One-Way Optimization in Urban Networks: Strong Orientation of Road Graphs in Tanjung Priok and Cipayung

Syaqina Octavia Rizha - 13524088
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: syaqinaoctavia@gmail.com , 13524088@std.stei.itb.ac.id

Abstract*—This paper addresses the critical challenge of urban traffic congestion by proposing a systematic method for converting road networks into fully functional one-way systems. We conduct a comparative analysis on two diverse case studies in Jakarta, Indonesia: the industrial port district of Tanjung Priok and the suburban residential area of Cipayung. The core of our methodology is modeling the network as a mixed graph, where existing one-way streets are treated as fixed directed arcs and two-way streets are treated as flexible, undirected edges requiring orientation. The process begins with a crucial validation step to ensure the underlying graph structure is 2-edge-connected, a necessary precondition for a valid solution. We then implement a deterministic, iterative algorithm that assigns a direction to each flexible edge by sequentially making choices that probably maintain the network's viability to be strongly connected. The result is a fully oriented, strongly connected directed graph for each case study, providing a robust and computationally sound framework for traffic engineers to redesign urban flow efficiently.

Keywords*—strong orientation, one-way street, traffic optimization, mixed graph, urban planning, graph theory, Jakarta.

I. Introduction

Urban traffic congestion remains one of the most critical challenges faced by growing metropolitan areas. In Jakarta, this issue manifests across diverse urban typologies, from the dense, industrial port area of Tanjung Priok to the sprawling residential district of Cipayung. Both regions experience daily traffic bottlenecks, driven by factors ranging from high commercial vehicle density to unbalanced commuter road usage. Among various strategies to manage congestion, converting selected two-way roads into one-way systems has shown significant promise in improving traffic flow and reducing conflict points. However, the decision to reconfigure road directions must be based on a systematic analysis to avoid causing isolated regions or inefficient circulation patterns.

To address this challenge, the city's road network can be represented mathematically as a graph, where intersections serve as nodes and road segments as edges. In this representation, one-way roads are modeled as directed arcs, and two-way roads as undirected edges. The key problem then becomes how to assign directions to the undirected edges such that the resulting graph is strongly connected, meaning every intersection remains reachable from any other. This requirement is vital for ensuring navigability and avoiding local disconnections after direction changes.

The problem of assigning directions while preserving strong connectivity is known in graph theory as finding a strong orientation. By leveraging the theoretical framework of strong bridges and bound edges, we can identify which roads must follow a specific direction to prevent the formation of one-way cuts. This paper proposes an algorithmic approach for generating strong orientations on real-world traffic graphs using data from Tanjung Priok. The resulting model provides a computational basis for one-way road planning that is both efficient and structurally robust, offering a scalable solution for traffic optimization in complex urban environments.

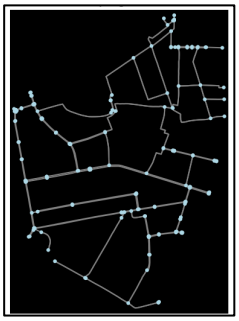

Fig. 1. Projected road network graph of Tanjung Priok generated using OSMnx, showing primary, secondary, and tertiary roads. (source: author)

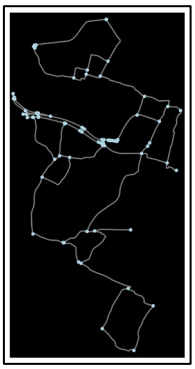

Fig. 2. Projected road network graph of Cipayung generated using OSMnx, showing primary, secondary, and tertiary roads.(source: author)

II. THEORETICAL FOUNDATIONS

A. Graph

A graph G = (V, E) consists of V, a nonempty set of vertices (or nodes) and E, a set of edges. For a G to be called graph, it should contains minimum one vertices and one edge. The number of edges that a node is connected to is called the degree. There are some kind of graph, the first one is simple graph which does not contain loops or multiple edges.

Graphs can be categorized based on how their edges are oriented. According to this classification, there are three fundamental types of graphs:

1. Directed graph
A type of graph where each edge has a direction, going from one vertex to another. In this structure, an edge from node A to node B does not imply a connection from B to A
2. Undirected graph
Consists of edges that have no direction. If two vertices A and B are connected, it is assumed that the connection works both ways.
3. Mixed graph
Combines both directed and undirected edges in a single structure.

