University of Science and Technology at Zewail City

Project Report

Chinese Postman Problem

Discrete Mathematics (MATH-308)
Spring 2023

Hazem Mohamed 202200777
Ahmed Amgad 202200393
Thomas Emad 202100184
Mohamed Ezz 202100789

Supervised by
Prof. Abdullah Abutahoon

Contents

1 Introduction

The Chinese Postman Problem, a classic conundrum in graph theory, revolves around the quest for an optimal route for a weary postman navigating through a neighborhood to deliver mail efficiently. In a weighted graph, where each edge carries a numerical weight, the problem entails finding the minimum-weight tour that covers every street exactly once before returning to the starting point. This definition aligns the Chinese Postman Problem with the broader concept of seeking optimal tours in weighted connected graphs with non-negative weights.

In the context of an Eulerian graph, where every vertex has an even degree, any Euler tour of the graph automatically qualifies as an optimal tour. An Euler tour traverses each edge exactly once, making it an ideal solution to the postman's dilemma. Fortunately, determining an Euler tour in an Eulerian graph is a tractable task, thanks to the algorithm devised by Fleury in 1921. This algorithm, elegantly outlined by Lucas, constructs an Euler tour methodically by tracing out a trail while ensuring that cut edges of the untraced subgraph are chosen judiciously, with preference given to alternatives whenever available.

The algorithm's efficiency in finding Euler tours simplifies the resolution of the Chinese Postman Problem in Eulerian scenarios, providing a clear path for the postman to follow. However, the problem becomes more intricate when the graph is not Eulerian. In such cases, the search for an optimal tour demands additional strategies to navigate through the graph efficiently while accounting for odd-degree vertices and the necessity of traversing certain edges multiple times.

2 Assumptions made about graphs used in this report

1. We ignore optional trails in this tutorial and focus on required trails only.
2. The CPP assumes that the cost of walking a trail is equivalent to its distance, regardless of which direction it is walked.
3. We assumed no multiple edges connecting the same nodes

3 Algorithm for CPP

3.1 General steps for the algorithm used

1. Find the vertices (nodes) of the odd degree
2. Find the minimum distance pair
3. Compute Eulerian circuit

CPP Step 1: Find Nodes of Odd Degree

This is a pretty straightforward counting computation.

CPP Step 2: Find Min Distance Pairs

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

Step 2.1: Compute Node Pairs

using the `itertools combination` function to compute all possible pairs of the odd degree nodes. our graph is undirected so we don't care about order: For example (ab) == (ba).

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

However if we had 3600 odd node pairs instead you'd have ∼6.5 million pairs to optimize. That's a ∼10000x increase in output given a 100x increase in input size.

$$\#\text{ of pairs} = \binom{n}{r} = \frac{n!}{r!(n-r)!} = \frac{36!}{2!(36-2)!} = 630$$

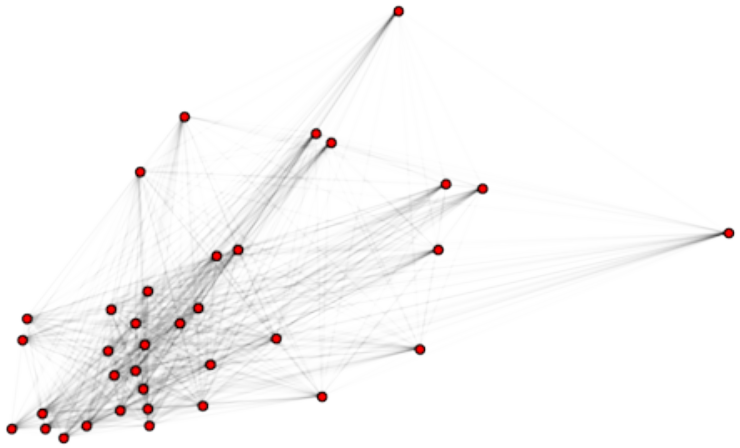
Step 2.2: Compute Shortest Paths between Node Pairs

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.weapply this function to every pair (all 630) calculated above in `odd_node_pairs`.

