Optimizing Urban Transportation Networks: Comparative Analysis of Dijkstra's Algorithm in Graph-Based Shortest Path Solutions

Varel Tiara and 13523008
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13523008@std.stei.itb.ac.id, vareltiara@gmail.com

***Abstract*— Urbanization in cities such as Jakarta and Bandung is rapid. This is problematic for transportation in general; therefore, its design calls for optimization solutions to improve overall efficiency. In the process, this work discusses how to apply Dijkstra's Algorithm on the shortest path problem of the graph-based representation for modeling urban transportation. Dijkstra's algorithm finds an optimal path due to its weighted graph that represents road conditions and travel metrics. Experimental investigations underline its efficacy and shortcomings regarding dynamic urban contexts. This paper offers insights into implementing adaptive solutions for real-world transportation systems.**

***Keywords*—Dijkstra's Algorithm, graph theory, shortest path problem, transportation networks**

I. Introduction

In today's rapidly urbanizing world, the efficiency of transportation networks is critical to shaping economic growth, environmental sustainability, and quality of life among urban residents. Megacities like Jakarta and Bandung in Indonesia epitomize these challenges of rapid urbanization and population growth. Jakarta usually suffers from heavy congestion that strands commuters for hours, while congestion spikes periodically in Bandung, especially on weekends and holidays when tourism peaks. In the end, productivity has gone down due to traffic congestion, as well as further degradation of the environment, with the high fuel consumption and resulting emissions.

This paper proposes a solution using a graph theory mathematical framework that models the transportation network of interconnected nodes representing intersections and edges representing roads. This abstractive notion of complex networks allows algorithms to effectively compute optimal pathways. In the context of this paper, this is achieved by Dijkstra's Algorithm: a powerful algorithm useful in finding the shortest paths on weighted graphs. By systematically exploring the graph, Dijkstra's Algorithm ensures that the most cost-effective route can be identified, considering travel distance, road conditions, and traffic density. Dijkstra's Algorithm is a part of navigation systems like Google Maps; it plays a vital role in helping millions of people navigate urban landscapes every day.

Dijkstra's Algorithm has the advantage of being comparatively simple and with a high success rate for static graphs. In natural conditions, real-life traffic scenarios may change frequently, even at any minute, considering road congestion or blockages in cities. Thus, despite these specific limitations, Dijkstra's Algorithm retains a crucial role in research toward the conceptualization of truly adaptive routing systems.

The paper seeks to discuss some of the basic concepts and applications of Dijkstra's Algorithm in an urban transportation network. The discussion will outline the capacity of the algorithm, from its mathematical formulation to its practical implementation, to address real-world challenges in Indonesian cities.

II. Basic Theory

A. Graph Theory

Graph theory offers a mathematical structure to model and analyze various forms of networks in transportation systems. A graph can be defined mathematically by a set of nodes or vertices connected using a set of edges. In this context, considering urban transportation networks, nodes represent points of interest while edges describe the roads, railways, or bus routes connecting two locations.

Formally, a graph G is defined as:

$$G = (V, E)$$

Where:

- V is a non-empty set of vertices (nodes), representing locations in the network, such as intersections or stations.

$$V = \{v_1, v_2, \ldots, v_n\}$$

- E is a set of edges (links) that connect pairs of vertices, representing roads or transport routes

between locations.

$$E = \{e_1, e_2, \ldots, e_m\}$$

In this definition:

- V must not be empty, meaning a graph cannot exist without at least one vertex.
- E can be empty, meaning a graph can exist even if no edges (connections) are present.

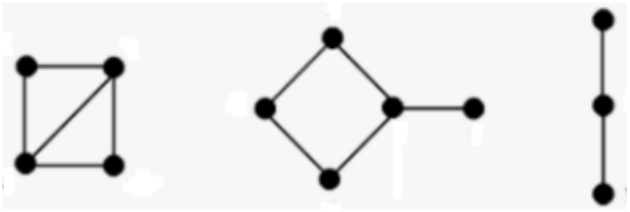


Image 1. Graph
(https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

A graph may be undirected or directed. In the latter, the edges are oriented; there is a sense of direction, permitting travel along an edge in only one direction. For example, one-way streets, and bus routes. Urban transportation networks use directed edges to model one-way streets or one-directional public transit routes.

