```
Number of nodes: 36
Number of edges: 630
```

Hide code explanation

OpenAI

• This code snippet does not have any programming language specified.
• It simply prints out the number of nodes and edges in a graph.
• The number of nodes is 36 and the number of edges is 630.
• This information is useful in analyzing the structure and complexity of the graph.

Was this helpful? ✓ Yes ✕ No

For a visual prop, the fully connected graph of odd degree node pairs is plotted below. Note that you preserve the X, Y coordinates of each node, but the edges do not necessarily represent actual trails. For example, two nodes could be connected by a single edge in this graph, but the shortest path between them could be 5 hops through even degree nodes (not shown here).

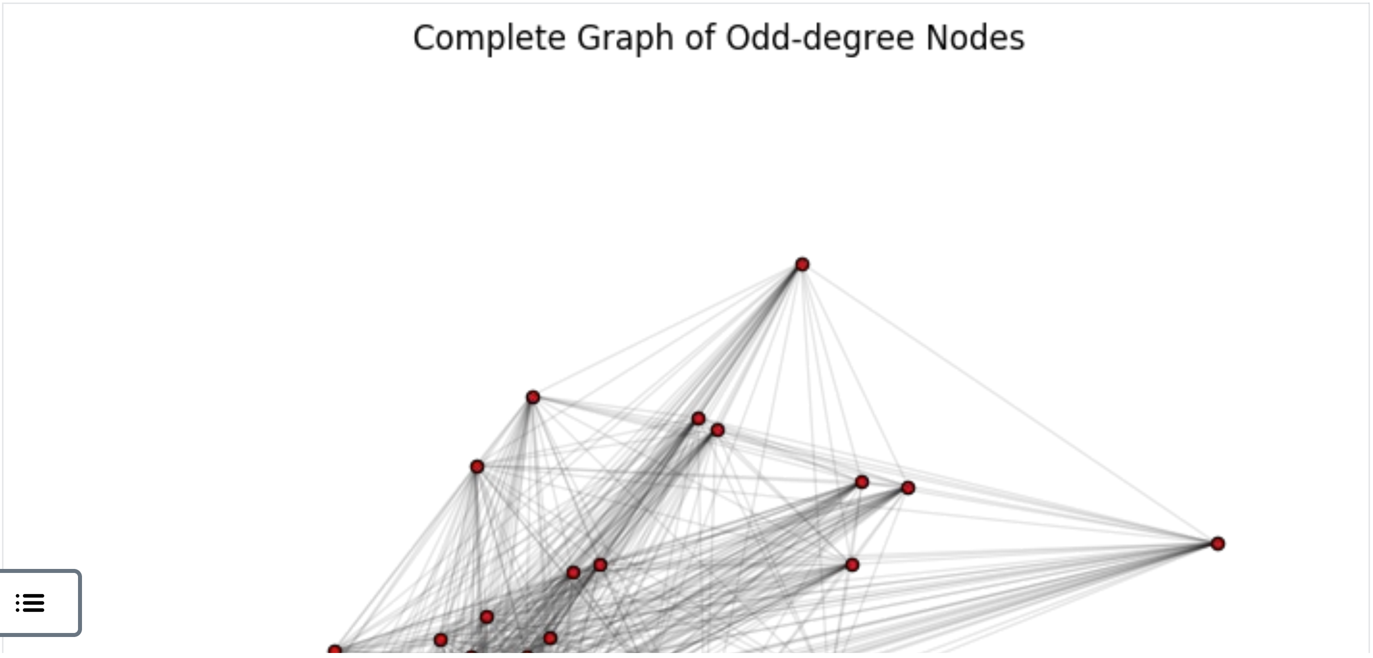
```
# Plot the complete graph of odd-degree nodes
plt.figure(figsize=(8, 6))
pos_random = nx.random_layout(g_odd_complete)
nx.draw_networkx_nodes(g_odd_complete, node_positions, node_size=20, node_color=
nx.draw_networkx_edges(g_odd_complete, node_positions, alpha=0.1)
plt.axis('off')
plt.title('Complete Graph of Odd-degree Nodes')
plt.show()
```

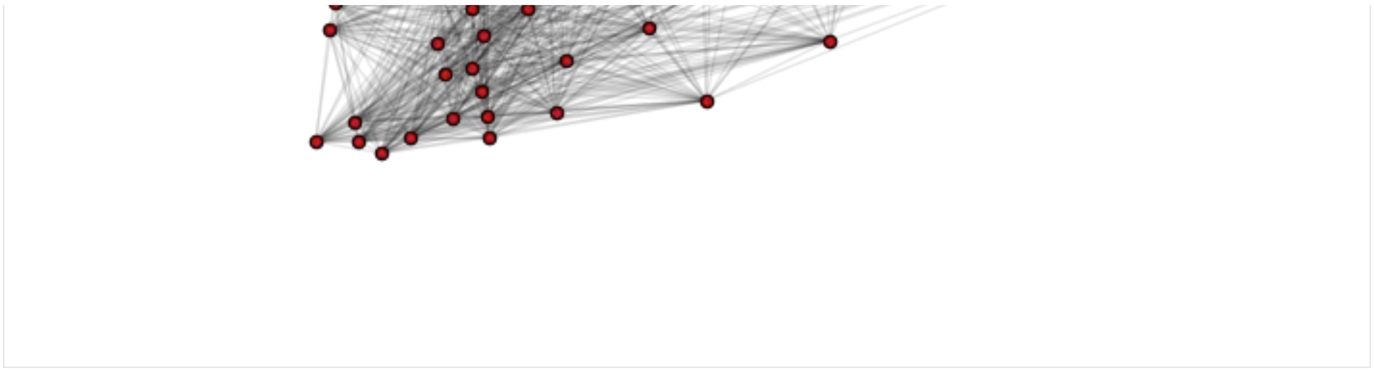
Hide code explanation

OpenAI

- This code uses the NetworkX library to plot a complete graph of odd-degree nodes.
- First, a figure is created with a size of 8 by 6 using **plt.figure(figsize=(8, 6))**.
- Then, a random layout is generated for the graph using **pos_random = nx.random_layout(g_odd_complete)**.
- Next, the nodes of the graph are drawn using **nx.draw_networkx_nodes(g_odd_complete, node_positions, node_size=20, node_color="red")**.
- The **node_positions** variable is not defined in this code snippet, so it must have been defined earlier in the code.
- After that, the edges of the graph are drawn using **nx.draw_networkx_edges(g_odd_complete, node_positions, alpha=0.1)**.
- The **alpha** parameter sets the transparency of the edges.
- The **plt.axis('off')** line removes the axis labels and ticks from the plot.
- Finally, the title of the plot is set using **plt.title('Complete Graph of Odd-degree Nodes')** and the plot is displayed using **plt.show()**.

Was this helpful? ✓ Yes ✕ No





Step 2.4: Compute Minimum Weight Matching

This is the most complex step in the CPP. You need to find the odd degree node pairs whose combined sum (of distance between them) is as small as possible. So for your problem, this boils down to selecting the optimal 18 edges (36 odd degree nodes / 2) from the hairball of a graph generated in **2.3**.

Both the implementation and intuition of this optimization are beyond the scope of this tutorial... like 800+ lines of code and a body of academic literature beyond this scope.

However, a quick aside for the interested reader:

A huge thanks to Joris van Rantwijk for writing the orginal implementation on his blog way back in 2008. I stumbled into the problem a similar way with the same intention as Joris. From Joris's 2008 post:

Since I did not find any Perl implementations of maximum weighted matching, I lightly decided to write some code myself. It turned out that I had underestimated the problem, but by the time I realized my mistake, I was so obsessed with the problem that I refused to give up.

However, I did give up. Luckily Joris did not.

This Maximum Weight Matching has since been folded into and maintained within the NetworkX package. Another big thanks to the 10+ contributors on GitHub who have maintained this hefty codebase.

This is a hard and intensive computation. The first breakthrough in 1965 proved that the Maximum Matching problem could be solved in polynomial time. It was published by Jack Edmonds with perhaps one of the most beautiful academic paper titles ever: "Paths, trees, and flowers" [1]. A body of literature has since built upon this work, improving the optimization procedure. The code implemented in the NetworkX function max_weight_matching is based on Galil, Zvi (1986) [2] which employs an O(n^3) time ithm.

```
# Compute min weight matching.
# Note: max_weight_matching uses the 'weight' attribute by default as the attrib
odd_matching_dupes = nx.algorithms.max_weight_matching(g_odd_complete, True)

print('Number of edges in matching: {}'.format(len(odd_matching_dupes)))
```

Hide code explanation

OpenAI

• This code computes the minimum weight matching for a graph using the NetworkX library in Python.
• The **nx.algorithms.max_weight_matching** function is used to find the maximum weight matching in the graph **g_odd_complete**, with the **True** parameter indicating that the function should allow duplicate edges in the matching.
• Since we want to find the minimum weight matching, we can use the maximum weight matching and negate the weights of the edges.
• The **odd_matching_dupes** variable stores the resulting matching as a dictionary of edges.
• Finally, the number of edges in the matching is printed using the **len** function and formatted into a string.

Was this helpful? ✓ Yes ✕ No

```
Number of edges in matching: 36
```

Hide code explanation

OpenAI

• This code snippet is simply displaying the number of edges in a matching.
• It does not involve any programming language as it is just a printed output.
• The context of the code is not provided, so it is difficult to provide further explanation.