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 of a 7 node complete graph with 21 (7 choose 2) edges:



(a) *Complete Graph of Odd-degree Nodes*

The graph we create above has 36 nodes and 630 edges with their corresponding edge weight (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 counterintuitive but is necessary for Step 2.4 where we calculate the minimum weight matching on the complete graph.

Ideally we'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.

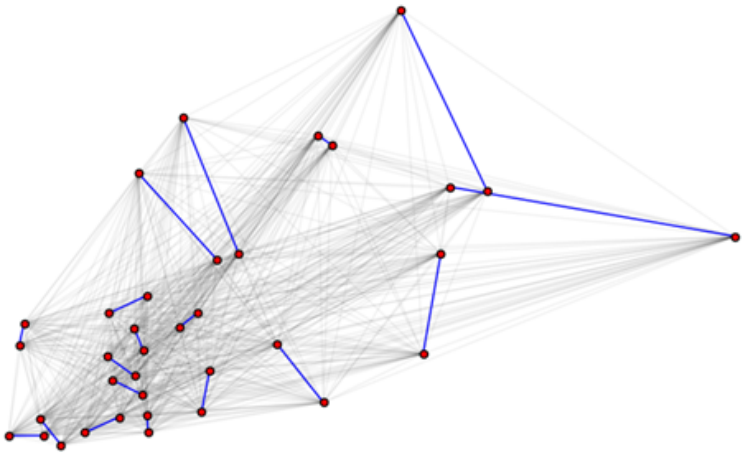
For a visual prop, the fully connected graph of odd degree node pairs is plotted above. Note thatwepreserve the X Y coordinates of each node but the edges do not necessarily represent actual trails. For example, two nodes could be connected by a single edge in this graph but the shortest path between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

Step 2.4: Compute Minimum Weight Matching

This is the most complex step in the CPP.weneed to find the odd degree node pairs whose combined sum (of distance between them) is as small as possible. So for our 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. This Maximum Weight Matching has since been folded into and maintained within the NetworkX package.

The matching output (`odd_matching_dupes`) is a dictionary. Although there are 36 edges in this matching we 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).

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 distances.

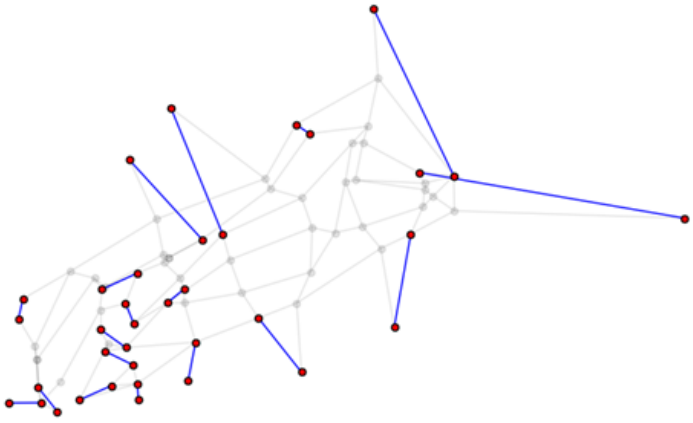


(b) *Min Weight Matching on Complete Graph*

To illustrate how this fits in with the original graph we 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). we still have a little bit of work to do to find the edges that comprise the shortest route between each pair in Step 3.

Step 2.5: Augment the Original Graph

Now we 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 we actually create the Eulerian circuit through the graph.



(c) *Min Weight Matching on original Graph*

CPP Step 3: Compute Eulerian Circuit

Now that we have a graph with an 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 that visits each edge exactly once if all nodes have even degrees. Carl Hierholzer formally proved this result later in the 1870s.

There are many Eulerian circuits with the same distance that can be constructed. we can get 90% of the way there with the NetworkX `eulerian_circuit` function. However, there are some limitations.

Limitations fixed:

- The augmented graph contains edges that didn't exist on the original graph. To get the circuit (without bushwhacking) in which breaks down these augmented edges into the shortest path through the edges that exist.
- `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 it is needed to keep track of which edges have been walked already when multiple edges exist between two nodes.