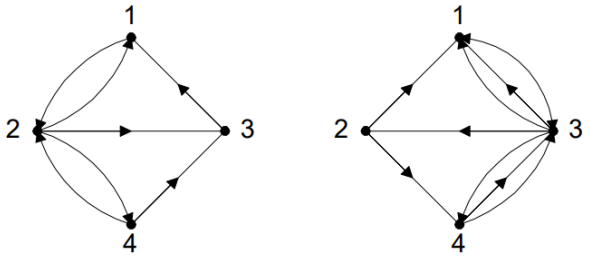


Image 2. Directed Graph
(https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

In undirected graphs, edges have no direction; thus, one can travel both ways on the same edge, or in other words, edges are bidirectional. This graph would model systems where the relations between nodes are symmetric, such as regular two-way streets in an urban area.

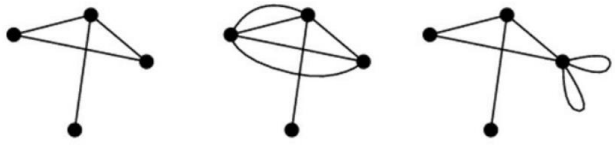


Image 3. Undirected Graph
(https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

Besides directionality, graphs can also be weighted, with each edge having a weight reflecting some attribute of the connection. This could be in the form of travel time, distance, congestion, cost, or even energy consumption. Weighted graphs are particularly useful for modeling real transportation systems where routes are not equal in terms of travel costs.

A weighted graph

$$G = (V, E, w)$$

consists of:

- V: A set of vertices (nodes).
- E: A set of edges (links) connecting pairs of vertices.
- w: A function that assigns a weight to each edge, where w(e) represents the weight of edge e.

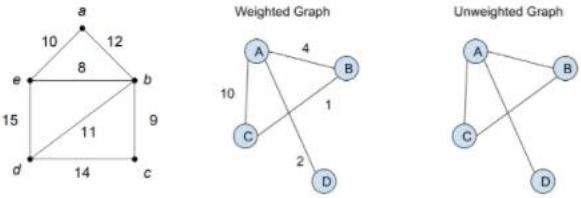
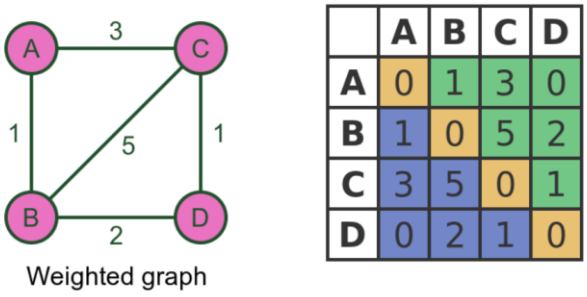


Image 4. Weighted Graph
(https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

The edges in a weighted directed graph model the "cost" of moving from one node to another. Costs could depend on distance, time, and road conditions, among others. Consider a city like Jakarta with congested traffic: the weight may be taken as the average time that it takes to travel from one intersection to another. This weight can change within the day.

Transport networks in cities are represented in two ways:

1. Adjacency Matrix: This representation is one in which the relationship between the nodes is depicted by using a square matrix. Each entry within the matrix describes the weight of the edge that connects two nodes. In the case of no direct relationship, the entry in the matrix is usually filled with a value of infinity or null. This usually applies to smaller networks since it allows direct access to any node connection and may be unsuitable for sparse graphs.



	A	B	C	D
A	0	1	3	0
B	1	0	5	2
C	3	5	0	1
D	0	2	1	0

Image 5. Adjacency Matrix
(https://graphicmaths.com/computer-science/graph-theory/adjacency-matrices/)

2. Adjacency List: This is a representation where each node contains a list of nodes directly connected to it and the respective weights of the edges. It's more memory-efficient for very big, sparse graphs, as it only stores the edges that exist. In an urban transportation system, there are many nodes with relatively fewer connections due to the limited direct routes; thus, the adjacency list is often used.