Was this helpful? ✓ Yes ✕ No

The matching output (`odd_matching_dupes`) is a dictionary. Although there are 36 edges in this matching, you only want 18. Each edge-pair occurs twice (once with node 1 as the key and a second time with node 2 as the key of the dictionary).

```
# Preview of matching with dupes
odd_matching_dupes
```

Hide code explanation

OpenAI

• This code snippet is simply displaying the contents of a variable called "odd_matching_dupes".
• It is likely that this variable contains some data related to matching with duplicates, and the purpose of this code is to provide a preview of that data.
• The "#" symbol at the beginning of the line indicates that this is a comment, and the code itself does not actually do anything other than display the contents of the variable.

Was this helpful? ✓ Yes ✕ No

```
{'b_bv': 'v_bv',
 'b_bw': 'rh_end_tt_1',
 'b_end_east': 'g_gy2',
 'b_end_west': 'rd_end_south',
 'b_tt_3': 'rt_end_north',
 'b_v': 'v_end_west',
 'g_gy1': 'rc_end_north',
 'g_gy2': 'b_end_east',
 'g_w': 'w_bw',
 'nature_end_west': 'o_y_tt_end_west',
 'o_rt': 'o_w_1',
 'o_tt': 'rh_end_tt_2',
 'o_w_1': 'o_rt',
 'o_y_tt_end_west': 'nature_end_west',
 'rc_end_north': 'g_gy1',
 'rc_end_south': 'y_gy1',
 'rd_end_north': 'rh_end_north',
 'rd_end_south': 'b_end_west',
 'rh_end_north': 'rd_end_north',
 'rh_end_south': 'y_rh',
 'rh_end_tt_1': 'b_bw',
 'rh_end_tt_2': 'o_tt',
 'rh_end_tt_3': 'rh_end_tt_4',
 'rh_end_tt_4': 'rh_end_tt_3',
 'rs_end_north': 'v_end_east',
 'rs_end_south': 'y_gy2',
 'rt_end_north': 'b_tt_3',
 'rt_end_south': 'y_rt',
 'v_bv': 'b_bv',
 'v_end_east': 'rs_end_north',
 'v_end_west': 'b_v',
 'w_bw': 'g_w',
 'y_gy1': 'rc_end_south',
 'y_gy2': 'rs_end_south',
 'y_rh': 'rh_end_south',
 'y_rt': 'rt_end_south'}
```

Hide code explanation

OpenAI

• This code snippet is a dictionary in Python.
• It contains key-value pairs where each key is a string representing a location in a transportation network and each value is a string representing the location that can be reached from the key tion.
r example, the key 'b_bv' has a value of 'v_bv', which means that location 'b_bv' can be

reached from location 'v_bv'.
• This dictionary can be used to represent a graph where the keys are nodes and the values are edges connecting the nodes.
• This can be useful for pathfinding algorithms or other graph-related operations.

Was this helpful? Yes No

You convert this dictionary to a list of tuples since you have an undirected graph and order does not matter. Removing duplicates yields the unique 18 edge-pairs that cumulatively sum to the least possible distance.

```
# Convert matching to list of deduped tuples
odd_matching = list(pd.unique([tuple(sorted([k, v])) for k, v in odd_matching_du

# Counts
print('Number of edges in matching (deduped): {}'.format(len(odd_matching)))
```

Hide code explanation

OpenAI

• This code snippet is written in Python and uses the pandas library.
• The code is converting a dictionary called **odd_matching_dupes** into a list of deduplicated tuples called **odd_matching**.
• The first line of code uses a list comprehension to iterate over the key-value pairs in **odd_matching_dupes**.
• For each pair, it creates a sorted tuple of the key and value using **tuple(sorted([k, v]))**.
• This ensures that the order of the key-value pairs doesn't matter and that duplicates are removed.
• The **pd.unique()** function is then used to remove any remaining duplicates in the list of tuples.
• The resulting list of tuples is assigned to the variable **odd_matching**.
• The second line of code simply prints the length of the **odd_matching** list, which represents the number of edges in the matching after deduplication.

Was this helpful? ✓ Yes ✕ No

```
Number of edges in matching (deduped): 18
```

Hide code explanation

OpenAI

• This code snippet is simply outputting the number of edges in a matching, which has been deduplicated.
• It does not include any programming language as it is just a printed statement.
• The context of the code is not provided, so it is difficult to provide further explanation.

Was this helpful? ✓ Yes ✕ No

```
# Preview of deduped matching
odd_matching
```

Hide code explanation

OpenAI

• This code is written in Python and it is simply displaying the contents of a variable named "odd_matching".
• The "#" symbol indicates a comment, so this line is not actually executing any code but rather providing a description of what is happening.
• The variable "odd_matching" likely contains some data that has been deduplicated and matched in some way, and this line is providing a preview of that data for the user to see.

Was this helpful? ✓ Yes ✕ No

```
[('rs_end_south', 'y_gy2'),
 ('b_end_west', 'rd_end_south'),
 ('b_bv', 'v_bv'),
 ('rh_end_tt_3', 'rh_end_tt_4'),
 ('b_bw', 'rh_end_tt_1'),
 ('o_tt', 'rh_end_tt_2'),
 ('g_w', 'w_bw'),
 ('b_end_east', 'g_gy2'),
 ('nature_end_west', 'o_y_tt_end_west'),
 ('g_gy1', 'rc_end_north'),
 ('o_rt', 'o_w_1'),
 ('rs_end_north', 'v_end_east'),
 ('rc_end_south', 'y_gy1'),
 ('rh_end_south', 'y_rh'),
 ('rt_end_south', 'y_rt'),
 ('b_tt_3', 'rt_end_north'),
 ('rd_end_north', 'rh_end_north'),
 ('b_v', 'v_end_west')]
```

Hide code explanation

OpenAI

• This code snippet is a list of tuples, where each tuple contains two strings.
• The first string in each tuple represents a location or object in a system, while the second string represents a connection or relationship between two locations or objects.
• For example, the first tuple ('rs_end_south', 'y_gy2') indicates that there is a connection between the location 'rs_end_south' and the object 'y_gy2'.
• This code could be used to represent a network or graph of connections between different locations or objects in a system.

Was this helpful? ✓ Yes ✕ No

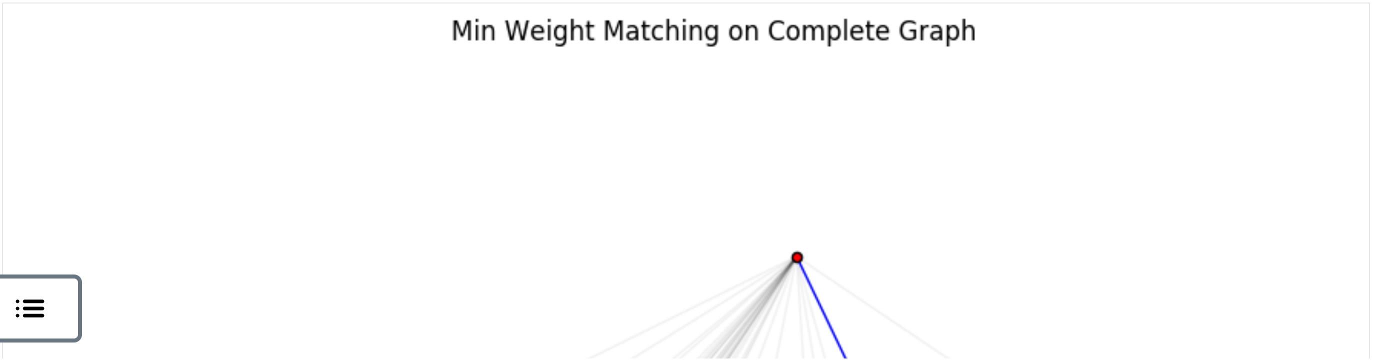
Let's visualize these pairs on the complete graph plotted earlier in step **2.3.** As before, while the node positions reflect the true graph (trail map) here, the edge distances shown (blue lines) are as the crow flies. The actual shortest route from one node to another could involve multiple edges that twist and turn with considerably longer distance.

