Solving the Chinese Postman Problem on Multigraphs

Davide Pinto Teixeira
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
up202109860@fe.up.pt

Rok Mušič
University of Ljubjana
Ljubljana, Slovenia
rm26805@student.uni-lj.si

ABSTRACT

This paper addresses the Chinese Postman Problem (CPP) on multigraphs, with a focus on determining optimal traversal strategies for real-world routing applications. We explore both theoretical foundations and practical implementations, including algorithms for Eulerian paths and heuristic methods for non-Eulerian cases.

KEYWORDS

Graph, Multigraph, Chinese Postman Problem, Algorithm, Data Structure

1 INTRODUCTION

The Chinese Postman Problem (CPP)[7] is a well-known problem in graph theory that involves finding the shortest closed path or circuit that visits every edge of a graph at least once. When applied to multigraphs, which allow multiple edges between the same vertices, the problem poses unique challenges. This study investigates the characteristics of CPP in multigraphs and presents a solution leveraging classical algorithms and heuristic strategies.

2 PROBLEM DEFINITION

In a multigraph, edges can repeat between the same pair of vertices. The goal is to find a path or cycle that traverses each edge exactly once with minimal cost. We analyze whether such a traversal—called an Eulerian path[8] or circuit—is possible and what methods can be applied when it is not.

3 RELATED WORK

The Chinese Postman Problem has been extensively studied in both directed and undirected graphs. Algorithms by Edmonds and Johnson laid the foundation for practical approaches to CPP. Extensions to multigraphs have appeared in transportation networks and urban planning [1, 4]. Our approach builds upon these methods while emphasizing implementation efficiency in Python using hash-based data structures.

4 SOLUTION

The core solution strategy is divided into three steps (figure 1):

(1) Determine whether an Eulerian path or circuit exists.
(2) If it does, apply Hierholzer's algorithm to construct the path.
(3) If not, apply heuristic or approximation techniques to get a near-optimal solution.

This ensures both correctness and efficiency, adapting to the graph's structure.

EDDA'25, May 31, 2025, Porto, Porto, Portugal
2025.

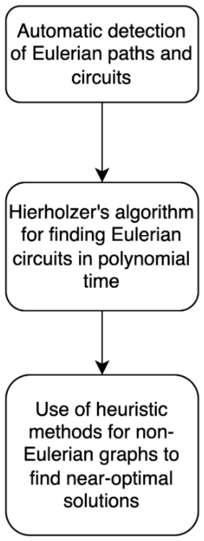
Automatic detection of Eulerian paths and circuits
Hierholzer's algorithm for finding Eulerian circuits in polynomial time
Use of heuristic methods for non-Eulerian graphs to find near-optimal solutions

Figure 1: High Level Implementation

5 DATA STRUCTURE

Before we make a deep dive into the implementation of the algorithms, let's quickly revise the data structure that allows us to efficiently store graph data and perform algorithms on it: the **multigraph**.

Our implementation should be able to store arbitrary multigraph, including nodes, node attributes, multiple edges between same pair of nodes, and edge attributes. Furthermore, we would like our implementation of the data structure to meet the following requirements:

- Add and remove nodes and edges in O(1).
- Retrieve nodes and edges (and their attributes) in O(1).

Meeting above criteria will allow our data structure to be efficient and performant.

5.1 Internal representation

To achieve the desired requirements, we decided to use two hash-maps, one for storing the edges and edge attributes → `adjacency`, and one for storing node attributes → `node_attributes`. The last part is a private `edge_key_counter`. Since there can be multiple edges between nodes u and v, we need to be able to differentiate between them; this is the goal of `edge_key_counter`. It is an integer that gets incremented every time we add a new edge, ensuring each edge is uniquely identifiable by its key.

An example of `adjacency` hash-map could look like this:

```
{
    "a": {
        "b": {
            0: {"weight": 5},
            1: {"weight": 3}
        }
    },
    "b": {
        "a": {
            0: {"weight": 5},
            1: {"weight": 3}
        }
    }
}
```

While an example of a corresponding `node_attributes` hash-map could look like this:

```
{
    "a": {"x": 12},
    "b": {"x": 42}
}
```

The choice to store each edge twice is deliberate. This way we are always guaranteed to return the neighbors and/or edges of node v in $O(1)$ time instead of $O(n)$.