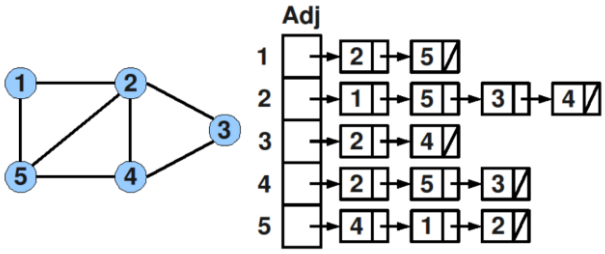


Image 5. Adjacency List
(https://graphicmaths.com/computer-science/graph-theory/adjacency-matrices/)

Both approaches allow the transport system to be effectively modeled as a graph and form the fundamental framework on which an array of graph-based algorithms, such as Dijkstra's Algorithm, can be implemented for finding the best route or controlling traffic flow.

In graph theory, the degree of a node is defined by the number of edges incident to the node. It gives the extent of its connectivity within that graph:

- In-degree: The number of incoming edges to a node (relevant in directed graphs).
- Out-degree: The number of outgoing edges from a node (also relevant in directed graphs).
- Degree: For undirected graphs, the degree simply refers to the total number of edges incident to a node, which reflects how well-connected a particular location is within the network.

This will be helpful in urban transportation networks where, for example, the degree of a node might provide some information. A high degree may mean an intersection with many incoming and outgoing roads; it is probably a busy traffic hub, whereas a node with a low degree may mean fewer roads and thus may be an insignificant location in the network.

Graphs are used to represent complex road networks or multi-modal transportation systems in urban transportation systems. Graph theory abstracts the system into nodes and edges, thus enabling the use of efficient algorithms to compute the best paths between any two nodes in the network, helping commuters navigate efficiently.

Graph theory can be applied to help transportation engineers and developers identify bottlenecks, optimize traffic flow, and reduce travel times, such as in cities with high congestion like Jakarta. Graph theory algorithms, such as Dijkstra's Algorithm, are the backbone of real-time traffic management systems that update routes with the shortest distance based on current conditions such as congestion and road closures.

B. The Shortest Path Problem

The shortest path problem is a widely used concept in graph theory, aimed at finding the path between two nodes in a graph that minimizes the total weight of the edges traversed. In other words, this means finding the best route between two locations in an urban transportation network concerning certain criteria such as travel time, distance, or cost. The capability to determine the shortest path enables solutions for route optimization, traffic management, and efficient navigation.

In graph theory, a path is defined as a series of edges in which a sequence of nodes is concatenated by each edge connecting two nodes adjacently. The total weight of a path is the sum of the weights of all edges included in the path. For instance, in a transport network, it would be the sequence of roads or public transit routes that would be followed to travel from one location to another, and the weight here could be the overall metrics of travel time or distance.

The paths in a graph can be differentiated by the nature of nodes and edges that make up the path. A simple path is a path where no nodes are repeated, meaning that the traversal will not go back to any previously visited location. This is very important in urban transportation networks, as loops or going back would waste time or increase costs unnecessarily. A cycle, on the other hand, is a path that starts and ends with the same node concept hardly applicable in the usual shortest path calculations but useful in the detection of redundancies or inefficiencies in a network. In an urban transportation system, the shortest path problem involves evaluating all possible paths between two nodes and selecting the one with minimum weight. In large dense networks, it is a problem to find an optimum path among hundreds of thousands of nodes and edges; hence, special algorithms should be applied to do this work quickly and efficiently. It is not confined to just a single pair of nodes but can also include all pairs of nodes, like finding the shortest path between all pairs of nodes for network-wide traffic optimization.