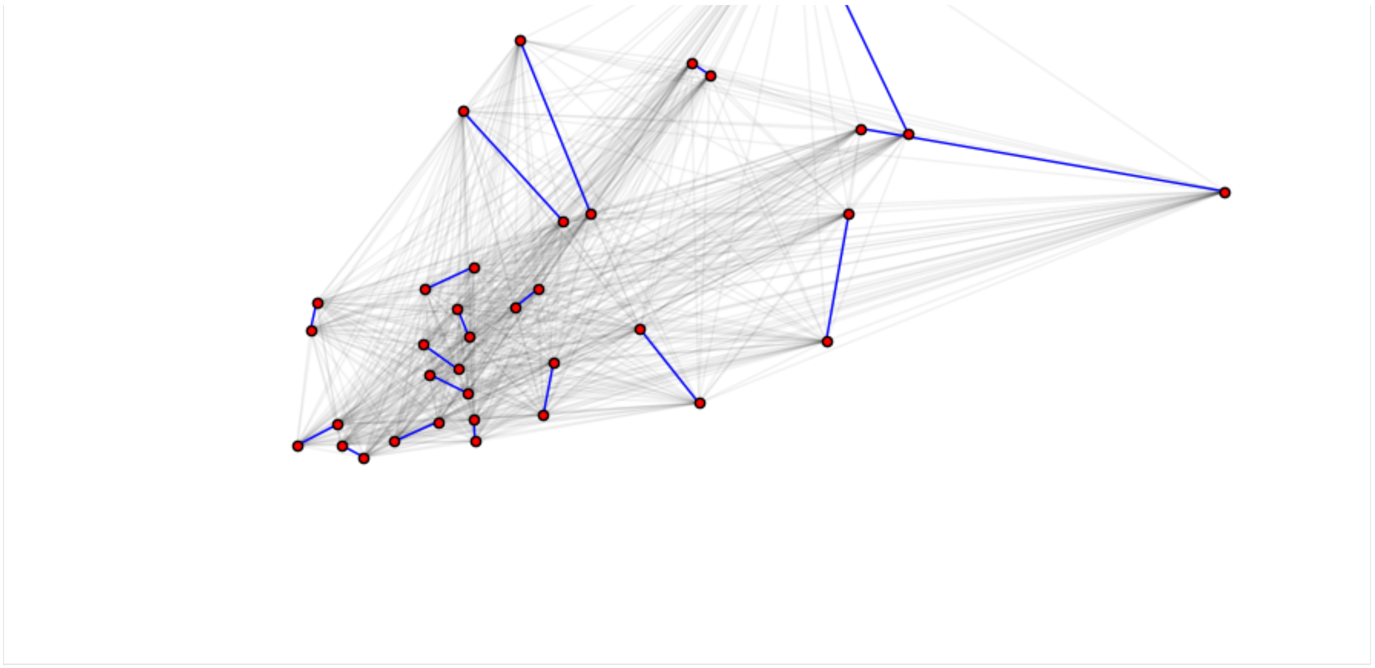
```
plt.figure(figsize=(8, 6))

# Plot the complete graph of odd-degree nodes
nx.draw(g_odd_complete, pos=node_positions, node_size=20, alpha=0.05)

# Create a new graph to overlay on g_odd_complete with just the edges from the m
g_odd_complete_min_edges = nx.Graph(odd_matching)
nx.draw(g_odd_complete_min_edges, pos=node_positions, node_size=20, edge_color='

plt.title('Min Weight Matching on Complete Graph')
plt.show()
```

Hide code explanation

• This code creates a visualization of a minimum weight matching on a complete graph of odd-degree nodes.
• First, a new figure is created with a size of 8 by 6 using **plt.figure(figsize=(8, 6))**.
• Then, the complete graph of odd-degree nodes is plotted using **nx.draw(g_odd_complete, pos=node_positions, node_size=20, alpha=0.05)**.
• **g_odd_complete** is the graph object and **node_positions** is a dictionary of node positions.
• The **node_size** parameter sets the size of the nodes and **alpha** sets the transparency of the nodes.
• Next, a new graph is created with just the edges from the minimum weight matching using **g_odd_complete_min_edges = nx.Graph(odd_matching)**.
• **odd_matching** is a list of edges representing the minimum weight matching.
• Finally, the minimum weight matching graph is overlayed on the complete graph using **nx.draw(g_odd_complete_min_edges, pos=node_positions, node_size=20, edge_color='blue', node_color='red')**.
• The **edge_color** parameter sets the color of the edges and **node_color** sets the color of the nodes.
• The title of the plot is set to 'Min Weight Matching on Complete Graph' using **plt.title('Min Weight Matching on Complete Graph')** and the plot is displayed using **plt.show()**.

Was this helpful? ✓ Yes ✕ No





To illustrate how this fits in with the original graph, you plot the same min weight pairs (blue lines), but over the trail map (faded) instead of the complete graph. Again, note that the blue lines are the bushwhacking route (as the crow flies edges, not actual trails). You still have a little bit of work to do to find the edges that comprise the shortest route between each pair in Step **3.**

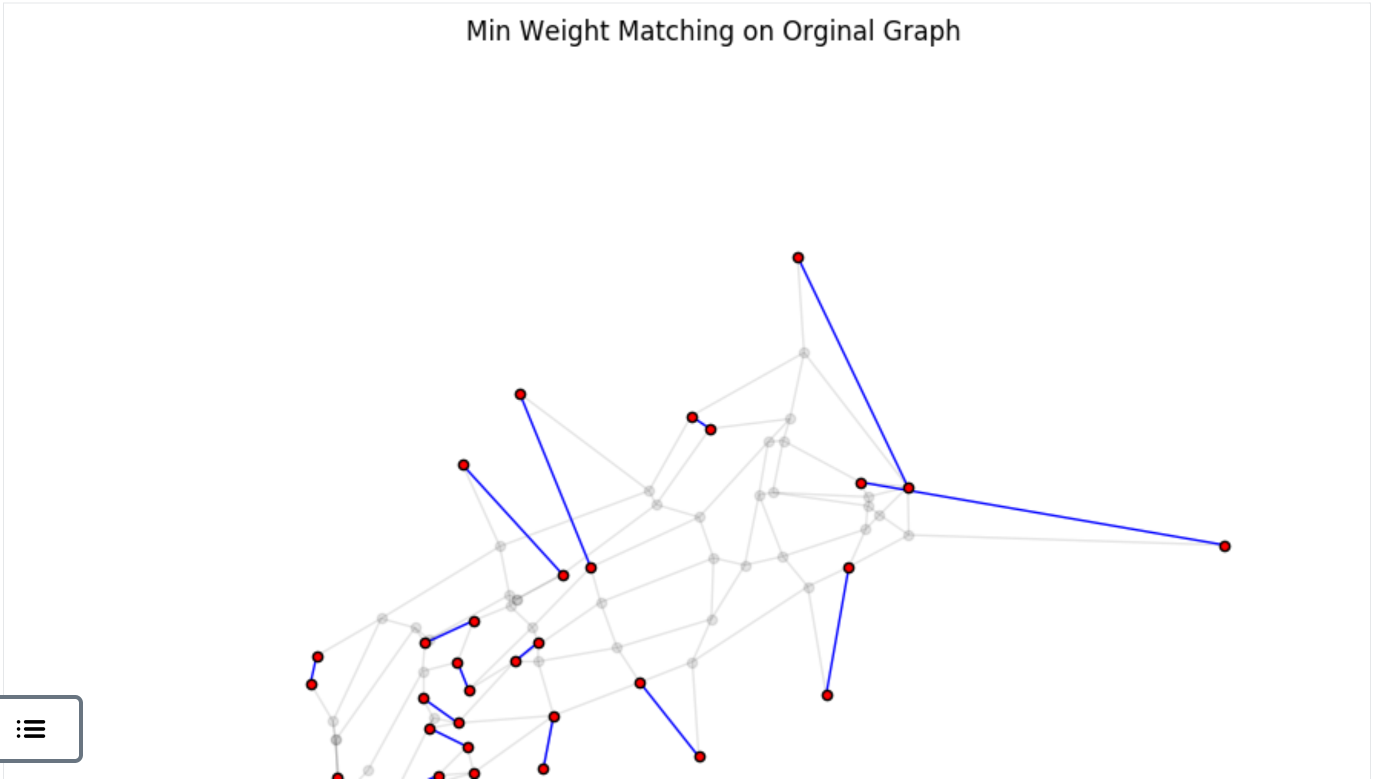
```
plt.figure(figsize=(8, 6))

# Plot the original trail map graph
nx.draw(g, pos=node_positions, node_size=20, alpha=0.1, node_color='black')

# Plot graph to overlay with just the edges from the min weight matching
nx.draw(g_odd_complete_min_edges, pos=node_positions, node_size=20, alpha=1, noc

plt.title('Min Weight Matching on Orginal Graph')
plt.show()
```

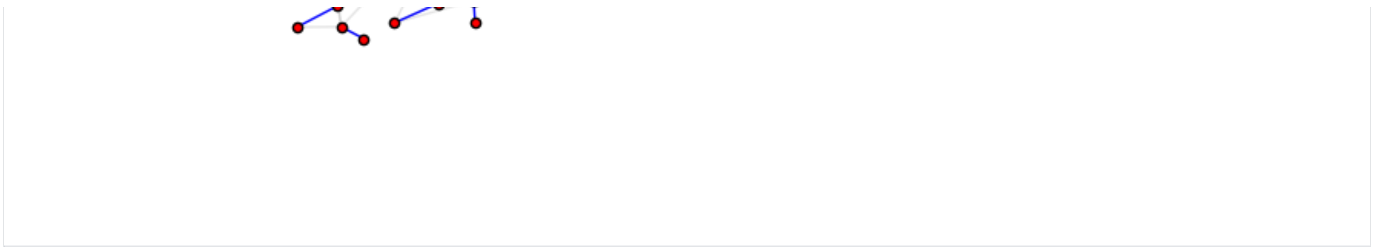
Hide code explanation

OpenAI

• This code creates a plot of a graph with two different sets of edges.
• The first set of edges is the original trail map graph, which is drawn using the **nx.draw()** function with the parameters **g**, **node_positions**, **node_size**, **alpha**, and **node_color**.
• The second set of edges is the minimum weight matching of the odd-degree nodes in the original graph, which is drawn using the **nx.draw()** function with the parameters **g_odd_complete_min_edges**, **node_positions**, **node_size**, **alpha**, **node_color**, and **edge_color**.
• The **plt.figure()** function sets the size of the plot to 8 inches by 6 inches, and the **plt.title()** function adds a title to the plot.
• Finally, the **plt.show()** function displays the plot.