Circuit

Now let's define a function that utilizes the original graph to tell we which trails to use to get from node A to node B. Although verbose in code this logic is actually quite simple. We simply generate a Eulerian circuit using only edges that exist in the original graph.

3.2 Algorithm Code

```
from Graph import *

#CPP Step 1: Find Nodes of Odd Degree

#CPP Step 2: Find Min Distance Pairs
#Step 2.1: Compute Node Pairs

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

#Step 2.2: Compute Shortest Paths between Node Pairs
def get_shortest_paths_distances(graph, pairs, edge_weight_name):
    """Compute shortest distance between each pair of nodes in a graph.  Return a dictionary keyed on node pairs (tuples).
    """
    distances = {}
    for pair in pairs:
        distances[pair] = nx.dijkstra_path_length(graph, pair[0], pair[1], weight=edge_weight_name)
    return distances
# Compute shortest paths.  Return a dictionary with node pairs keys and a single value equal to shortest path distance.
odd_node_pairs_shortest_paths = get_shortest_paths_distances(g, odd_node_pairs, 'distance')

#Step 2.3: Create Complete Graph
def create_complete_graph(pair_weights, flip_weights=True):
    """
    Create a completely connected graph using a list of vertex pairs and the shortest path distances between them
    Parameters:
        pair_weights: list[tuple] from the output of get_shortest_paths_distances
        flip_weights: Boolean. Should we negate the edge attribute in pair_weights?
    """
    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], **{'distance': v, 'weight': wt_i})
    return g
# Generate the complete graph
g_odd_complete = create_complete_graph(odd_node_pairs_shortest_paths, flip_weights=True)

def plot_odd_G():
```

```
    # 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="red")
    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()

#Step 2.4: Compute Minimum Weight Matching

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



# Convert matching set to list of deduped tuples
odd_matching = list({tuple(sorted(pair)) for pair in odd_matching_dupes})
# Create a new graph to overlay on g_odd_complete with just the edges from the min weight matching
g_odd_complete_min_edges = nx.Graph(odd_matching)

def plot_min_weight_K():
    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)
    g_odd_complete_min_edges = nx.Graph(odd_matching)
    nx.draw(g_odd_complete_min_edges, pos=node_positions, node_size=20, edge_color='blue', node_color='red')
    plt.title('Min Weight Matching on Complete Graph')
    plt.show()

def plot_min_weight():
    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, node_color='red', edge_color='blue')
    plt.title('Min Weight Matching on Orginal Graph')
    plt.show()

#Step 2.5: Augment the Original Graph
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 edges
    graph_aug = nx.MultiGraph(graph.copy())
    for pair in min_weight_pairs:
        graph_aug.add_edge(pair[0],
                           pair[1],
                           **{'distance': nx.dijkstra_path_length(graph, pair[0], pair[1]), 'trail': 'augmented'}
                           # attr_dict={'distance': nx.dijkstra_path_length(graph, pair[0], pair[1]),
                           #            'trail': 'augmented'} # deprecated after 1.11
                           )
    return graph_aug
# Create augmented graph: add the min weight matching edges to g
g_aug = add_augmenting_path_to_graph(g, odd_matching)


#CPP Step 3: Compute Eulerian Circuit
def create_eulerian_circuit(graph_augmented, graph_original, starting_node=None):
    """Create the eulerian path using only edges from the original graph."""
    euler_circuit = []
    naive_circuit = list(nx.eulerian_circuit(graph_augmented, source=starting_node))

    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 add to eulerian circuit.
            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='distance')
            aug_path_pairs = list(zip(aug_path[:-1], aug_path[1:]))

            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))
```

```
123
124     return euler_circuit
125
126 euler_circuit = create_eulerian_circuit(g_aug, g)
```

Listing 1: Algorithm Steps

4 Visualization of the solution

Create CPP Graph

our first step is to convert the list of edges to walk in the Euler circuit into an edge list with plot-friendly attributes. Creating an edge list with some additional attributes that you'll use for plotting:

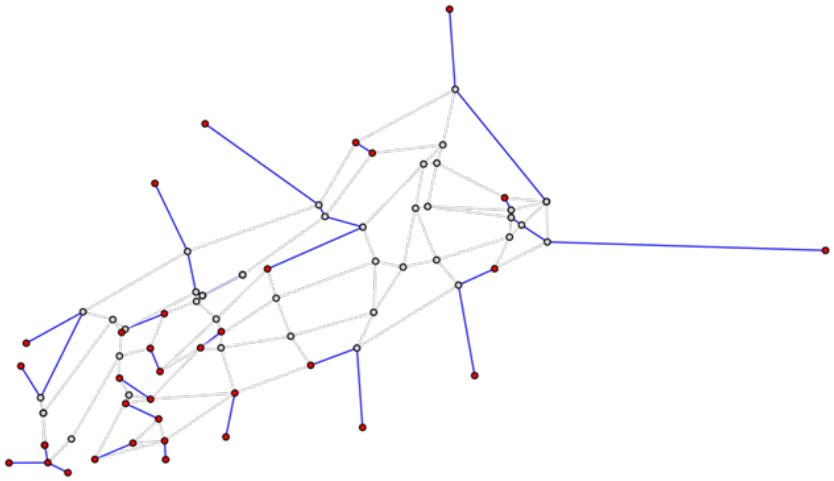
- `sequence`: records a sequence of when we walk each edge.
- `visits`: is simply the number of times we walk a particular edge.

As expected our edge list has the same number of edges as the original graph. The CPP edge list looks similar to `euler_circuit` just with a few additional attributes.

Now let's make the graph:

Visualization 1: Retracing Steps

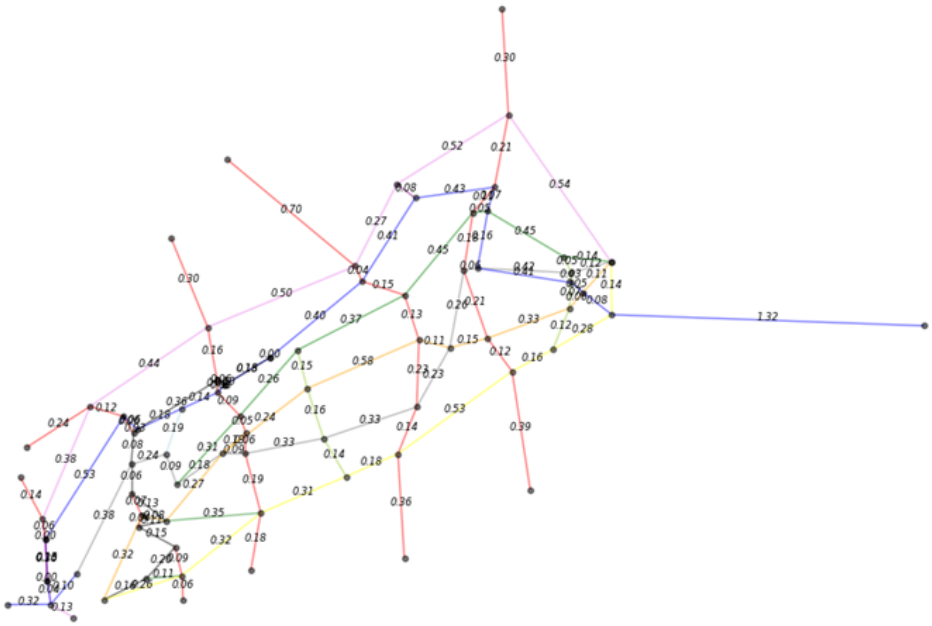
Here illustration of which edges are walked once (gray) and more than once (blue). This is the "correct" version of the visualization created in 2.4 which showed the naive (as the crow flies) connections between the odd node pairs (red). That is corrected here by tracing the shortest path through edges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



(d)

Visualization 2: CPP Solution Sequence

Here is a plot for 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.