Suppose we stored each edge only once in lexicographical order. If we want to return edges for node v, we would need to iterate over all nodes $u < v$, and checking if v is in `adjacency`$[u]$, which in the worst case is $O(n)$. So for the cost of double storage, we get constant retrievals.

Other choices for storing nodes and edges could be a list, or a linked list, but we again run into the problem of quick access. To check if a node v is in the graph, one would have to traverse the entire list, resulting in $O(n)$. Queues and stacks don't make much sense to use in this context either. We conclude hash-maps are the most suitable data structure for saving the nodes and edges.

5.2 Methods

To make it easy to work with the data structure, we created some convenient methods. Their API is inspired by NetworkX [6].

For convenience, we list them in total here, grouped by functionality:

- **Addition**: `add_node`, `add_nodes_from`, `add_edge`, `add_edges_from`
- **Removal**: `remove_node`, `remove_edge`
- **Membership**: `has_node`, `has_edge`
- **Node retrieval**: `nodes`, `degree`, `neighbors`
- **Edge retrieval**: `edges`, `number_of_edges`, `get_edge_data`

The implementation details of these methods should be for the most part intuitive, for example `add_node` takes in a pair of a node label and its attributes and stores them in `node_attributes`; `remove_edge` takes in u, v, and an optional *key*, and deletes either an edge between u and v with the given *key*, if *key* is given, else all of the edges between u and v; etc.

What we want to emphasize is that the concrete understanding of how these methods are implemented is not crucial for the reader to understand, apart from what they help accomplish (which should be evident from the method name), as they only provide convenience for working with the underlying representation of the multigraph, i.e. `adjacency` and `node_attributes`.

6 IMPLEMENTATION

The solution is implemented in Python 3[3]. The system includes:

- Automatic detection of Eulerian properties.
- A polynomial-time implementation of Hierholzer's algorithm.
- Heuristics for graphs without Eulerian solutions.
- Data validation routines to mitigate input errors.

Optimizations are included for handling large graphs via efficient data structures and potential parallel processing.

6.1 Hierholzer's Algorithm

If one exists, Hierholzer's algorithm is used to find an Eulerian circuit or path in a graph. The algorithm operates as follows[9]:

1. Start from a vertex with a non-zero degree (for Eulerian circuit, any vertex; for Eulerian path, a vertex with odd degree if it exists).
2. Follow edges one at a time, removing them from the graph, until returning to the starting vertex and forming a cycle.
3. If all edges in the graph are part of the cycle, the algorithm terminates, and the cycle is an Eulerian circuit.
4. Otherwise, select a vertex on the current cycle that has remaining unused edges, and repeat the process to form another cycle starting from that vertex.
5. Splice the new cycle into the previous cycle, and continue until all edges are used.

Implementation details: We represent the graph using adjacency lists to allow efficient edge removal and traversal. A stack is used to maintain the current path, and a list to store the final Eulerian circuit.

Asymptotic Analysis. Let V be the number of vertices and E the number of edges in the graph.

- The algorithm visits each edge exactly twice: once when traversing it, and once when backtracking.
- Edge removals and adjacency list updates occur in $O(1)$ time if adjacency lists are implemented with efficient data structures (e.g., linked lists or stacks).
- Vertices are pushed and popped from the stack a number of times proportional to their incident edges.

Therefore, the overall **time complexity** is:

$$O(E)$$

This is because each edge is processed a constant number of times, and vertex operations are bounded by the sum of degrees, which is $2E$.

The **space complexity** is $O(V + E)$, due to the adjacency list representation of the graph and the stack used for traversal.

Summary. Hierholzer's algorithm provides an efficient method to find Eulerian circuits or paths in graphs with linear time complexity relative to the number of edges, making it practical for large graphs especially when combined with optimizations such as adjacency lists and parallel processing of independent components.

6.2 Fleury's Algorithm

Fleury's algorithm[2] is a classical method to find an Eulerian path or circuit by carefully choosing edges to avoid disconnecting the graph prematurely. The key idea is to never remove a "bridge" edge unless no alternative exists.

(1) Verify that an Eulerian path or circuit exists in the graph.
(2) Start from a vertex with an odd degree if any exist (Eulerian path), or any vertex otherwise (Eulerian circuit).
(3) At each step, select the next edge to traverse as follows:
  - Among all edges incident to the current vertex, choose one that is *not* a bridge in the remaining graph.
  - If all available edges are bridges, choose any one of them.
(4) Remove the chosen edge from the graph and move to the adjacent vertex.
(5) Repeat until all edges have been used.