Was this helpful? ✓ Yes ✕ No





Step 2.5: Augment the Original Graph

Now you augment the original graph with the edges from the matching calculated in **2.4**. A simple function to do this is defined below which also notes that these new edges came from the augmented graph. You'll need to know this in **3.** when you actually create the Eulerian circuit through the graph.

```python
def add_augmenting_path_to_graph(graph, min_weight_pairs):
    """
    Add the min weight matching edges to the original graph
    Parameters:
        graph: NetworkX graph (original graph from trailmap)
        min_weight_pairs: list[tuples] of node pairs from min weight matching
    Returns:
        augmented NetworkX graph
    """

    # We need to make the augmented graph a MultiGraph so we can add parallel ed
    graph_aug = nx.MultiGraph(graph.copy())
    for pair in min_weight_pairs:
        graph_aug.add_edge(pair[0],
                           pair[1],
                           attr_dict={'distance': nx.dijkstra_path_length(graph,
                                      'trail': 'augmented'}
                          )
    return graph_aug
```

Hide code explanation

OpenAI

• This code defines a function called **add_augmenting_path_to_graph** that takes in two parameters: **graph** and **min_weight_pairs**.
• The **graph** parameter is a NetworkX graph object representing the original graph from a trailmap, while **min_weight_pairs** is a list of tuples representing the node pairs from the minimum weight matching.
• The function first creates a copy of the original graph and converts it into a **MultiGraph** object.
• This is necessary because the augmented graph will have parallel edges, which are not allowed in a regular graph.
• Next, the function iterates through each pair of nodes in **min_weight_pairs** and adds an edge between them to the augmented graph.
• The **add_edge** method is used to add the edge, and an attribute dictionary is passed in as an argument to specify the distance and trail type of the edge.
• The distance is calculated using Dijkstra's algorithm, which finds the shortest path between the two nodes in the original graph.
• Finally, the function returns the augmented graph with the added edges.

Was this helpful? ✓ Yes ✕ No

Let's confirm that your augmented graph adds the expected number (18) of edges:

```
# Create augmented graph: add the min weight matching edges to g
g_aug = add_augmenting_path_to_graph(g, odd_matching)

# Counts
print('Number of edges in original graph: {}'.format(len(g.edges())))
print('Number of edges in augmented graph: {}'.format(len(g_aug.edges())))
```

Hide code explanation

OpenAI

• This code creates an augmented graph by adding the minimum weight matching edges to the original graph **g**.
• The function **add_augmenting_path_to_graph()** takes two arguments: the original graph **g** and the minimum weight matching edges **odd_matching**.
• The resulting augmented graph is stored in the variable **g_aug**.
• The code then prints the number of edges in the original graph and the augmented graph using the **len()** function and string formatting.
• This allows the user to compare the number of edges in the original graph to the number of edges in the augmented graph.

Was this helpful? ✓ Yes ✕ No

```
Number of edges in original graph: 123
Number of edges in augmented graph: 141
```

Hide code explanation

OpenAI

• This code snippet does not have any programming language specified.
• It simply prints out the number of edges in an original graph and the number of edges in an augmented graph.
• The numbers 123 and 141 represent the respective number of edges in each graph.
• It is not clear from this code snippet how the graphs were created or what the purpose of the augmentation was.

Was this helpful? ✓ Yes ✕ No

Let's also confirm that every node now has even degree:

```
pd.value_counts(g_aug.degree())
```

Hide code explanation

OpenAI

• This code uses the pandas library in Python to count the number of occurrences of each unique value in the "degree" column of the "g_aug" dataframe.
• The "pd.value_counts()" function takes the "degree()" method of the "g_aug" dataframe as its argument, which returns a series of the "degree" column.
• The function then counts the number of occurrences of each unique value in the series and returns a new series with the counts as values and the unique values as indices.
• Overall, this code is useful for quickly understanding the distribution of values in a particular column of a dataframe.

Was this helpful? ✓ Yes ✕ No

```
4    54
2    18
6     5
dtype: int64
```

Hide code explanation

OpenAI

• This code snippet appears to be a pandas Series object with three rows of data.
• The first column contains the index values (4, 2, and 6) and the second column contains the corresponding integer values (54, 18, and 5).
• The "dtype: int64" indicates that the data type of the values in the Series is 64-bit integers.
• However, without more context or information about the code that generated this output, it is difficult to provide a more detailed explanation.

Was this helpful? ✓ Yes ✕ No

CPP Step 3: Compute Eulerian Circuit

Now that you have a graph with even degree the hard optimization work is over. As Euler famously postulated in 1736 with the Seven Bridges of Königsberg problem, there exists a path which visits each edge exactly once if all nodes have even degree. Carl Hierholzer fomally proved this result later in the 1870s.

There are many Eulerian circuits with the same distance that can be constructed. You can 0% of the way there with the NetworkX `eulerian_circuit` function. However there are some limitations.

Limitations you will fix:

1. The augmented graph could (and likely will) contain edges that didn't exist on the original graph. To get the circuit (without bushwhacking), you must break down these augmented edges into the shortest path through the edges that actually exist.
2. `eulerian_circuit` only returns the order in which we hit each node. It does not return the attributes of the edges needed to complete the circuit. This is necessary because you need to keep track of which edges have been walked already when multiple edges exist between two nodes.

Limitations you won't fix:

3. To save your legs some work, you could relax the assumption of the Eulerian circuit that one start and finish at the same node. An **Eulerian path** (the general case of the Eulerian circuit), can also be found if there are exactly two nodes of odd degree. This would save you a little bit of double backing...presuming you could get a ride back from the other end of the park. However, at the time of this writing, NetworkX does not provide a Euler Path algorithm. The **eulerian_circuit code** isn't too bad and could be adopted for this case, but you'll keep it simple here.

Naive Circuit

Nonetheless, let's start with the simple yet incomplete solution:

```python
naive_euler_circuit = list(nx.eulerian_circuit(g_aug, source='b_end_east'))
```

Hide code explanation

OpenAI

• This code uses the NetworkX library in Python to find a naive Euler circuit in a graph represented by the variable **g_aug**.
• An Euler circuit is a path that visits every edge in a graph exactly once and ends at the starting vertex.
• A naive Euler circuit is a circuit that starts and ends at the same vertex, but may visit some edges more than once.
• The **nx.eulerian_circuit()** function in NetworkX returns an iterator over the edges in an Euler circuit of the graph.
• The **source** parameter specifies the starting vertex for the circuit.
• The **list()** function is used to convert the iterator into a list of edges, which is then assigned to the variable **naive_euler_circuit**.

Was this helpful? ✓ Yes ✕ No

As expected, the length of the naive Eulerian circuit is equal to the number of the edges in the augmented graph.

```python
print('Length of eulerian circuit: {}'.format(len(naive_euler_circuit)))
```

Hide code explanation

OpenAI

• This code uses the **print()** function to output a string that includes the length of a variable called **naive_euler_circuit**.
• The **format()** method is used to insert the length of **naive_euler_circuit** into the string.
• The output will be a message that displays the length of the eulerian circuit.

Was this helpful? ✓ Yes ✕ No

```
Length of eulerian circuit: 141
```

Hide code explanation

OpenAI

• This code snippet is simply outputting the length of an Eulerian circuit.
• An Eulerian circuit is a path in a graph that visits every edge exactly once and ends at the starting vertex.
• The length of the circuit is the sum of the weights of all the edges in the circuit.
• It is not clear from this code snippet how the Eulerian circuit was generated or what graph it was generated on.

Was this helpful? ✓ Yes ✕ No

The output is just a list of tuples which represent node pairs. Note that the first node of each pair is the same as the second node from the preceding pair.

```
# Preview naive Euler circuit
naive_euler_circuit[0:10]
```

Hide code explanation

OpenAI

• This code is written in Python.
• The code is simply previewing the first 10 elements of a list called "naive_euler_circuit".
• The list likely contains the nodes or vertices of a graph in the order they are visited in a naive implementation of an Euler circuit algorithm.
• The purpose of this code is to allow the user to quickly check the contents of the list and ensure that it matches their expectations.

Was this helpful? ✓ Yes ✕ No

```
[('b_end_east', 'g_gy2'),
 ('g_gy2', 'b_g'),
 ('b_g', 'b_w'),
 ('b_w', 'b_gy2'),
 ('b_gy2', 'w_gy2'),
 ('w_gy2', 'b_w'),
 ('b_w', 'w_rs'),
 ('w_rs', 'g_rs'),
 ('g_rs', 'b_g'),
 ('b_g', 'b_rs')]
```

Hide code explanation

OpenAI

• This code snippet is a list of tuples, where each tuple represents a connection between two nodes in a graph.
• The nodes are represented by strings, and the connections are represented by the tuples.
• For example, the first tuple ('b_end_east', 'g_gy2') represents a connection between the nodes 'b_end_east' and 'g_gy2'.
• This code is commonly used in graph theory and network analysis to represent the edges of a graph.
• The tuples can be used to create a graph object, which can then be analyzed using various algorithms and techniques.

Was this helpful? ✓ Yes ✕ No

Correct Circuit

Now let's define a function that utilizes the original graph to tell you which trails to use to get from node A to node B. Although verbose in code, this logic is actually quite simple. You simply transform the naive circuit which included edges that did not exist in the original graph to a Eulerian circuit using only edges that exist in the original graph.

