Optimizing Road Networks with Graph Theory Selecting the Most Valuable Connections Between Nodes

Yonatan Edward Njoto - 13523036[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523036@std.stei.itb.ac.id, [2]yonatan.njoto@gmail.com

***Abstract*— This paper explores optimizing road networks using graph theory to identify the most efficient connections between nodes, minimizing travel time across the area. By modeling geographical and travel distances, we determine the most strategic new connections that reflect real-world conditions. The findings aim to prioritize road construction, improving connectivity and optimizing resource allocation.**

***Keywords*—Graph Theory, Road Network Optimization, Travel Time Reduction, Map.**

I. Introduction

The optimization of road networks plays a crucial role in improving transportation and connectivity, directly impacting travel time, economic growth, and regional development. In large or expanding areas, ensuring that all locations are efficiently connected can be challenging. Classical algorithms like Kruskal's and Prim's have provided foundational tools for solving this problem by helping design networks that connect all points (or nodes) with the shortest possible total distance. These algorithms focus on forming minimum spanning trees – essentially selecting the shortest paths that link every node without creating unnecessary loops or redundancies.

However, the reality of road networks often goes beyond just connecting points for the first time. In many cases, infrastructure already exists, but some areas suffer from bottlenecks, long detours, or sparse connections that slow down travel and limit access. This paper builds on the principles of Kruskal and Prim by shifting focus to a different question: Where should we add new roads to make travel faster and more efficient for everyone? Instead of just minimizing the initial cost of connection, we look at how to improve the existing network by strategically adding new links that cut down overall travel time.

By modeling the geography and current travel patterns, we identify potential locations for new connections that would bring the greatest improvements to the entire network. Our goal is to provide insights that can help prioritize road construction projects, ensuring resources are allocated effectively to improve accessibility and connectivity for the greatest number of people. This approach not only enhances transportation efficiency but also supports economic growth by making it easier to move goods, services, and people across regions.

II. Theoretical Background

A. Latitude and Longitude

Latitude and longitude are angular measurements that describe a point’s position on the Earth's surface.

a. Latitude measures the distance north or south of the equator, ranging from 0° at the equator to 90° at the poles.
b. Longitude measures the distance east or west of the Prime Meridian, ranging from 0° to 180°.

These coordinates are expressed in degrees, and sometimes in minutes and seconds (DMS format) or decimal degrees (DD format).

B. Distance Calculation Methods

Calculating the distance between two points given their latitude and longitude can be approached in several ways, depending on the required level of accuracy and the assumptions made about the Earth's shape.

The Haversine formula provides a reliable method to calculate the shortest distance between two points along the surface of a sphere. This great-circle distance calculation is widely used due to its balance of simplicity and accuracy.

The Haversine formula is expressed as:

$$a = sin^2\left(\frac{\Delta\varphi}{2}\right) + cos(\varphi_1)\cdot cos(\varphi_2)\cdot sin^2\left(\frac{\Delta\lambda}{2}\right)$$
$$c = 2 \cdot atan2(\sqrt{a}, \sqrt{(1-a)}) \qquad (1)$$
$$d = R \cdot c$$

Where:

a. φ_1 – Latitudes of point 1 (in radians)
b. φ_2 – Latitudes of point 2 (in radians)
c. $\Delta\varphi$ – Difference in latitude ($\varphi_2 - \varphi_1$)
d. $\Delta\lambda$ – Difference in longitude
e. R – Radius of the Earth (typically 6,371 km)
f. d – Distance between the two points

C. Graph

A graph G is a mathematical structure used to represent relationships between objects. It is defined as $G = (V, E)$ where V is the set of vertices (or nodes) and E is the set of edges that connect pairs of vertices [1]. The set of vertices V must be non-empty, indicating that a graph must contain at least one vertex. However, the set of edges E can be empty, meaning that a graph can exist without any connections between its vertices. This flexibility allows graphs to represent a wide range of structures, from isolated points to complex networks with numerous connections.

Graph connectivity can be represented using various methods, such as adjacency matrices, incidence matrices, and adjacency lists [2]. In this paper, the adjacency matrix is chosen because it allows for the storage of additional values at each node, facilitating easier access to adjacent nodes when retrieving geographical information.