Connectivity Check. After removing candidate edges, the algorithm relies on a connectivity check to determine if they are bridges. This can be implemented via depth-first search (DFS)[10] or breadth-first search (BFS)[5].

Asymptotic Analysis. Let V be the number of vertices and E the number of edges.

- At each step, the algorithm examines adjacent edges to find a suitable non-bridge edge.
- Checking whether an edge is a bridge involves temporarily removing it and verifying connectivity.
- Connectivity check using DFS or BFS takes $O(V + E)$ time.
- Since each edge can be checked multiple times in the worst case, the total time complexity is:

$$O(E \cdot (V + E))$$

This quadratic complexity (or worse for dense graphs) makes Fleury's algorithm less efficient than Hierholzer's algorithm for large graphs. However, it is conceptually simpler and useful for educational purposes or small graphs.

Summary. Fleury's algorithm constructs an Eulerian path by carefully avoiding bridges until necessary. While it is intuitive and straightforward, its reliance on repeated connectivity checks leads to higher computational cost compared to more optimized algorithms like Hierholzer's.

6.3 CSP-Inspired Heuristic for Eulerian Path

This method attempts to find an Eulerian path using a constraint satisfaction problem (CSP) inspired heuristic. It operates in two phases:

- If the graph contains an Eulerian path or circuit, the method directly computes it using an existing algorithm.
- Otherwise, it applies a greedy heuristic that incrementally constructs a path by selecting edges according to local degree constraints, aiming to approximate a valid traversal.

Algorithm description:

(1) Begin at an arbitrary start node.
(2) Maintain a record of visited edges to avoid retracing.
(3) At each step, consider all unvisited edges incident to the current node.
(4) Select the next edge leading to the neighbor node with the lowest degree, promoting traversal through less connected parts of the graph.
(5) Mark the chosen edge as visited and move to the adjacent node.
(6) Continue until no unvisited edges remain or a dead-end is reached.

This heuristic does not guarantee a full Eulerian path when one does not exist, but attempts to maximize edge coverage by leveraging local graph structure.

Visualization: When a visualization tool is available, the algorithm highlights the traversal progress and edge selection, aiding debugging and analysis.

Asymptotic Analysis. Let V denote the number of vertices and E the number of edges.

- Each edge is visited at most once, leading to an $O(E)$ baseline for traversal.
- Selecting the next edge involves iterating over neighbors of the current node and computing degrees, which can take up to $O(V)$ in the worst case.
- Thus, in the worst case, the time complexity is:

$$O(E \cdot V)$$

- The space complexity is $O(E)$ for storing visited edges and $O(V)$ for the path.

Summary. This CSP-inspired heuristic provides a practical approach to approximate Eulerian paths in non-Eulerian graphs by greedily selecting edges based on local degree constraints. While it does not guarantee an optimal or complete solution, it effectively traverses a large portion of the graph, particularly useful when exact Eulerian paths are unavailable.

6.4 Greedy Eulerian Path Approximation

To heuristically approximate an Eulerian path without modifying the graph or adding edges, we implemented a greedy strategy that incrementally walks through unused edges. This method is beneficial for large graphs where modifying the graph (e.g., using Christofides or Edmonds' algorithm) may be computationally expensive or inappropriate.

The algorithm starts from a vertex of odd degree if one exists, or any vertex otherwise, and continues traversing unused edges greedily until no further moves are possible. While this method does not guarantee optimality, it often yields a relatively short trail in practice.

6.5 Risk Mitigation

To ensure robustness and performance, several risk mitigation strategies were employed:

- **Input Validation:** A preprocessing stage validates the graph structure to prevent traversal failures due to malformed or incomplete input data.

- **Scalability Concerns:** For large-scale graphs, the algorithm leverages optimized data structures such as adjacency lists and uses algorithmic shortcuts to maintain acceptable runtimes.
- **Fallback Mechanisms:** In non-Eulerian cases, the system gracefully falls back to heuristic-based approximations to avoid failure or excessive computation.

These strategies improve the reliability and applicability of the solution across a wide range of real-world scenarios.

7 RESULTS & EVALUATION

Test dataset

The algorithm was tested on various multigraph configurations. When there was an optimal solution (Euler path exists), the algorithms found the optimal solution every time.