You loop through each edge in the naive Eulerian circuit (`naive_euler_circuit`). Wherever you encounter an edge that does not exist in the original graph, you replace it with the sequence of edges comprising the shortest path between its nodes using the original graph.

```
def create_eulerian_circuit(graph_augmented, graph_original, starting_node    ne)
    """Create the eulerian path using only edges from the original graph.""
    euler_circuit = []
    naive_circuit = list(nx.eulerian_circuit(graph_augmented, source=starting_nc

    for edge in naive_circuit:
        edge_data = graph_augmented.get_edge_data(edge[0], edge[1])

        if edge_data[0]['trail'] != 'augmented':
            # If `edge` exists in original graph, grab the edge attributes and a
            edge_att = graph_original[edge[0]][edge[1]]
            euler_circuit.append((edge[0], edge[1], edge_att))
        else:
            aug_path = nx.shortest_path(graph_original, edge[0], edge[1], weight
            aug_path_pairs = list(zip(aug_path[:-1], aug_path[1:]))

            print('Filling in edges for augmented edge: {}'.format(edge))
            print('Augmenting path: {}'.format(' => '.join(aug_path)))
            print('Augmenting path pairs: {}\n'.format(aug_path_pairs))

            # If `edge` does not exist in original graph, find the shortest path
            #  add the edge attributes for each link in the shortest path.
            for edge_aug in aug_path_pairs:
                edge_aug_att = graph_original[edge_aug[0]][edge_aug[1]]
                euler_circuit.append((edge_aug[0], edge_aug[1], edge_aug_att))

    return euler_circuit
```

Hide code explanation

OpenAI

• This code defines a function called **create_eulerian_circuit** that takes in two arguments: **graph_augmented** and **graph_original**, which are both graphs.
• The function creates an Eulerian path using only edges from the original graph.
• First, the function creates an empty list called **euler_circuit** to store the edges of the Eulerian path.
• Then, it calls the **eulerian_circuit** function from the **networkx** library to create a naive circuit using the augmented graph and a starting node (if specified).
• The **naive_circuit** variable stores the edges of this circuit.
• Next, the function loops through each edge in the **naive_circuit** list.
• For each edge, it checks if the edge exists in the original graph by using the **get_edge_data** method of the **graph_augmented** object.
• If the edge exists in the original graph, the function retrieves the edge attributes from the **ph_original** object and adds them to the **euler_circuit** list.
the edge does not exist in the original graph, the function finds the shortest path between the

nodes of the edge in the original graph using the **shortest_path** function from **networkx**.
• It then creates a list of pairs of nodes that represent the edges in the shortest path.
• For each edge in this list, the function retrieves the edge attributes from the **graph_original** object and adds them to the **euler_circuit** list.
• Finally, the function returns the **euler_circuit** list, which contains the edges of the Eulerian path using only edges from the original graph.
• The function also includes some print statements for debugging purposes, which print out information about the augmented edges and the shortest paths between their nodes.

Was this helpful? Yes No

You hack **limitation 3** a bit by starting the Eulerian circuit at the far east end of the park on the Blue trail (node "b_end_east"). When actually running this thing, you could simply skip the last direction which doubles back on it.

Verbose print statements are added to convey what happens when you replace nonexistent edges from the augmented graph with the shortest path using edges that actually exist.

```
# Create the Eulerian circuit
euler_circuit = create_eulerian_circuit(g_aug, g, 'b_end_east')
```

Hide code explanation

OpenAI

• This code creates an Eulerian circuit using the function **create_eulerian_circuit()**.
• The circuit is created based on the input graph **g_aug** and the original graph **g**.
• The starting point for the circuit is specified as the node **'b_end_east'**.
• An Eulerian circuit is a path in a graph that visits every edge exactly once and ends at the starting node.
• The **create_eulerian_circuit()** function uses the Hierholzer's algorithm to find such a circuit in the input graph.
• Without more context, it is difficult to determine the specific implementation of **create_eulerian_circuit()** and the structure of the input graph.

Was this helpful? ✓ Yes ✕ No

```
Filling in edges for augmented edge: ('b_end_east', 'g_gy2')
Augmenting path: b_end_east => b_y => b_o => b_gy2 => w_gy2 => g_gy2
Augmenting path pairs: [('b_end_east', 'b_y'), ('b_y', 'b_o'), ('b_o', 'b_gy2'),

Filling in edges for augmented edge: ('b_bw', 'rh_end_tt_1')
Augmenting path: b_bw => b_tt_1 => rh_end_tt_1
Augmenting path pairs: [('b_bw', 'b_tt_1'), ('b_tt_1', 'rh_end_tt_1')]

Filling in edges for augmented edge: ('b_tt_3', 'rt_end_north')
Augmenting path: b_tt_3 => b_tt_2 => tt_rt => v_rt => rt_end_north
Augmenting path pairs: [('b_tt_3', 'b_tt_2'), ('b_tt_2', 'tt_rt'), ('tt_rt', 'v_

Filling in edges for augmented edge: ('rc_end_north', 'g_gy1')
Augmenting path: rc_end_north => v_rc => b_rc => g_rc => g_gy1
Augmenting path pairs: [('rc_end_north', 'v_rc'), ('v_rc', 'b_rc'), ('b_rc', 'g_

Filling in edges for augmented edge: ('y_gy1', 'rc_end_south')
Augmenting path: y_gy1 => y_rc => rc_end_south
Augmenting path pairs: [('y_gy1', 'y_rc'), ('y_rc', 'rc_end_south')]

Filling in edges for augmented edge: ('b_end_west', 'rd_end_south')
Augmenting path: b_end_west => b_v => rd_end_south
Augmenting path pairs: [('b_end_west', 'b_v'), ('b_v', 'rd_end_south')]

Filling in edges for augmented edge: ('rh_end_north', 'rd_end_north')
Augmenting path: rh_end_north => v_rh => v_rd => rd_end_north
Augmenting path pairs: [('rh_end_north', 'v_rh'), ('v_rh', 'v_rd'), ('v_rd', 'rc

Filling in edges for augmented edge: ('v_end_east', 'rs_end_north')
Augmenting path: v_end_east => v_rs => rs_end_north
Augmenting path pairs: [('v_end_east', 'v_rs'), ('v_rs', 'rs_end_north')]

Filling in edges for augmented edge: ('y_gy2', 'rs_end_south')
Augmenting path: y_gy2 => y_rs => rs_end_south
Augmenting path pairs: [('y_gy2', 'y_rs'), ('y_rs', 'rs_end_south')]
```

Hide code explanation

OpenAI

• This code snippet is not actually code, but rather a series of print statements that show the results of some algorithm that is augmenting paths in a graph.
• The **Augmenting path** lines show the path that was found, and the **Augmenting path pairs** lines show the pairs of nodes that make up that path.
he **Filling in edges for augmented edge** lines indicate which edge was added to the graph to ment the path.

• Without more context, it is difficult to say exactly what algorithm is being used or what the graph looks like.

Was this helpful? Yes No

You see that the length of the Eulerian circuit is longer than the naive circuit, which makes sense.

```
print('Length of Eulerian circuit: {}'.format(len(euler_circuit)))
```

Hide code explanation

OpenAI

• This code uses the **print()** function to output a message to the console.
• The message includes a string that says "Length of Eulerian circuit: " and the length of the **euler_circuit** variable.
• The **len()** function is used to determine the length of the **euler_circuit** variable, which is then inserted into the string using the **.format()** method.
• The output will display the length of the **euler_circuit** variable.

Was this helpful? ✓ Yes ✗ No

```
Length of Eulerian circuit: 158
```

Hide code explanation

OpenAI

• This code snippet is simply displaying the length of an Eulerian circuit.
• An Eulerian circuit is a path in a graph that visits every edge exactly once and ends at the starting vertex.
• The length of the circuit is the sum of the weights of all the edges in the circuit.
• Without additional context or code, it is difficult to provide more information about how this specific Eulerian circuit was calculated or what graph it was calculated on.

Was this helpful? ✓ Yes ✗ No

Compute CPP Solution

Text

Here's a printout of the solution in text:

```python
# Preview first 20 directions of CPP solution
for i, edge in enumerate(euler_circuit[0:20]):
    print(i, edge)
```

Hide code explanation

OpenAI

• This code is written in Python.
• The code is previewing the first 20 directions of a CPP (Chinese Postman Problem) solution.
• The **for** loop iterates over the first 20 edges in the **euler_circuit** list and prints out the index **i** and the edge itself **edge**.
• The **enumerate()** function is used to get both the index and the value of each edge in the list.
• The output will be a list of the first 20 edges in the **euler_circuit** list, with each edge's index printed alongside it.