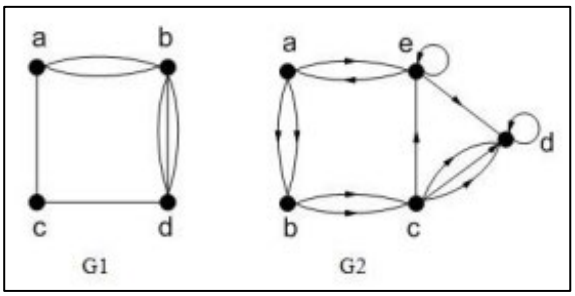

Fig. 3. G1: Undirected Graph, G2: Directed Graph
(source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

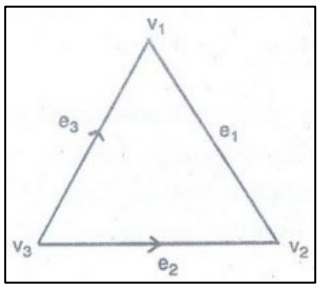

Fig. 4. Mixed Graph
(source: https://cse.poriyaan.in/topic/graphs-and-graph-models-50655/)

Another common way to differentiate graphs is by examining whether they contain loops or multiple edges. Based on this criterion, graphs fall into two main categories:

1. Simple Graph
A simple graph is defined as one that has neither loops nor multiple edges between any pair of nodes. It represents the most basic form of a graph, where each edge connects two distinct nodes with only a single connection.

2. Unsimple Graph
An unsimple graph includes at least one loop or more than one edge between the same pair of nodes. This type of graph is further divided into two subtypes:

- Multigraph: A multigraph allows multiple edges (also known as parallel edges) between the same pair of vertices. These repeated connections distinguish it from a simple graph.
- Pseudograph: A pseudograph permits the existence of loops, edges that originate and terminate at the same vertex, alongside the possibility of multiple edges between nodes.

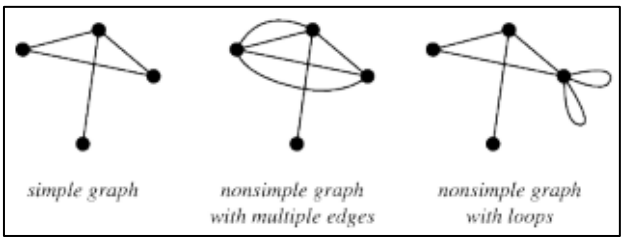

Fig. 5. Simple Graph and Unsimple Graph
(source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

B. Theorem

1. Theorem 1 (Robbins Theorm)
Let G = (V, E), a multigraph where the set of edges E is decomposable into trails. Graph G has strong orientation if and only if it is 2-edge connected.[1]
2. Theorem 2 (Boesch and Tindell)
A mixed multigraph G admits a strong orientation if and only if it is both connected and 2-edge-connected.[2]
3. Theorem 3
A 2-edge connected mixed graph G = (V, E, A) can be completed to form a strong orientation if and only if G does not admit a one-way cut.[4]

C. Definition

These are some term that will be used throughout this paper.

1. One-Way Cut
A one-way cut occurs in a mixed graph when its nodes can be divided into two groups, V1 and V2, such that all connections between the groups are exclusively directed arcs pointing in a single direction, for instance, all flowing into V1 from V2, with no return path.[4]
2. Strong orientation
Process of taking an undirected graph (or a mixed graph with two-way streets) and assigning a single, specific direction to every edge, such that the final directed graph (DiGraph) is strongly connected.
3. Strongly Connected
A directed graph is considered strongly connected if it meets the following condition:
For every possible pair of nodes in the graph, let's call them node A and node B, there must be:
   - A directed path (following the arrows) to get from A to B.
   - A directed path to get back from B to A.

III. IMPLEMENTATION

In this implementation, a combination of specialized libraries is utilized to download, process, analyze, and visualize geospatial network data. The primary tools are:

1. OSMnx: This library is fundamental to the project. It provides tools to download real-world road network data directly from OpenStreetMap. It automatically constructs a topologically corrected networkx graph object (MultiDiGraph) from the geographic data, including crucial metadata such as road types and whether a street is one-way (oneway=True).
2. NetworkX: This library is the workhorse for all graph-theoretical analysis. It is used for creating and manipulating graph structures (DiGraph, Graph, MultiDiGraph), checking fundamental graph properties such as k-edge-connectivity and strong connectivity, and executing algorithms like finding bridges and strongly connected components. It also serves as the backend for the final network visualization.
3. Matplotlib.pyplot: As Python's foundational plotting library, it is used to create the figure and axes upon which the network graph is drawn. It provides the canvas for visualization and allows for detailed customization of titles, colors, and layout, as well as saving the final output to a file.