The hardest part was assessing the performance on graphs where an optimal solution did not exist. We were unable to find a readily available dataset online, so we had to resort to creating our own. Due to this limitations, the size of the dataset is smaller than we would have liked.

We further wanted to see by how much does the approximate solution differ from the optimal one based on the number of nodes and the number of edges in the graph. To be able to see these results, we created three multigraph configurations for graphs on 5, 10, 15 and 20 nodes respectively. At each node count we tried to construct a graph with different number of edges and optimal path length, to gain more variety. This resulted in us having 12 graphs for evaluating our solution. A future direction would be to definitely increase this sample to get more reliable results.

Results & Analysis

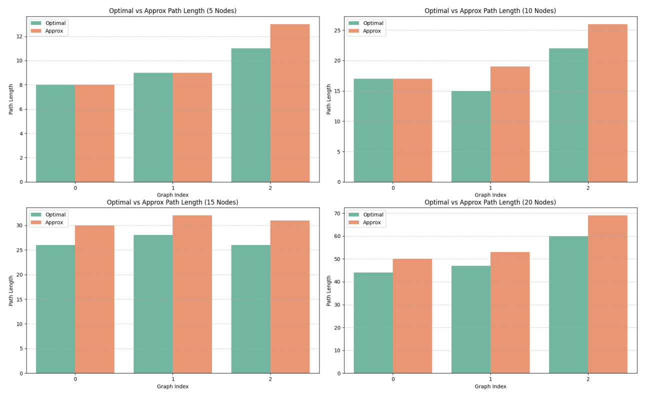

Figure 2: Optimal vs. approximate path length for all graph configurations.

We can see on Figure 2 that our approximate algorithm successfully found optimal or near-optimal solution in all cases. However, it must be mentioned that the optimal solution looks like to be diverging from the optimal in a linear manner, as seen in Figures 3 and 4. We expected to observe a more prominent pattern in the number of edges, as we thought they will be the determining factor for how well our solution performs; more edges, more possibility for the algorithm to pick a sub-optimal solution.

We, however, observe (in Figure 3) a far more recognizable linear growth of how good our solution is based on the number of nodes. The growth in the number of edges (Figure 4) still appears to be predominantly linear, but less so.

It is very important to note that the observed patterns might arise from the lack of proper testing set, and may be biased. In any case, they provide sensible and interpretable results.

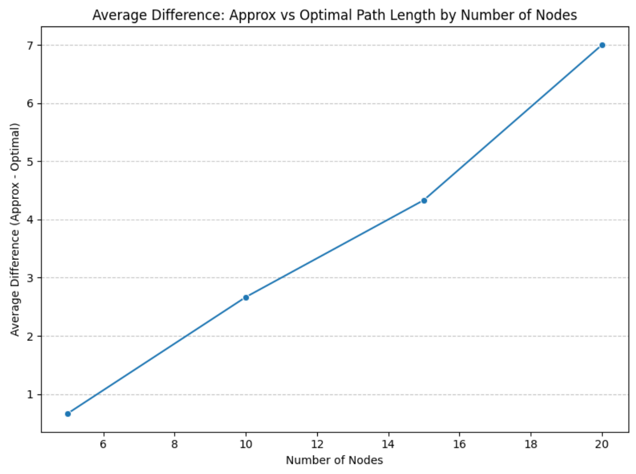

Figure 3: Optimal vs. approximate path length for different number of nodes.

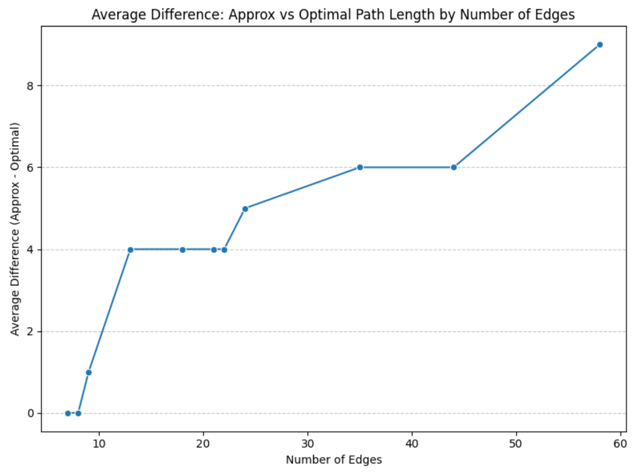

Figure 4: Optimal vs. approximate path length for different number of edges.