D. Directed Weighed Graphs

A directed weighted graph models road networks where edges have both direction and cost, reflecting one-way streets or highways with varying travel distances, times, or tolls. Each edge has a starting point (tail), an endpoint (head), and a weight, representing the effort required to traverse it. Unlike undirected graphs, this structure allows for more accurate modeling of real-world scenarios, where congestion may occur on one path of the road but not the other. This enables more precise optimization of transportation networks.

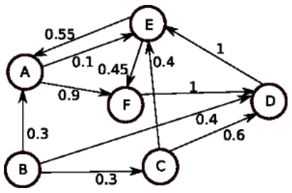


Figure 1. Directed Weighed Graph
Source: https://www.researchgate.net/profile/Flavio-Coelho-5/publication/301842622/figure/fig7/AS:668875754573847@1536483815756/A-weighted-directed-graph.ppm

III. Implementation

The implementation begins by designing a directed, weighted graph to represent the road network, where nodes correspond to locations and edges represent the connecting roads. Each edge is assigned a weight reflecting relevant costs, such as travel time. Once the graph structure is established, the next step involves calculating road travel distances using Dijkstra’s algorithm and determining the geographical distances between nodes with the Haversine formula, which accounts for the Earth's curvature.

Finally, we evaluate the ratios between road travel distances and geographical distances across various paths to assess network efficiency and identify potential areas for optimization. It is important to note that the shortest path calculation does not account for negative edge weights (costs) within the graph.

A. Observation

The goal of the implementation is to identify the most effective road to construct, focusing on locations with the highest disparity between geographical distance and travel time.

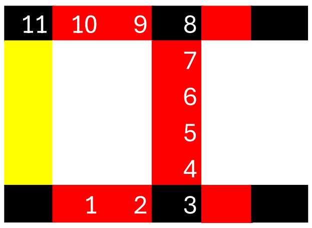


Figure 2. Simple Observation using Grid
Source: Personal Illustration

For example, in **Figure 2** (where black represents nodes, red indicates existing roads, and yellow marks the target road), the geographic distance between two locations is 5 units, while the travel distance by road is 11 units. This 2.2 times longer distance suggests a significant increase in travel time, making the yellow (target road) the optimal choice for construction, as it offers the highest ratio compared to other potential roads.

B. Designing Directed Weighted Graph

First, represent locations as nodes (or vertices) and the roads connecting them as edges.

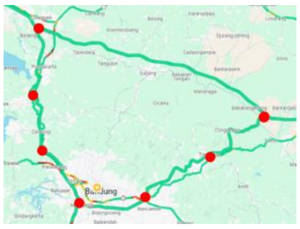


Figure 3. Map
Source: https://www.google.com/maps/@-6.6685285,107.7985122,10.49z/data=!5m1!1e1?entry=ttu&g_ep=EgoyMDI0MTIxMS4wIKXMDSoASAFQAw%3D%3D

Next, assign each edge a weight, such as the time required to travel between the nodes (for example inserting 10). Additionally, store the latitude and longitude coordinates for each node for the geographical calculation purpose later.

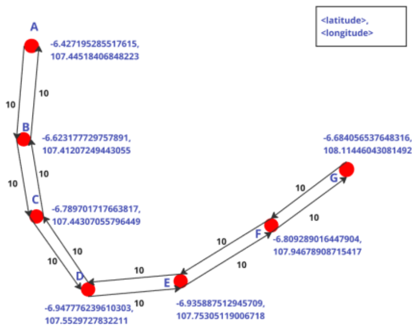
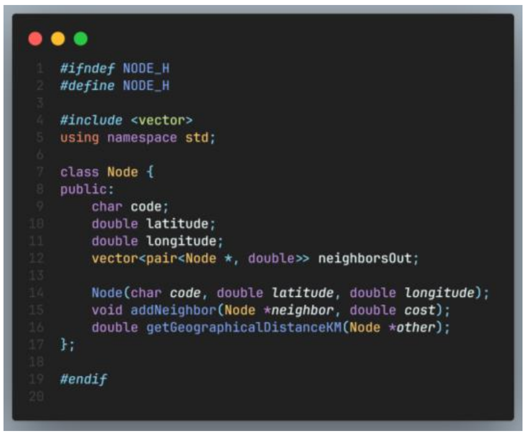