These libraries work in concert to provide an end-to-end workflow, from raw real-world data to a fully analyzed and visualized network solution.

A. Projecting Map into a Graph using OSMnx

The first step of the implementation is to acquire and model the real-world road networks of the case study locations: Tanjung Priok and Cipayung. This is accomplished using the osmnx.graph_from ”place” function. A custom_filter is applied to select specific road types (e.g., primary, secondary, tertiary) to focus the analysis on the main traffic arteries and reduce computational complexity. OSMnx parses the OpenStreetMap data and returns a nx.MultiDiGraph object. This graph type is crucial as it accurately represents the real world, where multiple parallel edges (e.g., a flyover and the road beneath it) can exist between two intersections (nodes). Furthermore, it preserves vital metadata for each edge, most importantly the oneway attribute, which indicates if a road is already a one-way street. This initial graph serves as the raw data for all subsequent preprocessing and analysis steps.

```python
place = "Tanjung Priok, North Jakarta, Indonesia"
custom_filter = '["highway"~"primary|secondary|tertiary"]'
G_raw = ox.graph_from_place(place, custom_filter=custom_filter, network_type="drive")
```

Fig. 6. (source: author’s sourcecode)

B. Connectivity Preprocessing and Bridge Removal

Before any orientation can be attempted, the foundational structure of the graph must be validated. According to Robbins Theorem[1], a graph can only have a strong orientation if its underlying undirected structure is 2-edge-connected. A graph that is not 2-edge-connected contains at least one bridge, a critical edge whose removal would split the network into disconnected components.

This validation is performed in two stages:

1. The input graph is converted to an undirected nx.Graph to analyze its structure.
2. The networkx.bridges() function is called on this undirected graph to identify all bridge edges.

If bridges are found, the graph is not suitable for global orientation. The implemented solution follows a "pruning" strategy: all identified bridge edges are removed from the graph. After their removal, the graph may consist of several disconnected components. To ensure we work on a coherent network, the largest of these components is identified using networkx.connected_components(), and a new subgraph is created containing only the nodes and edges of this main component. This preprocessing step results in a clean, robust graph that is guaranteed to be 2-edge-connected and thus theoretically solvable.

```python
def create_custom_orientation(G_raw: nx.MultiDiGraph) -> Optional[nx.MultiDiGraph]:

    G = G_raw.copy()
    print(f"Initial Graph: {G.number_of_nodes()} node, {G.number_of_edges()} edge.")

    for u, v, data in G.edges(data=True):
        if not data.get("oneway", False):
            G.edges[u, v, 0]['flexible'] = True

    print("\n--- STEP 1: Checking 2-Edge Connectivity ---")
    G_undirected = nx.Graph(G)
    bridges = list(nx.bridges(G_undirected))
    if bridges:
        print(f"{len(bridges)} bridge founded. Erasing...")
        for u, v in bridges:
            if G.has_edge(u, v): G.remove_edge(u, v)
            if G.has_edge(v, u): G.remove_edge(v, u)

        largest_component_nodes = max(nx.connected_components(nx.Graph(G)), key=len)
        G = G.subgraph(largest_component_nodes).copy()
        print(f"Main component have {G.number_of_nodes()} node.")
    else:
        print("Graph 2-edge-connected.")
```

Fig. 7. (source: author’s sourcecode)