Shortest path algorithms find a variety of real-life applications in transport systems and other areas of application. These algorithms are used to estimate the fastest or shortest route in navigation systems, such as Google Maps and Waze, considering traffic congestion, the closure of certain lanes or roads, and even accidents. They work based on turn-by-turn data continuously updated to make sure that commuters can reach their destination efficiently. The algorithms of the shortest paths are also applied in intelligent public transit systems, which enable passengers to identify the most suitable route and schedule, considering buses, trains, and subways.

Traffic management systems also apply shortest-path algorithms to reduce congestion and optimize traffic

flow. Such systems dynamically reroute vehicles and adjust signal timings to reduce delays and improve the overall efficiency of transportation. Beyond urban transport, shortest-path algorithms are integral to applications in logistics and supply chain management, where they're used to optimize delivery routes and minimize transportation costs.

C. Dijkstra's Algorithm

Dijkstra's Algorithm is among the most useful methods of solving the shortest path problem in a graph. It very efficiently computes the minimum-cost path from a source node to all other nodes in the graph and hence finds applications in navigation systems and traffic management.

It systematically explores all possible routes from the source node, updates the shortest known distance to each node, and ensures every node is visited exactly once. The algorithm uses weights of edges, which are criteria specified for a graph such as travel time, distance, or congestion amongst others, in calculating the cost of a certain path. The step-by-step breakdown of Dijkstra's Algorithm is shown below:

1. Initialization:
    - Assign all nodes a preliminary distance of infinity, except for the source node to which a distance of 0 is assigned. This reflects that the shortest path to the source node is already known.
    - Mark all nodes as unvisited and create a priority queue to manage the nodes based on their tentative distances.
2. Current Node Selection:
    - Select the unvisited node with the smallest tentative distance as the current node. If the smallest distance among unvisited nodes is infinity, then the algorithm stops since no more reachable nodes are available.
3. Exploration of Neighbors:
    - For each unvisited neighbor of the current node, calculate its tentative distance by adding to the distance to the current node the weight of the edge between them.
    - If this tentative distance is smaller than the recorded distance for that neighbor, renew the distance of the neighbor and store the current node as its predecessor.
4. Mark Node as Visited:
    - Once all neighbors of the current node have been assessed, mark the current node as visited. A visited node will not be revisited, and that ensures that the algorithm will eventually terminate.
5. Repeat:
    - For the next smaller tentative distance node in the unvisited node set repeat, until all the nodes have been visited or if the shortest route to the destination node is found.

The algorithm operates on a weighted graph, where each edge $e \in E$ has a weight $w(e)$, representing the cost of traversing the edge. At each step, the algorithm ensures that the shortest known distance $d(u)$ from the source node s to any node u satisfies:

$$d(u) = min\{d(v) + w(v,u)\}$$

for all neighbors v of u, where $w(v,u)$ is the weight of the edge connecting v and u. This iterative update process is known as the relaxation step.

The algorithm maintains an invariant that once a node has been marked as visited, its shortest distance is fixed and won't change thereafter.

The efficiency in Dijkstra's Algorithm lies in the way it manages nodes and edges using appropriate data structures:

- Basic Implementation: Using a simple array or list to store distances, the algorithm has a time complexity of $O(V^2)$, where V is the number of vertices in the graph. This approach is suitable for smaller graphs or dense networks.
- Optimized Implementation: Using a priority queue (e.g., a binary heap or Fibonacci heap), the time complexity reduces to $O((V + E)\log V)$, where E is the number of edges. This makes the algorithm much more efficient for sparse graphs, where the number of edges is significantly smaller than the square of the number of vertices.

Dijkstra's Algorithm's simplicity and efficiency have made it a cornerstone of pathfinding in various applications, particularly in static graphs where edge weights remain constant. Its performance, however, can be challenged in dynamic graphs or real-time systems where frequent updates to edge weights occur, requiring adaptations or alternative algorithms.

D. Challenges in Applying Dijkstra's Algorithm in Dynamic Environments

Application of Dijkstra's Algorithm in urban transportation networks faces some challenges, more so in a dynamic environment due to rapid changes in conditions. These result mainly from the differences between a static and a dynamic graph with the need for updates in real-time.