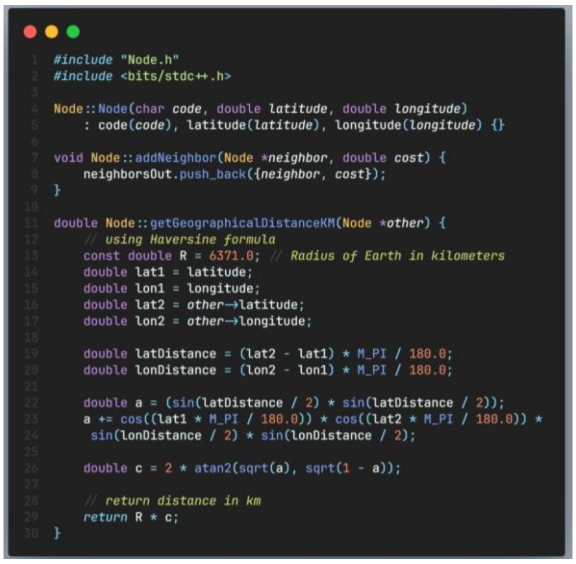
Figure 4. Directed Weighed Graph
Source: Personal Illustration

The representation of the graph or node is structured as adjacency list [2] as follows:



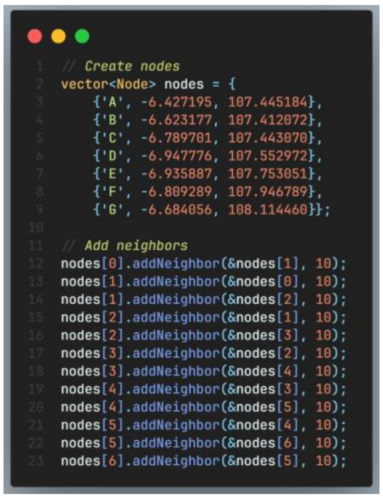
```cpp
#ifndef NODE_H
#define NODE_H

#include <vector>
using namespace std;

class Node {
public:
    char code;
    double latitude;
    double longitude;
    vector<pair<Node *, double>> neighborsOut;

    Node(char code, double latitude, double longitude);
    void addNeighbor(Node *neighbor, double cost);
    double getGeographicalDistanceKM(Node *other);
};

#endif
```

Figure 5. Node Header File
Source: https://github.com/yonatan-nyo/road-optimization



```cpp
#include "Node.h"
#include <bits/stdc++.h>

Node::Node(char code, double latitude, double longitude)
    : code(code), latitude(latitude), longitude(longitude) {}

void Node::addNeighbor(Node *neighbor, double cost) {
    neighborsOut.push_back({neighbor, cost});
}

double Node::getGeographicalDistanceKM(Node *other) {
    // using Haversine formula
    const double R = 6371.0; // Radius of Earth in kilometers
    double lat1 = latitude;
    double lon1 = longitude;
    double lat2 = other->latitude;
    double lon2 = other->longitude;

    double latDistance = (lat2 - lat1) * M_PI / 180.0;
    double lonDistance = (lon2 - lon1) * M_PI / 180.0;

    double a = (sin(latDistance / 2) * sin(latDistance / 2));
    a += cos((lat1 * M_PI / 180.0)) * cos((lat2 * M_PI / 180.0)) *
     sin(lonDistance / 2) * sin(lonDistance / 2);

    double c = 2 * atan2(sqrt(a), sqrt(1 - a));

    // return distance in km
    return R * c;
}
```

Figure 6. Node C++ Implementation
Source: https://github.com/yonatan-nyo/road-optimization