C. Pruning One-Way Cut Nodes

Real-world data often contains topological dead-ends or sinks, which can complicate orientation algorithms. A heuristic step is implemented to handle obvious cases of "one-way cuts" at the node level. The provided detection_oneway_cut function inspects each node in the preprocessed graph. It checks if a node acts as a sink (only has incoming oneway edges and no outgoing paths) or a source (only has outgoing oneway edges and no incoming paths), without any connecting two-way streets. Such nodes are candidates for creating isolated regions. By identifying and removing these nodes, the graph is further simplified, removing trivial dead-ends and allowing the main algorithm to focus on the more complex, cyclically connected core of the network. The largest strongly connected component of the resulting graph is then carried forward.

```python
print("\n--- STEP 2: Checking & Erasing One-Way Cut Node ---")
    one_way_cuts = detection_oneway_cut(G)
    if one_way_cuts:
        print(f"{len(one_way_cuts)} candidate of one-way cut node founded. Erasing...")
        nodes_to_remove = [item['node'] for item in one_way_cuts]
        G.remove_nodes_from(nodes_to_remove)
        G = get_largest_scc(G) # Ambil SCC terbesar setelah menghapus node
        print(f"{G.number_of_nodes()} node remaining.")
    else:
        print("One-way cut node not found.")
```

Fig. 8. (source: author’s sourcecode)

```python
def detection_oneway_cut(G: nx.MultiDiGraph) -> List[Dict[str, Any]]:
    hasil = []
    for node in G.nodes:
        in_arcs = [(u, v) for u, v, data in G.in_edges(node, data=True) if data.get("oneway")]
        out_arcs = [(u, v) for u, v, data in G.out_edges(node, data=True) if data.get("oneway")]

        two_way_neighbor = False
        for neighbor in G.neighbors(node):
            if G.has_edge(node, neighbor) and G.has_edge(neighbor, node):
                two_way_neighbor = True
                break

        if in_arcs and not out_arcs and not two_way_neighbor:
            hasil.append({"node": node, "direction": "all in"})
        elif out_arcs and not in_arcs and not two_way_neighbor:
            hasil.append({"node": node, "direction": "all out"})
    return hasil
```

Fig. 9. (source: author's sourcecode)

```python
def get_largest_scc(G_input: nx.MultiDiGraph) -> nx.MultiDiGraph:
    if not G_input or G_input.number_of_nodes() == 0:
        return G_input.__class__()
    try:
        largest_scc_nodes = max(nx.strongly_connected_components(G_input), key=len)
        return G_input.subgraph(largest_scc_nodes).copy()
    except (ValueError, TypeError):
        return G_input.__class__()
```

Fig. 10. (source: author's sourcecode)

D. *Orientation Algorithm*

This is the core of the implementation, where directions are assigned to create a strongly connected system. The approach handles a mixed graph, respecting existing traffic rules.

1. Decomposition: First, the algorithm iterates through the cleaned graph and decomposes its edges into two distinct sets:
   - Fixed Arcs: A set of directed edges from streets that were already marked as oneway=True in the original data. These directions are immutable.
   - Edges to Decide: A set of conceptually undirected edges representing all two-way streets. These are the edges that require orientation.
2. Iterative Orientation: The algorithm proceeds iteratively through the Edges to Decide. For each edge {u, v}, a direction is tentatively assigned (e.g., $u \rightarrow v$).
3. The Viability Check: It runs is_strongly_connected() on this temporary graph. If this check returns False, it means the tentative direction would create a one-way cut, and the algorithm is forced to choose the opposite direction ($u \rightarrow v$), which is guaranteed to be viable.

```python
print("\n--- STEP 3: Give Direction to Flexible Edges ---")
    fixed_arcs = set()
    edges_to_decide_set = set()
    for u, v, data in G.edges(data=True):
        if data.get('flexible', False):
            edges_to_decide_set.add(frozenset([u, v]))
        else:
            fixed_arcs.add((u, v))

    undecided_edges = [tuple(edge) for edge in edges_to_decide_set]
    oriented_arcs = list(fixed_arcs)
    nodes = list(G.nodes())

    print(f"Orientation started, {len(undecided_edges)} Flexible Edges...")
    for i, edge in enumerate(undecided_edges):
        u, v = edge[0], edge[1]
        remaining_undecided = undecided_edges[i+1:]

        if _is_still_strongly_connected(nodes, oriented_arcs + [(u, v)], remaining_undecided):
            oriented_arcs.append((u, v))
        else:
            oriented_arcs.append((v, u))
```

Fig. 11. (source: author's sourcecode)

```python
def _is_still_strongly_connected(nodes: List, oriented_arcs: List[Tuple], undecided_edges: List[Tuple]) -> bool:
    '''Cek graf strongly connected.'''
    temp_graph = nx.DiGraph()
    temp_graph.add_nodes_from(nodes)
    temp_graph.add_edges_from(oriented_arcs)
    for u, v in undecided_edges:
        temp_graph.add_edge(u, v)
        temp_graph.add_edge(v, u)
    return nx.is_strongly_connected(temp_graph)
```

Fig. 12. (source: author's sourcecode)

This process continues until all flexible edges have been assigned a direction.