Transportation networks can be modeled as either static or dynamic graphs. In static graphs, nodes, edges, and edge weights are fixed during algorithm execution. That would be applicable for idealized or planned networks in which conditions such as travel times or road availability do not change. However, static graphs cannot capture the complexity of real transportation systems.

Dynamic graphs consider real-time changes in the network, such as traffic congestion, road closure, or

accidents. In this model, edge weights can change, for example, travel times or congestion levels the structure of the graph itself may change, such as new edges added (newly opened roads) or existing ones removed (closed lanes). The variability in dynamic graphs introduces challenges in maintaining accurate and efficient route recommendations.

The real-time adaptation of Dijkstra's Algorithm is fraught with several critical issues.

- Frequent Updates: Real-time transportation systems demand constant updates of edge weights and graph structure. Running Dijkstra's Algorithm for minor changes may be inefficient, especially for large-scale networks.
- Computational Overhead: The traditional implementation of Dijkstra's Algorithm has a time complexity of $O(V^2)$ or $O((V+E)\log V)$ when using a priority queue. Repeating this process for every update can strain computational resources, particularly in dense urban graphs with many nodes and edges.
- Decision-Making Delay: Any latency in updating graphs and recalculating the shortest paths results in stale or, worse, nonoptimal route recommendations being made to commuters. For example, unexpected congestion could not be instantly caught by the algorithm output to provide efficiency.
- Data Integration: Real-time integration of data sources, such as live feeds over traffic, sensor data from the IoT, and weather updates. These feeds may be inaccurate, incomplete, and/or late. This makes it hard for a consistent pathfinding algorithm to support.

A variety of strategies are proposed in this regard to ensure smooth and greater functionality for an enhanced algorithm in real-life dynamic situations:

- Incremental Update Methods: Instead of recalculating the shortest path from scratch, incremental algorithms update the current solution to accommodate changes in the graph. This approach is computationally efficient and more suitable for real-time applications.
- Heuristic-Based Alternatives: Algorithms such as A* make use of heuristics that prefer certain nodes or paths over others to reduce superfluous exploration and computation time. Such algorithms are quite effective in dynamic scenarios where speed is crucial.
- Dynamic Adaptations of Dijkstra: Variants like Dynamic Dijkstra are designed to handle changes in edge weights or graph structure without full recalculation.
- Integration with Advanced Frameworks: Combining Dijkstra's Algorithm with real-time data processing systems can offer more responsive and adaptable pathfinding solutions. For example, traffic management systems can utilize streaming data to dynamically adjust routes and minimize commuter delays.

III. IMPLEMENTATION

A. Graph Representation

The Graph class represents a directed, weighted graph. It uses arrays to store vertex details and edges, providing a clear structure for graph-related operations.

1. Class Initialization
   The graph stores:
   - vertexNameArray: List of vertex names.
   - vertexIndexMap: Dictionary mapping vertex names to indices.
   - vertexPositionArray: List of (x, y) positions of vertices.
   - edgeArray: List of directed edges with weights.

```
class Graph:
    def __init__(self):
        self.vertexNameArray = []
        self.vertexIndexMap = {}
        self.vertexPositionArray = []
        self.edgeArray = []
```

Image 6. Class Initialization
(source: writer's archive)

B. Adding Vertices and Edges

Vertices and edges are added using dedicated methods.

1. Adding Vertices
   The addVertex() method takes a vertex name and its position (x, y). It updates the vertex name array, index map, and position array.

```
# Add a new vertex with its name and position
def addVertex(self, name, x, y):
    self.vertexIndexMap[name] = self.vCount()
    self.vertexNameArray.append(name)
    self.vertexPositionArray.append((x, y))
```

Image 7. Adding Vertices
(source: writer's archive)

2. Adding Edges
   The addEdge() method establishes a

directed connection between two vertices with a given weight.

Image 8. Adding Edges
(source: writer's archive)

C. Finding The Shortest Path (Dijkstra's Algorithm)

The dijkstra() method calculates the shortest paths from a source vertex using a priority queue.