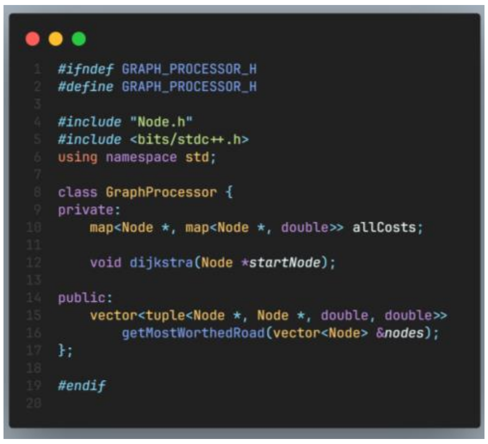


```cpp
// Create nodes
vector<Node> nodes = {
    {'A', -6.427195, 107.445184},
    {'B', -6.623177, 107.412072},
    {'C', -6.789701, 107.443070},
    {'D', -6.947776, 107.552972},
    {'E', -6.935887, 107.753051},
    {'F', -6.809289, 107.946789},
    {'G', -6.684056, 108.114460}};

// Add neighbors
nodes[0].addNeighbor(&nodes[1], 10);
nodes[1].addNeighbor(&nodes[0], 10);
nodes[1].addNeighbor(&nodes[2], 10);
nodes[2].addNeighbor(&nodes[1], 10);
nodes[2].addNeighbor(&nodes[3], 10);
nodes[3].addNeighbor(&nodes[2], 10);
nodes[3].addNeighbor(&nodes[4], 10);
nodes[4].addNeighbor(&nodes[3], 10);
nodes[4].addNeighbor(&nodes[5], 10);
nodes[5].addNeighbor(&nodes[4], 10);
nodes[5].addNeighbor(&nodes[6], 10);
nodes[6].addNeighbor(&nodes[5], 10);
```

Figure 7. Node C++ Usage
Source: https://github.com/yonatan-nyo/road-optimization

C. Graph Processing

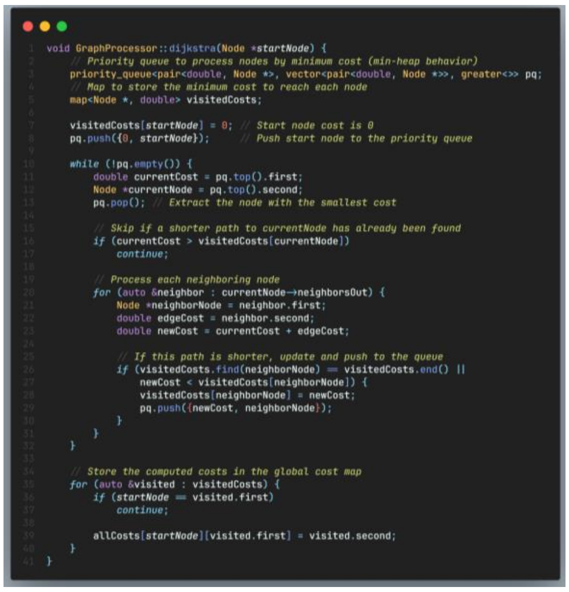
The node representation is then processed to calculate the shortest path between nodes using Dijkstra's algorithm. Next, the roads are evaluated based on the ratio between their cost and geographical distance to identify the most optimal routes. The implementation is as follows:



```cpp
#ifndef GRAPH_PROCESSOR_H
#define GRAPH_PROCESSOR_H

#include "Node.h"
#include <bits/stdc++.h>
using namespace std;

class GraphProcessor {
private:
    map<Node *, map<Node *, double>> allCosts;

    void dijkstra(Node *startNode);

public:
    vector<tuple<Node *, Node *, double, double>>
        getMostWorthedRoad(vector<Node> &nodes);
};

#endif
```

Figure 8. Graph Processor Header File
Source: https://github.com/yonatan-nyo/road-optimization

We apply Dijkstra's algorithm, assuming non-negative costs, to ensure the precise identification of the most efficient path between nodes. By systematically evaluating all possible routes and accounting for their costs, the algorithm guarantees that the shortest path, in terms of total cost, is accurately determined [3].