E. *Final Validation*

Once the orientation algorithm completes, a final nx.DiGraph object is constructed from the complete set of oriented arcs (both the original fixed ones and the newly assigned ones). As a final verification step, the networkx.is_strongly_connected() function is called on this final graph. If the preceding steps have been executed correctly, this check should always return True, confirming that the output is a valid, fully connected one-way system. This step ensures the integrity of the final solution before it is passed to the visualization stage.

```python
print("\n--- STEP 4: Building Final Graph ---")
    final_graph = nx.MultiDiGraph()
    final_graph.add_nodes_from(G.nodes(data=True))
    final_graph.add_edges_from(oriented_arcs)

    if nx.is_strongly_connected(final_graph):
        print("Verification Success! The Final Graph is strong orientation.")
        return final_graph
    else:
        print("Verification Fail. Final Graph not Strongly Connected.")
        return get_largest_scc(final_graph)
```

Fig. 13. (source: author's sourcecode)

F. *Visualization*

Nodes are drawn on top as large, colored circles, and custom text labels (e.g., "1", "2") are rendered onto each node for easy identification. This detailed, multi-step visualization process produces an informative map that clearly distinguishes between pre-existing and newly engineered traffic flows.

```python
if G_to_plot and G_to_plot.number_of_nodes() > 0:
    print("\n--- Preparing Visualization ---")

    for i, node in enumerate(G_to_plot.nodes()):
        G_to_plot.nodes[node]["label"] = f"{i}"

    fig, ax = plt.subplots(figsize=(15, 15), facecolor='black')
    ax.set_facecolor('black')

    node_positions = {node: (G_raw.nodes[node]["x"], G_raw.nodes[node]["y"]) for node in G_to_plot.nodes() if node in G_raw.nodes}

    for u, v in G_to_plot.edges():
        is_original_oneway = (G_raw.has_edge(u,v) and G_raw.get_edge_data(u,v,0).get('oneway', False)) or \
                             (G_raw.has_edge(v,u) and G_raw.get_edge_data(v,u,0).get('oneway', False))

        color = "red" if is_original_oneway else "yellow"

        nx.draw_networkx_edges(
            G_to_plot,
            pos=node_positions,
            edgelist=[(u, v)],
            ax=ax,
            edge_color=color,
            width=1.5,
            arrows=True,
            arrowsize=12,
            arrowstyle="->",
            connectionstyle="arc3,rad=0.05"
        )

    nx.draw_networkx_nodes(
        G_to_plot, pos=node_positions,
        ax=ax, node_size=100,
        node_color="lightblue", alpha=0.9,
        edgecolors="white", linewidths=0.5
    )

    for node, pos in node_positions.items():
        label = G_to_plot.nodes[node].get("label", node)
        ax.text(
            pos[0], pos[1],
            str(label),
            fontsize=5,
            color="black",
            fontweight='bold',
            ha="center",
            va="center",
        )

    ax.text(0.01, 0.99, 'Red: Original One Way Road\nYellow: Two Way Road (Given New Direction)',
            transform=ax.transAxes, fontsize=10, color='white',
            verticalalignment='top', bbox=dict(boxstyle="round,pad=0.5", fc='black', ec='gray', alpha=0.8))

    ax.set_title(f"\n{place}", fontsize=16, color='white')
    plt.tight_layout()
    plt.show()

else:
    print("No Valid Graph.")
```

Fig. 14. (source: author's sourcecode)

IV. TESTING AND RESULT

This section presents the results of applying the strong orientation algorithm to two distinct urban case studies in Jakarta: the dense commercial network of Tanjung Priok and the residential grid of Cipayung. The final, fully oriented graph for each location is visualized, demonstrating the successful creation of a valid one-way system. A subsequent discussion will analyze the characteristics of these results and their practical implications for traffic management in diverse urban settings.

```
Projecting Map Tanjung Priok, North Jakarta, Indonesia
Initial Graph: 174 node, 325 edge.

--- STEP 1: Checking 2-Edge Connectivity ---
23 bridge founded. Erasing...
Main component have 151 node.

--- STEP 2: Checking & Erasing One-Way Cut Node ---
2 candidate of one-way cut node founded. Erasing...
149 node remaining.

--- STEP 3: Give Direction to Flexible Edges ---
Orientation started, 68 Flexible Edges...

--- STEP 4: Building Final Graph ---
Verification Success! The Final Graph is strong orientation.

--- Preparing Visualization ---
```