Was this helpful? ✓ Yes ✕ No

```
0 ('b_end_east', 'b_y', {'color': 'blue', 'estimate': 0, 'trail': 'b', 'di nce
1 ('b_y', 'b_o', {'color': 'blue', 'estimate': 0, 'trail': 'b', 'distance': 0.08
2 ('b_o', 'b_gy2', {'color': 'blue', 'estimate': 1, 'trail': 'b', 'distance': 0.
3 ('b_gy2', 'w_gy2', {'color': 'yellowgreen', 'estimate': 1, 'trail': 'gy2', 'di
4 ('w_gy2', 'g_gy2', {'color': 'yellowgreen', 'estimate': 0, 'trail': 'gy2', 'di
5 ('g_gy2', 'b_g', {'color': 'green', 'estimate': 0, 'trail': 'g', 'distance':
6 ('b_g', 'b_w', {'color': 'blue', 'estimate': 0, 'trail': 'b', 'distance': 0.16
7 ('b_w', 'b_gy2', {'color': 'blue', 'estimate': 0, 'trail': 'b', 'distance': 0.
8 ('b_gy2', 'w_gy2', {'color': 'yellowgreen', 'estimate': 1, 'trail': 'gy2', 'di
9 ('w_gy2', 'b_w', {'color': 'gray', 'estimate': 0, 'trail': 'w', 'distance': 0.
10 ('b_w', 'w_rs', {'color': 'gray', 'estimate': 1, 'trail': 'w', 'distance': 0.
11 ('w_rs', 'g_rs', {'color': 'red', 'estimate': 0, 'trail': 'rs', 'distance':
12 ('g_rs', 'b_g', {'color': 'green', 'estimate': 1, 'trail': 'g', 'distance':
13 ('b_g', 'b_rs', {'color': 'blue', 'estimate': 0, 'trail': 'b', 'distance': 0.
14 ('b_rs', 'g_rs', {'color': 'red', 'estimate': 0, 'trail': 'rs', 'distance':
15 ('g_rs', 'g_rc', {'color': 'green', 'estimate': 0, 'trail': 'g', 'distance':
16 ('g_rc', 'g_gy1', {'color': 'green', 'estimate': 0, 'trail': 'g', 'distance':
17 ('g_gy1', 'g_rt', {'color': 'green', 'estimate': 0, 'trail': 'g', 'distance':
18 ('g_rt', 'g_w', {'color': 'green', 'estimate': 0, 'trail': 'g', 'distance':
19 ('g_w', 'o_w_1', {'color': 'gray', 'estimate': 0, 'trail': 'w', 'distance':
```

Hide code explanation

OpenAI

• This code snippet appears to be a list of tuples, where each tuple represents a connection between two points on a map.
• The first element of each tuple is the starting point, the second element is the ending point, and the third element is a dictionary containing information about the connection, such as the color of the trail, the estimated time to travel the distance, and the distance between the two points.
• Without more context or information about the tutorial, it is difficult to provide a more detailed explanation of how this code works or what it is used for.

Was this helpful? ✓ Yes ✕ No

You can tell pretty quickly that the algorithm is not very loyal to any particular trail, jumping from one to the next pretty quickly. An extension of this approach could get fancy and build in some notion of trail loyalty into the objective function to make actually running this route more manageable.

Stats

Let's peak into your solution to see how reasonable it looks.
important to dwell on this verbose code, just the printed output)

```
# Computing some stats
total_mileage_of_circuit = sum([edge[2]['distance'] for edge in euler_circuit])
total_mileage_on_orig_trail_map = sum(nx.get_edge_attributes(g, 'distance').valu
_vcn = pd.value_counts(pd.value_counts([(e[0]) for e in euler_circuit]), sort=Fa
node_visits = pd.DataFrame({'n_visits': _vcn.index, 'n_nodes': _vcn.values})
_vce = pd.value_counts(pd.value_counts([sorted(e)[0] + sorted(e)[1] for e in nx.
edge_visits = pd.DataFrame({'n_visits': _vce.index, 'n_edges': _vce.values})

# Printing stats
print('Mileage of circuit: {0:.2f}'.format(total_mileage_of_circuit))
print('Mileage on original trail map: {0:.2f}'.format(total_mileage_on_orig_trai
print('Mileage retracing edges: {0:.2f}'.format(total_mileage_of_circuit-total_m
print('Percent of mileage retraced: {0:.2f}%\n'.format((1-total_mileage_of_circu

print('Number of edges in circuit: {}'.format(len(euler_circuit)))
print('Number of edges in original graph: {}'.format(len(g.edges())))
print('Number of nodes in original graph: {}\n'.format(len(g.nodes())))

print('Number of edges traversed more than once: {}\n'.format(len(euler_circuit)

print('Number of times visiting each node:')
print(node_visits.to_string(index=False))

print('\nNumber of times visiting each edge:')
print(edge_visits.to_string(index=False))
```

Hide code explanation OpenAI

• This code computes and prints various statistics related to a graph and its Euler circuit.
• First, it calculates the total mileage of the circuit by summing the distances of each edge in the Euler circuit.
• It also calculates the total mileage on the original trail map by summing the distances of all edges in the graph.
• Next, it uses pandas to count the number of times each node and edge is visited in the Euler circuit.
• This is done by first creating a list of the starting nodes of each edge in the Euler circuit, then counting the number of times each node appears in that list.
• Similarly, a list of the sorted pairs of nodes in each edge is created, and the number of times each pair appears is counted.
• These counts are then stored in dataframes for easier printing.
• Finally, the code prints out all the computed statistics, including the mileage of the circuit, the number of edges and nodes in the graph and circuit, the number of edges traversed more than [illegible]e, and the number of times each node and edge is visited.
[illegible]e code uses various Python libraries such as networkx and pandas to perform these

calculations and print the results.

Was this helpful? Yes No

```
Mileage of circuit: 33.59
Mileage on original trail map: 25.76
Mileage retracing edges: 7.83
Percent of mileage retraced: 30.40%

Number of edges in circuit: 158
Number of edges in original graph: 123
Number of nodes in original graph: 77

Number of edges traversed more than once: 35

Number of times visiting each node:
n_nodes  n_visits
     18         1
     38         2
     20         3
      1         4

Number of times visiting each edge:
n_edges  n_visits
     88         1
     35         2
```

Hide code explanation

OpenAI

• This code snippet does not contain any programming language.
• It is simply a set of output values that were likely generated by a program or script.
• The values represent various statistics related to a circuit or trail map, including the mileage of the circuit, the number of edges and nodes in the original graph, and the number of times each node and edge was visited.
• The purpose of this output is likely to provide insights into the efficiency or effectiveness of a particular route or path.

Was this helpful? ✓ Yes ✕ No

Visualize CPP Solution

While NetworkX also provides functionality to visualize graphs, they are **notably humble** in epartment:

NetworkX provides basic functionality for visualizing graphs, but its main goal is to enable graph analysis rather than perform graph visualization. In the future, graph visualization functionality may be removed from NetworkX or only available as an add-on package.

Proper graph visualization is hard, and we highly recommend that people visualize their graphs with tools dedicated to that task. Notable examples of dedicated and fully-featured graph visualization tools are Cytoscape, Gephi, Graphviz and, for LaTeX typesetting, PGF/TikZ.

That said, the built-in NetworkX drawing functionality with matplotlib is powerful enough for eyeballing and visually exploring basic graphs, so you stick with NetworkX `draw` for this tutorial.

I used graphviz and the dot graph description language to visualize the solution in my Python package postman_problems. Although it took some legwork to convert the NetworkX graph structure to a dot graph, it does unlock enhanced quality and control over visualizations.

Create CPP Graph

Your first step is to convert the list of edges to walk in the Euler circuit into an edge list with plot-friendly attributes.

`create_cpp_edgelist` Creates an edge list with some additional attributes that you'll use for plotting:

- **sequence:** records a sequence of when we walk each edge.
- **visits:** is simply the number of times we walk a particular edge.

TOPICS

Python Data Science Data Visualization

Learn more about Python

Introduction to Network Analysis in Python

• Beginner 4 hr 68.3K

This course will equip you with the skills to analyze, visualize, and make sense of networks using the NetworkX library.

See Details → Start Course

```
        if edge in cpp_edgelist:
            cpp_edgelist[edge][2]['sequence'] += ', ' + str(i)
            cpp_edgelist[edge][2]['visits'] += 1

        else:
            cpp_edgelist[edge] = e
            cpp_edgelist[edge][2]['sequence'] = str(i)
            cpp_edgelist[edge][2]['visits'] = 1

    list(cpp_edgelist.values())
```

See More →

Hide code explanation

OpenAI

efines a function called **create_cpp_edgelist** that takes in a list of tuples called **euler_circuit**.

• The function creates an edgelist without parallel edges for visualization purposes.

n initializes an empty dictionary called **cpp_edgelist**.

s through each edge in the **euler_circuit** list and creates a frozenset of the two make up the edge.

• If the edge already exists in the **cpp_edgelist** dictionary, the function updates the edge's sequence and visit count.

• Otherwise, it adds the edge to the **cpp_edgelist** dictionary and initializes its sequence and visit count.

See More →

• Finally, the function returns a list of the values in the **cpp_edgelist** dictionary.

• Overall, this function is used to create a simplified edgelist for visualization purposes by combining duplicate edges and keeping track of their sequence and number of visits.

Was this helpful? ✓ Yes ✗ No

Grow your data skills with DataCamp for Mobile

create the CPP edge list:

ke progress on the go with our mobile courses and daily 5-minute coding challenges.

```
cpp_edgelist = create_cpp_edgelist(euler_circuit)
```

Hide code explanation

OpenAI

- This code calls the function **create_cpp_edgelist** and passes in the variable **euler_circuit** as an argument.
- The function likely takes the input **euler_circuit**, which is likely a list of edges in a graph, and converts it into a format that can be used in C++ code.
- The resulting **cpp_edgelist** variable is likely a C++-compatible representation of the graph edges.

Was this helpful? ✓ Yes ✕ No

As expected, your edge list has the same number of edges as the original graph.

```
print('Number of edges in CPP edge list: {}'.format(len(cpp_edgelist)))
```

Hide code explanation

OpenAI

- This code uses the **print()** function to output a message to the console.
- The message includes a string that says "Number of edges in CPP edge list: " and the length of the **cpp_edgelist** variable, which is obtained using the **len()** function.
- The **format()** method is used to insert the length of **cpp_edgelist** into the string.
- This code is likely used to display the number of edges in a graph represented by the **cpp_edgelist** variable.

Was this helpful? ✓ Yes ✕ No

```
Number of edges in CPP edge list: 123
```

Hide code explanation

OpenAI

- This code snippet is simply printing out the number of edges in a CPP (C++) edge list, which is 123.
- There is no actual code being executed here, just a print statement displaying the value of the variable holding the number of edges.