Time efficiency

For Eulerian graphs, the execution was both fast and exact. This is expected, as Hierholzer's algorithm is designed to efficiently find Eulerian circuits in linear time relative to the number of edges, yielding a complexity of $O(n)$. However, performance significantly declined when dealing with non-Eulerian graphs, where an approximation algorithm was required to compute a near-optimal solution by adding the minimum number of edges to form an Eulerian circuit. This step introduces additional computational overhead due to path-finding and graph traversal strategies involved in the transformation process.

As illustrated in Figure 5, the performance gap between Eulerian and non-Eulerian graphs is clearly noticeable. The execution time remains consistently low for Eulerian graphs, while it increases considerably for non-Eulerian graphs, especially as the number of nodes grows. This behavior highlights the computational cost of approximating Eulerian paths in graphs that do not naturally support them.

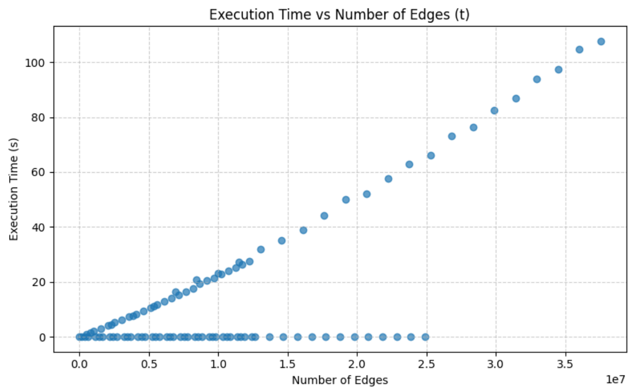

Figure 5: Execution time as a function of the number of nodes.

Summary

When there was an Euler path in the graph, our algorithms found it every time. When there wasn't one, our approximation algorithms found an optimal in near optimal solution. However, the approximate solution linearly diverges from the optimal one based on the number of nodes in the graph. The algorithms are very efficient and find solutions even on very large graphs.

8 CONCLUSION

This work demonstrates that classical algorithms, combined with smart heuristics, offer viable solutions to the CPP on multigraphs. Real-world applications such as mail delivery, snow plowing, and road maintenance can benefit from such an approach. Future work may involve refining heuristics or integrating machine learning to predict optimal traversal strategies based on graph characteristics.

9 CONCLUSION AND FUTURE WORK

Future work includes extending the system to directed multigraphs, integrating cost constraints, and benchmarking performance on real-world logistics datasets.

10 ACKNOWLEDGEMENTS

We would like to thank our advisor, Jácome Miguel Costa da Cunha, for providing guidance and suggestions. We would also like to proudly state that this report has been written in its entirety by manual human labor, no LLM-s included.

REFERENCES

[1] Jack Edmonds. 1973. Matching: A well-solved class of computational problems. *Combinatorial Structures and their Applications* 1 (1973), 89–92.
[2] M. Fleury. 1883. Deux problèmes de géométrie de situation. *Journal de mathématiques élémentaires* (1883), 257–261.
[3] Python Software Foundation. 2025. Python Programming Language. https://www.python.org/ Accessed: 2025-05-11.
[4] Hendrik W Lenstra and Alexander H G Rinnooy Kan. 1981. The complexity of finding elementary proofs of the pigeonhole principle. *SIAM Journal on Algebraic Discrete Methods* 2, 2 (1981), 265–270.
[5] Edward F. Moore. 1959. The Shortest Path Through a Maze. In *Proceedings of the International Symposium on the Theory of Switching*. Harvard University Press, 285–292.
[6] NetworkX Organization. 2025. NetworkX. https://networkx.org/ Accessed: 2025-05-29.
[7] Paul E. Black (ed.). 2020. Chinese Postman Problem. *Dictionary of Algorithms and Data Structures [online].* https://www.nist.gov/dads/HTML/chinesePostman.html Algorithms and Theory of Computation Handbook, CRC Press LLC, 1999. Accessed 11 May 2025.
[8] Lex Schrijver. 1983. Bounds on the number of eulerian orientations. *Combinatorica* 3 (Jan. 1983), 375–380.
[9] L.J. Simenthy, R. Bobanand, and M Soumya Krishnan. 2015. A comparison based analysis of euler circuit finding algorithms. 10 (01 2015), 2511–2514.
[10] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

Received May 31, 2025