1. Initialization
   Distances to all vertices are initialized to infinity (inf), except for the source vertex.

Image 9. Dijkstra's Algorithm Initialization
(source: writer's archive)

2. Edge Relaxation
   Edges are relaxed to update distances when shorter paths are found.

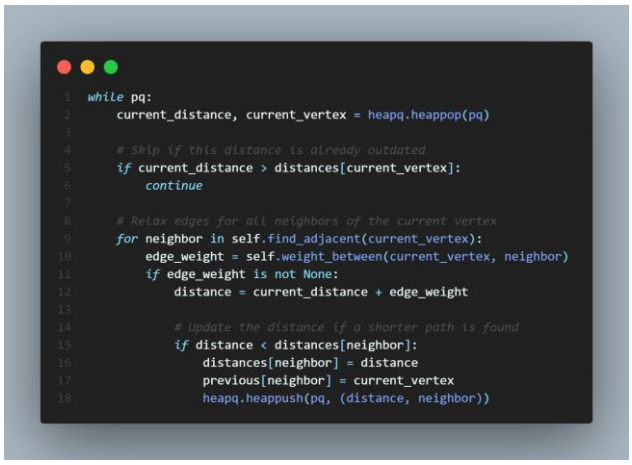


Image 10. Edge Relaxation Process
(source: writer's archive)

3. Tracking State
   States of the distance map are stored for visualization purposes.

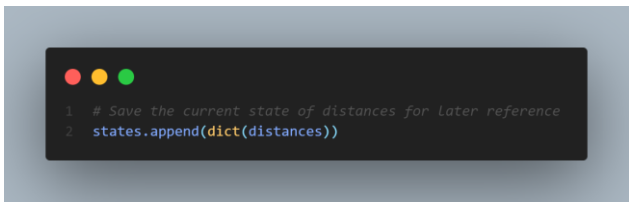


Image 11. Tracking State
(source: writer's archive)

Image 12. Dijkstra's Algorithm
(source: writer's archive)

D. Visualization

Graph visualization is performed using the draw_graph() method, leveraging the NetworkX library

1. Node and Edge Representation
   Nodes and edges are added to a NetworkX-directed graph.

Image 13. Node and Edge Representation
(source: writer's archive)

2. Displaying the Graph
   Using Matplotlib, nodes, edges, and labels are displayed with custom styling.

```
# Draw the graph
plt.figure(figsize=(10, 7))
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=3000, font_size=10, font_weight='bold', arrowsize=20)
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=9)
plt.title("Graph Visualization")
plt.show()
```

Image 14. Displaying the Graph
(source: writer's archive)

```
# Draw the graph using NetworkX and Matplotlib
def draw_graph(self):
    G = nx.DiGraph()

    # Add vertices with positions
    pos = {}
    for i, (name, (x, y)) in enumerate(zip(self.vertexNameArray, self.vertexPositionArray)):
        G.add_node(name)
        pos[name] = (x, y)

    # Add edges with weights
    for edge in self.edgeArray:
        v1 = self.vertexNameArray[edge[0]]
        v2 = self.vertexNameArray[edge[1]]
        weight = edge[2]
        G.add_edge(v1, v2, weight=weight)

    # Draw the graph
    plt.figure(figsize=(10, 7))
    nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=3000, font_size=10, font_weight='bold', arrowsize=20)
    edge_labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=9)
    plt.title("Graph Visualization")
    plt.show()
```

Image 15. Graph Visualization
(source: writer's archive)

E. Testing the Implementation

The graph functionality is tested by creating vertices, adding edges, and calculating shortest paths.