Was this helpful? ✓ Yes ✕ No

Learn Python
Learn R
Learn AI
Learn SQL
Learn Power BI
Learn Tableau
Assessments
Career Tracks
Skill Tracks
Courses
Data Science Roadmap
Upcoming Courses
Python Courses
R Courses
Courses

The CPP edge list looks similar to `euler_circuit`, just with a few additional attributes.

```
# Preview CPP plot-friendly edge list
cpp_edgelist[0:3]
```

Hide code explanation

OpenAI

• This code is written in Python.
• The code is simply previewing the first three rows of the "cpp_edgelist" data frame, which is assumed to be in a format that is friendly for plotting.
• The output will show the first three rows of the data frame.

Was this helpful? ✓ Yes ✕ No

```
[('rh_end_tt_4',
  'nature_end_west',
  {'color': 'black',
   'distance': 0.2,
   'estimate': 0,
   'sequence': '73',
   'trail': 'tt',
   'visits': 1}),
 ('rd_end_south',
  'b_rd',
  {'color': 'red',
   'distance': 0.13,
   'estimate': 0,
   'sequence': '95',
   'trail': 'rd',
   'visits': 1}),
 ('w_gy1',
  'w_rc',
  {'color': 'gray',
   'distance': 0.33,
   'estimate': 0,
   'sequence': '151',
   'trail': 'w',
   'visits': 1})]
```

Hide code explanation OpenAI

- This code snippet is a list of tuples, where each tuple contains three elements.
- The first element is a string representing a location or endpoint on a trail, the second element is a string representing the name of the trail, and the third element is a dictionary containing information about the trail and location.
- The dictionary contains the following key-value pairs: 'color': a string representing the color of the trail 'distance': a float representing the distance of the location from the start of the trail 'estimate': an integer representing an estimate of the time it takes to travel from the start of the trail to the location 'sequence': a string representing the sequence number of the location on the trail 'trail': a string representing the name of the trail 'visits': an integer representing the number of times the location has been visited.
- Overall, this code is a data structure that contains information about various locations and trails, which could be used for mapping or navigation purposes.

Was this helpful? ✓ Yes ✕ No

Blog
Tutorials
Open Source
RDocumentation
Course Editor
Book a Demo with DataCamp for Business
Data Portfolio
Portfolio Leaderboard
Pricing
For Business
For Universities
Discounts, Promos & Sales
DataCamp Donates
Help Center
Become an Instructor
Become an Affiliate
About Us
er Stories

```python
# Create CPP solution graph
g_cpp = nx.Graph(cpp_edgelist)
```

Hide code explanation

OpenAI

- This code creates a graph object using the NetworkX library in Python.
- The graph is created from an edgelist called **cpp_edgelist**.
- The **nx.Graph()** function takes the edgelist as an argument and returns a graph object.
- The resulting graph object is assigned to the variable **g_cpp**.
- This code is used to create a graph representation of a solution in C++ programming language.

Was this helpful? ✓ Yes ✗ No

Visualization 1: Retracing Steps

Here you illustrate which edges are walked once (gray) 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.

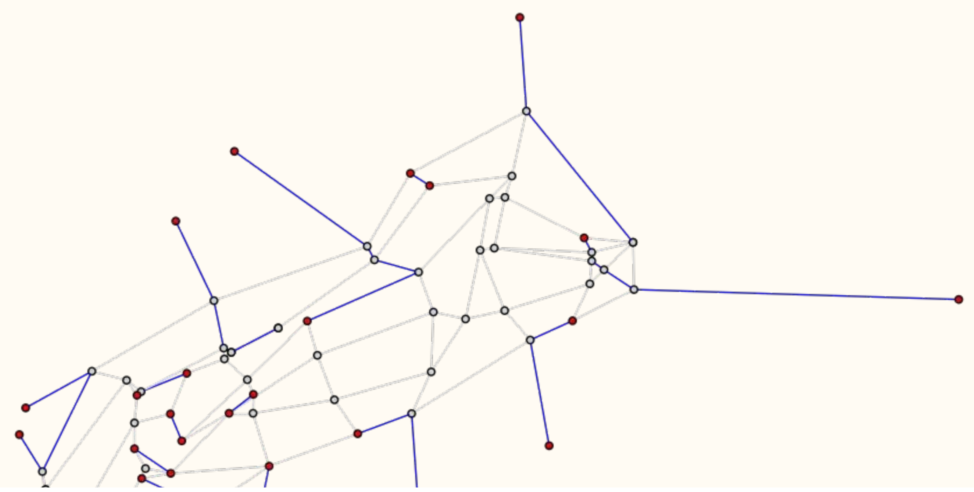
```
plt.figure(figsize=(14, 10))

visit_colors = {1:'lightgray', 2:'blue'}
edge_colors = [visit_colors[e[2]['visits']] for e in g_cpp.edges(data=True)]
node_colors = ['red'  if node in nodes_odd_degree else 'lightgray' for node in g

nx.draw_networkx(g_cpp, pos=node_positions, node_size=20, node_color=node_colors
plt.axis('off')
plt.show()
```

Hide code explanation

OpenAI

- This code uses the NetworkX library to draw a network graph.
- First, a figure is created with a size of 14×10 using **plt.figure(figsize=(14, 10))**.
- Then, two dictionaries are created: **visit_colors** maps the number of visits to a color, and **edge_colors** uses this mapping to assign a color to each edge in the graph based on the number of visits.
- Next, a list comprehension is used to create **node_colors**, which assigns the color red to nodes with an odd degree and light gray to all other nodes.
- Finally, **nx.draw_networkx()** is used to draw the graph with the specified node positions, node size, node color, and edge color.
- **with_labels=False** is used to hide the node labels.
- **plt.axis('off')** is used to remove the axis labels and ticks, and **plt.show()** is used to display the graph.

Was this helpful? ✓ Yes ✕ No



Visualization 2: CPP Solution Sequence

Here you 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).

```
plt.figure(figsize=(14, 10))

edge_colors = [e[2]['color'] for e in g_cpp.edges(data=True)]
nx.draw_networkx(g_cpp, pos=node_positions, node_size=10, node_color='black', ec

bbox = {'ec':[1,1,1,0], 'fc':[1,1,1,0]}  # hack to label edges over line (rather
edge_labels = nx.get_edge_attributes(g_cpp, 'sequence')
nx.draw_networkx_edge_labels(g_cpp, pos=node_positions, edge_labels=edge_labels,

plt.axis('off')
plt.show()
```

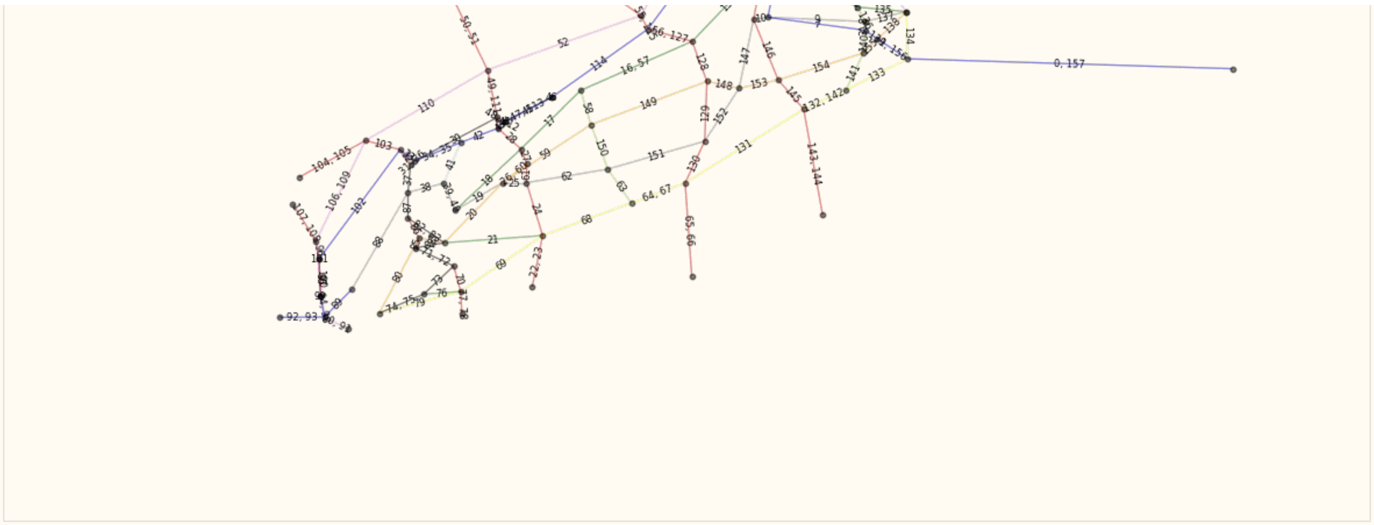
Hide code explanation

OpenAI

• This code creates a visualization of a graph using the NetworkX library in Python.
• First, a figure is created with a size of 14 by 10 using **plt.figure(figsize=(14, 10))**.
• Next, the colors of the edges are extracted from the graph **g_cpp** using a list comprehension and stored in **edge_colors**.
• The graph is then drawn using **nx.draw_networkx()**, with the node positions specified by **node_positions**, a node size of 10, black node color, and edge colors specified by **edge_colors**.
• The labels for the nodes are not shown (**with_labels=False**) and the transparency of the graph is set to 0.5 (**alpha=0.5**).
• A dictionary **bbox** is created to specify the edge label box style, with the edge color set to transparent (**'ec':[1,1,1,0]**) and the fill color also set to transparent (**'fc':[1,1,1,0]**).
• The edge labels are then extracted from the graph using **nx.get_edge_attributes(g_cpp, 'sequence')** and drawn on the graph using **nx.draw_networkx_edge_labels()**, with the node positions specified by **node_positions**, the edge labels specified by **edge_labels**, and the edge label box style specified by **bbox**.
• The font size of the edge labels is set to 6.
• Finally, the axis is turned off using **plt.axis('off')** and the graph is displayed using **plt.show()**.