Fig. 15. (source: author's sourcecode)

```
Projecting Map Cipayung, East Jakarta, Indonesia
Initial Graph: 96 node, 191 edge.

--- STEP 1: Checking 2-Edge Connectivity ---
7 bridge founded. Erasing...
Main component have 73 node.

--- STEP 2: Checking & Erasing One-Way Cut Node ---
4 candidate of one-way cut node founded. Erasing...
67 node remaining.

--- STEP 3: Give Direction to Flexible Edges ---
Orientation started, 37 Flexible Edges...

--- STEP 4: Building Final Graph ---
Verification Success! The Final Graph is strong orientation.

--- Preparing Visualization ---
```

Fig. 16. (source: author's sourcecode)

This is a visualization of the graph of the area before and after the two-way lane converted into a one-way lane

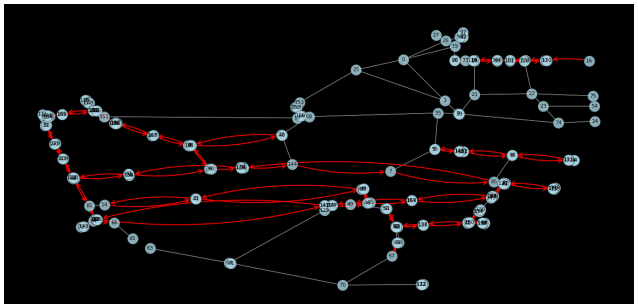

Fig. 17. Tanjung Priok Before (source: author's sourcecode)

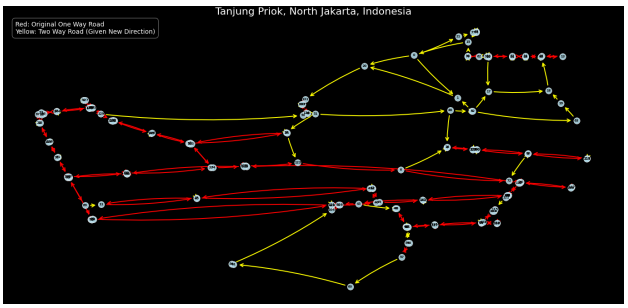

Fig. 18. Tanjung Priok After (source: author's sourcecode)

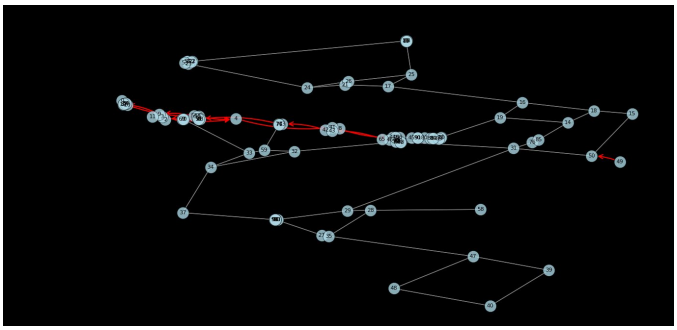

Fig. 19. Cipayung Before (source: author's sourcecode)

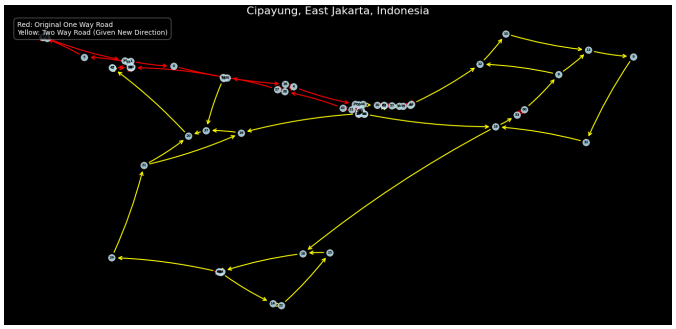

Fig. 20. Cipayung After (source: author's sourcecode)