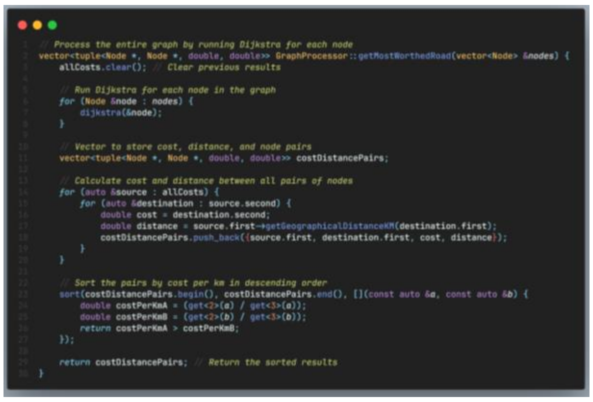


```cpp
void GraphProcessor::dijkstra(Node *startNode) {
    // Priority queue to process nodes by minimum cost (min-heap behavior)
    priority_queue<pair<double, Node *>, vector<pair<double, Node *>>, greater<>> pq;
    // Map to store the minimum cost to reach each node
    map<Node *, double> visitedCosts;

    visitedCosts[startNode] = 0; // Start node cost is 0
    pq.push({0, startNode});     // Push start node to the priority queue

    while (!pq.empty()) {
        double currentCost = pq.top().first;
        Node *currentNode = pq.top().second;
        pq.pop(); // Extract the node with the smallest cost

        // Skip if a shorter path to currentNode has already been found
        if (currentCost > visitedCosts[currentNode])
            continue;

        // Process each neighboring node
        for (auto &neighbor : currentNode->neighborsOut) {
            Node *neighborNode = neighbor.first;
            double edgeCost = neighbor.second;
            double newCost = currentCost + edgeCost;

            // If this path is shorter, update and push to the queue
            if (visitedCosts.find(neighborNode) == visitedCosts.end() ||
                newCost < visitedCosts[neighborNode]) {
                visitedCosts[neighborNode] = newCost;
                pq.push({newCost, neighborNode});
            }
        }
    }

    // Store the computed costs in the global cost map
    for (auto &visited : visitedCosts) {
        if (startNode == visited.first)
            continue;

        allCosts[startNode][visited.first] = visited.second;
    }
}
```

Figure 9. Dijkstra
Source: https://github.com/yonatan-nyo/road-optimization

After calculating the shortest path between nodes, we can determine the cost per kilometer by comparing the total cost with the geographical distance.

$$costPerKilometer = \frac{cost}{distance} \tag{2}$$



```cpp
// Process the entire graph by running Dijkstra for each node
vector<tuple<Node *, Node *, double, double>> GraphProcessor::getMostWorthedRoad(vector<Node> &nodes) {
    allCosts.clear(); // Clear previous results

    // Run Dijkstra for each node in the graph
    for (Node &node : nodes) {
        dijkstra(&node);
    }

    // Vector to store cost, distance, and node pairs
    vector<tuple<Node *, Node *, double, double>> costDistancePairs;

    // Calculate cost and distance between all pairs of nodes
    for (auto &source : allCosts) {
        for (auto &destination : source.second) {
            double cost = destination.second;
            double distance = source.first->getGeographicalDistanceKM(destination.first);
            costDistancePairs.push_back({source.first, destination.first, cost, distance});
        }
    }

    // Sort the pairs by cost per km in descending order
    sort(costDistancePairs.begin(), costDistancePairs.end(), [](const auto &a, const auto &b) {
        double costPerKmA = (get<2>(a) / get<3>(a));
        double costPerKmB = (get<2>(b) / get<3>(b));
        return costPerKmA > costPerKmB;
    });

    return costDistancePairs; // Return the sorted results
}
```

Figure 10. Calculation for Most Worthed Road
Source: https://github.com/yonatan-nyo/road-optimization

By sorting the nodes in descending order of cost per kilometer, we can identify the most inefficient unconnected nodes. In other words, this reveals the roads that must be constructed to connect these two cities or areas.

IV. USAGE

The use of this algorithm can be applied in various contexts, as the cost or weight (non-negative) of the edges can be influenced by multiple factors, allowing for more accurate results through further investigation. However, the main challenge addressed by this algorithm is preventing the creation of redundant roads; it ensures that we only need to connect the nodes without constructing unnecessary roads between them.

A. Half-Circular Road Problem

When a road follows a half-circle route to reach its destination, the most efficient approach is to connect both the start and end points of the route directly. This minimizes unnecessary travel and reduces costs.