Was this helpful? ✓ Yes ✕ No

120, 121
117
122
118, 119
53, 54
123
116
126
5
11
6

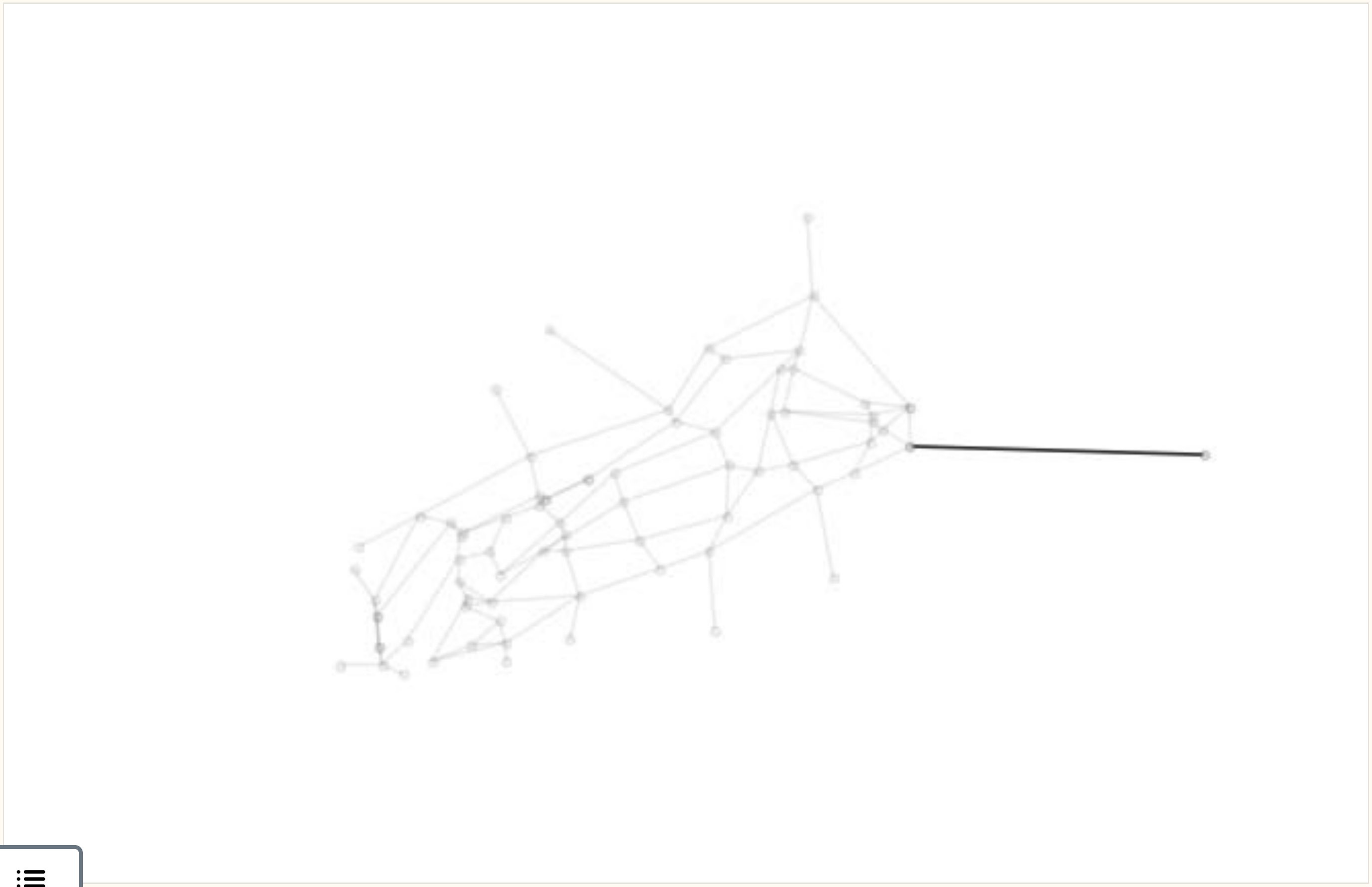


Visualization 3: Movie

The movie below that traces the Euler circuit from beginning to end is embedded below. Edges are colored black the first time they are walked and red the second time.

Note that this gif doesn't do give full visual justice to edges which overlap another or are too small to visualize properly. A more robust visualization library such as graphviz could address this by plotting splines instead of straight lines between nodes.

The code that creates it is presented below as a reference.



First a PNG image is produced for each direction (edge walked) from the CPP solution.

```
visit_colors = {1:'black', 2:'red'}
edge_cnter = {}
g_i_edge_colors = []
for i, e in enumerate(euler_circuit, start=1):

    edge = frozenset([e[0], e[1]])
    if edge in edge_cnter:
        edge_cnter[edge] += 1
    else:
        edge_cnter[edge] = 1

    # Full graph (faded in background)
    nx.draw_networkx(g_cpp, pos=node_positions, node_size=6, node_color='gray',

    # Edges walked as of iteration i
    euler_circuit_i = copy.deepcopy(euler_circuit[0:i])
    for i in range(len(euler_circuit_i)):
        edge_i = frozenset([euler_circuit_i[i][0], euler_circuit_i[i][1]])
        euler_circuit_i[i][2]['visits_i'] = edge_cnter[edge_i]
    g_i = nx.Graph(euler_circuit_i)
    g_i_edge_colors = [visit_colors[e[2]['visits_i']] for e in g_i.edges(data=Tr

    nx.draw_networkx_nodes(g_i, pos=node_positions, node_size=6, alpha=0.6, node
    nx.draw_networkx_edges(g_i, pos=node_positions, edge_color=g_i_edge_colors,

    plt.axis('off')
    plt.savefig('fig/png/img{}.png'.format(i), dpi=120, bbox_inches='tight')
    plt.close()
```

Hide code explanation

OpenAI

• This code is a part of a program that generates a sequence of images to visualize the process of traversing an Eulerian circuit in a graph.
• The code first initializes a dictionary **visit_colors** that maps the number of times an edge is visited to a color.
• It also initializes an empty dictionary **edge_cnter** and an empty list **g_i_edge_colors**.
• The program then enters a loop that iterates over each edge in the Eulerian circuit.
• For each edge, it creates a **fro**

Was this helpful? ✓ Yes ✗ No

Then the the PNG images are stitched together to make the nice little gif above.

First the PNGs are sorted in the order from 0 to 157. Then they are stitched together using `imageio` at 3 frames per second to create the gif.

```
import glob
import numpy as np
import imageio
import os

def make_circuit_video(image_path, movie_filename, fps=5):
    # sorting filenames in order
    filenames = glob.glob(image_path + 'img*.png')
    filenames_sort_indices = np.argsort([int(os.path.basename(filename).split('.
    filenames = [filenames[i] for i in filenames_sort_indices]

    # make movie
    with imageio.get_writer(movie_filename, mode='I', fps=fps) as writer:
        for filename in filenames:
            image = imageio.imread(filename)
            writer.append_data(image)

make_circuit_video('fig/png/', 'fig/gif/cpp_route_animation.gif', fps=3)
```

Hide code explanation

OpenAI

- This code defines a function called **make_circuit_video** that takes in three arguments: **image_path**, **movie_filename**, and **fps**.
- The function first imports the necessary modules: **glob**, **numpy**, **imageio**, and **os**.
- It then uses **glob** to find all files in the **image_path** directory that match the pattern **'img*.png'**.

Was this helpful? ✓ Yes ✕ No

Next Steps

Congrats, you have finished this tutorial solving the Chinese Postman Problem in Python. You have covered a lot of ground in this tutorial (33.6 miles of trails to be exact). For a deeper dive into network fundamentals, you might be interested in Datacamp's Network Analysis in Python course which provides a more thorough treatment of the core concepts.

Don't hesitate to check out the NetworkX documentation for more on how to create, ...ipulate and traverse these complex networks. The docs are comprehensive with a ... number of examples and a series of tutorials.

If you're interested in solving the CPP on your own graph, I've packaged the functionality within this tutorial into the postman_problems Python package on Github. You can also piece together the code blocks from this tutorial with a different edge and node list, but the postman_problems package will probably get you there more quickly and cleanly.

One day I plan to implement the extensions of the CPP (Rural and Windy Postman Problem) here as well. I also have grand ambitions of writing about these extensions and experiences testing the routes out on the trails on my blog here. Another application I plan to explore and write about is incorporating lat/long coordinates to develop (or use) a mechanism to send turn-by-turn directions to my Garmin watch.

And of course one last next step: getting outside and trail running the route!

If you would like to learn more about Networks in Python, check out these DataCamp's courses:

Introduction to Network Analysis in Python

Intermediate Network Analysis in Python

References

1: Edmonds, Jack (1965). "Paths, trees, and flowers". Canad. J. Math. 17: 449–467.
2: Galil, Z. (1986). "Efficient algorithms for finding maximum matching in graphs". ACM Computing Surveys. Vol. 18, No. 1: 23-38.