This result validates the core theoretical premise of the study. The successful generation of a strong orientation for both Tanjung Priok and Cipayung confirms that a functional, fully connected one-way system is achievable if and only if the creation of any one-way cut is meticulously avoided. The implemented algorithm adheres to this principle by using an iterative viability check, which ensures that each assigned direction does not compromise the network's overall strong connectivity. By successfully applying this systematic approach to complex, real-world mixed-traffic networks, this research provides a powerful and verifiable method for urban planners. It serves as a valuable tool for re-engineering traffic flow with a mathematical guarantee against creating isolated zones, thereby enhancing urban mobility efficiently and safely.

V. CONCLUSION

The application of graph theory, specifically the principle of strong orientation, has been successfully demonstrated as a robust and systematic framework for re-engineering complex urban road networks. By modeling the traffic systems of Tanjung Priok and Cipayung as mixed graphs—honoring existing one-way streets while systematically assigning direction to two-way roads—the implemented algorithm effectively transformed these networks. The methodology, which includes crucial preprocessing steps like pruning bridges and isolating the largest 2-edge-connected component, successfully generated a complete and verifiable one-way system for each case study, guaranteeing a final network that is fully and strongly connected.

While the project confirmed the algorithm's logical correctness, it also highlighted challenges inherent in real-world applications, such as handling the structural imperfections of geospatial data and navigating the high computational cost of enumerating all possible solutions. Nevertheless, the results powerfully demonstrate that this graph-based approach offers a significant advantage over traditional, often intuitive, traffic planning. It provides city planners and traffic engineers with a data-driven tool to design efficient, deadlock-free traffic systems with mathematical certainty. This work not only validates a powerful theoretical concept on Indonesian case studies but also lays the groundwork for future extensions, such as incorporating traffic volume data to find not just a valid, but an optimal, traffic orientation.

VI. APPENDIX

The full codebase for this implementation can be found on GitHub. A video walkthrough explaining the paper's model and findings is also available here for viewing.

VII. ACKNOWLEDGMENT

The author wishes to express the deepest gratitude to the Lord Almighty for His guidance and blessings throughout the research and writing of "One-Way Optimization in Urban Networks: Strong Orientation of Road Graphs in Tanjung Priok and Cipayung". Through His grace, perseverance was granted to navigate the complexities of this project and bring it to a successful conclusion. Heartfelt gratitude is also extended to all who have supported this endeavor:

1. Arrival Dwi Sentosa, S.Kom., M.T., as the lecturer for the K1 IF1220 Discrete Mathematics course, for his essential guidance in applying abstract graph theory concepts to tangible, real-world transportation problems.
2. The author's parents, for their endless support, prayers, and motivation.
3. Friends and colleagues, for the constructive discussions and encouragement that were vital during the development of this paper.

The completion of this work would not have been possible without their invaluable contributions.

REFERENCES

[1] Robbins, H. E. (1939). A Theorem on Graphs, with an Application to a Problem of Traffic Control. The American Mathematical Monthly, 46(5), 281–283. https://doi.org/10.2307/2303897. Accessed on June 18, 2025.

[2] Boesch, F., & Tindell, R. (1980). Robbins's theorem for mixed multigraphs. The American Mathematical Monthly, 87(9), 716–720. https://doi.org/10.2307/2321858. Accessed on June 18,

[3] Aamand, A., Hjuler, N., Holm, J., & Rotenberg, E. (2017). *One-way trail orientations*. arXiv. https://doi.org/10.48550/arXiv.1708.07389. Accessed on June 18, 2025.

[4] Conte, A., Grossi, R., Marino, A., Rizzi, R., & Versari, L. (2016). *Directing road networks by listing strong orientations*. arXiv. https://doi.org/10.48550/arXiv.1506.05977. Accessed on June 18, 2025.

[5] Thomassen, C. (2014). Strongly 2-connected orientations of graphs. *Journal of Combinatorial Theory, Series B, 109*, 108–118. https://doi.org/10.1016/j.jctb.2014.07.004. Accessed on June 19, 2025.

[6] Rosen, K. H. (2012). *Discrete mathematics and its applications* (7th ed.). McGraw-Hill. Accessed on June 18, 2025.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Juni 2025

Syaqina Octavia Rizha–13524088