(e) *CPP Graph Map*

5 Applications

The Chinese Postman Problem (CPP), also known as the Route Inspection Problem, is a classic problem in graph theory and combinatorial optimization. The problem seeks to find the shortest closed circuit (or walk) that visits every edge of a graph at least once. While the problem originated in the context of optimizing postal delivery routes, it has found applications in various fields. Some applications of the Chinese Postman Problem include:

1. Postal Delivery and Route Optimization: The original motivation for the problem stems from postal delivery routes, where postal workers aim to minimize the distance traveled while ensuring that every street or road is covered at least once. Solving the CPP helps postal services optimize their delivery routes, reduce travel time, and improve efficiency.

2. Telecommunications Network Design: In telecommunications networks, the Chinese Postman Problem can be applied to optimize the routing of data packets through network links. By finding the shortest circuit that traverses every link in the network, network operators can minimize data transmission delays, improve network reliability, and reduce congestion.

3. Waste Collection and Recycling Routes: Waste collection companies face the challenge of efficiently collecting garbage or recycling materials from various locations while minimizing travel distances. The CPP can be used to optimize waste collection routes, ensuring that every street or neighborhood is serviced while minimizing fuel consumption and vehicle wear and tear.

4. Cable or Wire Routing: In industries such as electrical engineering and telecommunications, the Chinese Postman Problem can be applied to optimize the routing of cables, wires, or fiber optic lines. By finding the shortest closed circuit that visits every connection point or node in a network, engineers can minimize the length of cables required and reduce installation costs.

5. Vehicle Routing and Logistics: The CPP has applications in vehicle routing and logistics, where companies aim to optimize the delivery or pickup routes of vehicles. By solving the problem, companies can minimize transportation costs, reduce delivery times, and improve customer satisfaction by ensuring timely deliveries.

6. Urban Planning and Traffic Management: Urban planners and transportation authorities can use the Chinese Postman Problem to optimize traffic flow and reduce congestion in cities. By identifying efficient routes for buses, taxis, or emergency vehicles, authorities can improve public transportation systems and reduce traffic-related emissions.

6 Conclusion

This project investigated the Chinese Postman Problem (CPP) and its application to find that It is a project on the Chinese Postman Problem and how it is applied to help determine the minimum closed path visiting every edge in the undirected graph at least once with the use of the Floyd-Warshall algorithm. Thus, the Floyd-Warshall algorithm works effectively in solving the problem of the CPP, and it gives the shortest path through all edges in a graph at least once. The following Python code implemented by the NetworkX library models a graph, the computation of all-pairs shortest paths using the Floyd–Warshall Algorithm, and the identification of the minimum postman route based on the distance matrix returned by the algorithm. NetworkX offers many conveniences to create, manipulate, and find paths in graphs that makes it an excellent tool to implement solutions for the CPP using Python. These applications can be found in a wide range of real-world situations, such as delivery route optimization, garbage collection route planning, inspection path planning, and many others that involve data collection path optimization. This work can then be used to optimize several routing and pathfinding problems in many applications, with the help of the CPP and algorithms like Floyd-Warshall that are efficient. Application of Python libraries like NetworkX makes CPP solution implementation and analysis easier, and thus this approach would be a helpful tool in solving practical problems.

7 References

[1] J. Bo, S. Xiaoying, and X. Zhibang. "A Novel Approach Model for Chinese Postman Problem". In: *Computational Intelligence and Bioinformatics*. 2006, pp. 230–237. DOI: 10.1007/11816102_25.

[2] W. L. Pearn and C. Liu. "Algorithms for the Chinese postman problem on mixed networks". In: *Computers & Operations Research* 22.5 (1995), pp. 479–489. DOI: 10.1016/0305-0548(94)00036-8.

[3] W. R. Stewart. "Chinese Postman Problem". In: *Encyclopedia of Operations Research and Management Science*. Ed. by S. I. Gass and M. C. Fu. Boston, MA: Springer, Jan. 2013. DOI: 10.1007/978-1-4419-1153-7_110.