1. Example Graph

```
# Testing the implementation
def test_graph_visualization():
    g = Graph()

    # Add vertices
    g.addVertex('s', 0, 0)
    g.addVertex('t', 1, 0)
    g.addVertex('x', 2, 0)
    g.addVertex('y', 1, 1)
    g.addVertex('z', 2, 1)

    # Add edges with weights
    g.addEdge('s', 't', 10)
    g.addEdge('s', 'y', 5)
    g.addEdge('t', 'x', 1)
    g.addEdge('t', 'y', 2)
    g.addEdge('y', 't', 3)
    g.addEdge('y', 'x', 9)
    g.addEdge('y', 'z', 2)
    g.addEdge('z', 'x', 4)
    g.addEdge('x', 'z', 6)
```

Image 15. Example Graph
(source: writer's archive)

2. Output Visualization
   The graph is visualized with labels and edge weights.

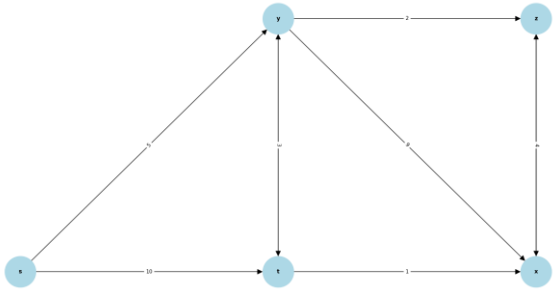


Image 16. Output Visualization
(source: writer's archive)

IV. Conclusion

The application and analysis of Dijkstra's Algorithm in the urban transportation network have demonstrated its conceptual strengths as well as realistic limitations while tackling current-day traffic challenges. This paper further demonstrates that, with its mathematical rigor and guarantee of optimum solution, Dijkstra's algorithm remains a very important tool for route optimization in transportation networks under stationary conditions. The Python-based implementation with visualization features of this algorithm has proved to be effective for the calculations of shortest paths and assures promising potential for large-scale transportation management systems.

However, the research also identifies the limitations of the application of Dijkstra's Algorithm to dynamic urban environments; for example, in rapidly changing cities such as Jakarta and Bandung, the algorithm gives good results under static conditions. However, for real-time dynamic operations, the complexity of computation, along with more frequent recalculations, badly affects its efficiency. These results indicate that this approach will require sophisticated modifications in the real world.

Key conclusions from this paper are as follows:

1. Static Network Performance: The algorithm serves as a strong starting point for route optimization with valuable initial traffic planning and network analysis.
2. Visualization Tools: These accompanying visualization tools give further insight into the operations of the algorithm and will be useful in transportation network planning and analysis.
3. Dynamic Environments Challenges: Limitations on real-time updates call for further research in adaptive algorithms and optimization techniques for more responsive applications.

In addition, future work can be done by including real-time traffic data, dynamic updates of weights, and hybrid approaches involving Dijkstra's Algorithm integrated with machine learning techniques. All these will, no doubt, enhance its adaptability and responsiveness in dynamic traffic scenarios. The work can also be extended by availing cutting-edge smart city technologies such as

IoT sensors and real-time traffic monitoring systems to come up with holistic solutions for urban transportation problems.

This paper will contribute to the wider understanding of graph-based solutions in urban transportation and lays a foundation for further work on advancing intelligent traffic management systems. As cities grow and their traffic patterns increase in complexity, so too will the foundations outlined here enable innovative, effective transportation solutions.

V. Appendix

- Github Repository for this paper: https://github.com/varel183/Optimizing-Urban-Transportation-Networks
- YouTube video: https://youtu.be/W233D2MKjcw

VI. Acknowledgment

First and foremost, the researcher would like to thank the Lord for His Grace and kindness. The researcher would also like to extend the biggest gratitude to Dr. Rinaldi, Dr. Rila Mandala, and Arrival Dwi Sentosa, M.T., and Arrival Dwi Sentosa, M.T for their invaluable guidance and teaching throughout the course, which have been so vital in writing this paper.

As the author of this paper, I would like to extend my deepest appreciation to all those who have been supportive and a source of inspiration to me during the writing of this paper, thus enabling me to bring it to its successful completion titled "Optimizing Urban Transportation Networks: Comparative Analysis of Dijkstra's Algorithm in Graph-Based Shortest Path Solutions."

I would also like to thank those authors whose works were the foundation of this paper, as well as the journals and articles that gave me so much valuable insight and greatly added to my knowledge regarding the topic.

This paper would not have been possible without the invaluable contribution of the individuals listed above. The support and assistance that I received all served to propel me to successfully completing the paper. Many thanks for the help and support that I needed and received. It is hoped that this paper could also be helpful to others.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[3] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Mathematical Programming*, vol. 73, no. 2, pp. 129–174, 1996.

[4] Brilliant, "Dijkstra's Algorithm," available online: https://brilliant.org/wiki/dijkstras-short-path-finder/, [Accessed: 4 Jan. 2025].

[5] S. K. Das and M. Kumar, "A comparative study on shortest path algorithms," *International Journal of Computer Applications*, vol. 182, no. 23, pp. 23–27, 2018.

[6] A. O. Folake, A. Eneh, E. Emmanuel, D. Ebem, and T. Bashiru, "Analysis and application of shortest path algorithms on urban road networks," *International Journal of Engineering Inventions*, vol. 13, no. 2, pp. 94–97, Feb. 2024.

[7] Y. F. Riti, J. S. Iskandar, and Hendra, "Comparison analysis of graph theory algorithms for shortest path problem," *Informatics Study Program, Faculty of Engineering Darma Cendika Catholic University*, Surabaya, Indonesia, vol. 12, no. 3, pp. 415–424, Nov. 2023.

[8] W. Alhoula, *Shortest Path Algorithms for Dynamic Transportation Networks*, PhD thesis, The Nottingham Trent University, May 2019.

[9] M. Miler, D. Odobašić, and D. Medak, "The shortest path algorithm performance comparison in graph and relational database on a transportation network," *Promet-Traffic & Transportation*, vol. 26, no. 1, pp. 75–82, Feb. 2014.

[10] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan, "Faster algorithms for the shortest path problem," Working Paper, Alfred P. Sloan School of Management, Massachusetts Institute of Technology, W.P. No. 2043-88, Apr. 1988.

[11] GeeksforGeeks, "Introduction to Dijkstra's Shortest Path Algorithm," available online: https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/?ref=header_outind, [Accessed: 4 Jan. 2025].

[12] M. Burst, "Dijkstra's Algorithm," available online: https://github.com/mburst/dijkstras-algorithm/, [Accessed: 4 Jan. 2025].

[13] V. Piotti, "Dijkstra Solver Bootstrap Vis," available online: https://github.com/vittorioPiotti/DijkstraSolver-Bootstrap-Vis, [Accessed: 4 Jan. 2025].

[14] Transport Geography, "Graph Theory: Definition & Properties," available online: https://transportgeography.org/contents/methods/graph-theory-definition-properties/, [Accessed: 4 Jan. 2025].

[15] Wikipedia, "Shortest Path Problem," available online: https://en.wikipedia.org/wiki/Shortest_path_problem, [Accessed: 4 Jan. 2025].

[16] Wikipedia, "Dijkstra's Algorithm," available online: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm, [Accessed: 4 Jan. 2025].

[17] R. Munir, *Graf (Bagian 1)*, Program Studi Teknik Informatika, STEI-ITB, available online: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf, [Accessed: 4 Jan. 2025].

[18] R. Munir, *Graf (Bagian 2)*, Program Studi Teknik Informatika, STEI-ITB, available online: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf, [Accessed: 4 Jan. 2025].

[19] R. Munir, *Graf (Bagian 3)*, Program Studi Teknik Informatika, STEI-ITB, available online: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf, [Accessed: 4 Jan. 2025].

[20] S. K. Bisen, "Application of Graph Theory in Transportation Networks," *International Journal of Scientific Research and Management (IJSRM)*, vol. 5, no. 7, pp. 6197–6201, Jul. 2017.

[21] Graphic Maths, "Adjacency Matrices in Graph Theory," available online: https://graphicmaths.com/computer-science/graph-theory/adjacency-matrices/#google_vignette, [Accessed: 4 Jan. 2025].

Statement

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 8 Januari 2024

Varel Tiara dan 13523008