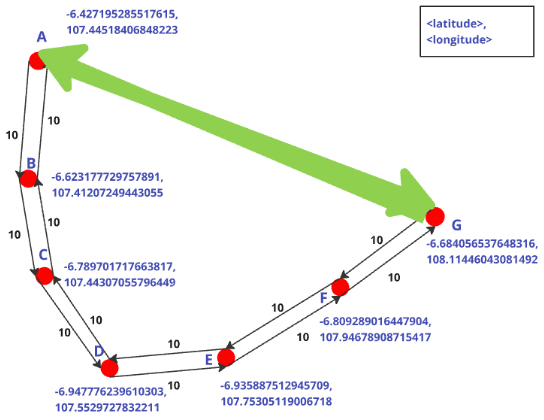
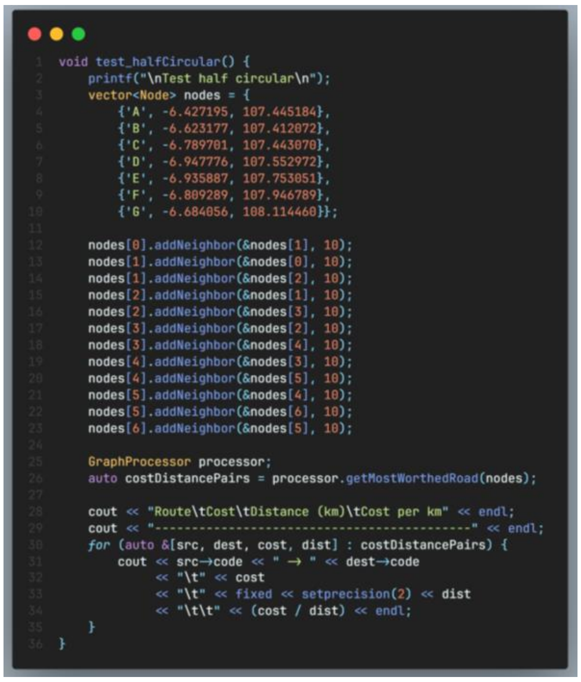
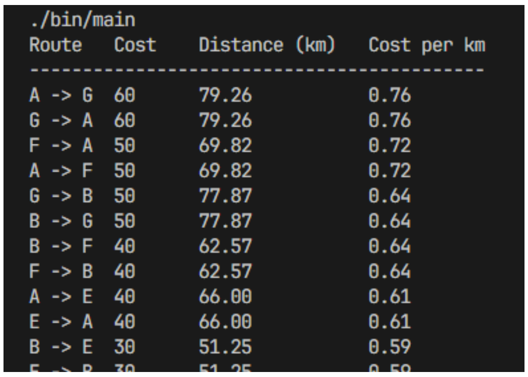


Figure 11. Expectation Result (green) of Half-Circular Road Problem
Source: Personal Illustration



```cpp
void test_halfCircular() {
    printf("\nTest half circular\n");
    vector<Node> nodes = {
        {'A', -6.427195, 107.445184},
        {'B', -6.623177, 107.412072},
        {'C', -6.789701, 107.443070},
        {'D', -6.947776, 107.552972},
        {'E', -6.935887, 107.753051},
        {'F', -6.809289, 107.946789},
        {'G', -6.684056, 108.114460}};

    nodes[0].addNeighbor(&nodes[1], 10);
    nodes[1].addNeighbor(&nodes[0], 10);
    nodes[1].addNeighbor(&nodes[2], 10);
    nodes[2].addNeighbor(&nodes[1], 10);
    nodes[2].addNeighbor(&nodes[3], 10);
    nodes[3].addNeighbor(&nodes[2], 10);
    nodes[3].addNeighbor(&nodes[4], 10);
    nodes[4].addNeighbor(&nodes[3], 10);
    nodes[4].addNeighbor(&nodes[5], 10);
    nodes[5].addNeighbor(&nodes[4], 10);
    nodes[5].addNeighbor(&nodes[6], 10);
    nodes[6].addNeighbor(&nodes[5], 10);

    GraphProcessor processor;
    auto costDistancePairs = processor.getMostWorthedRoad(nodes);

    cout << "Route\tCost\tDistance (km)\tCost per km" << endl;
    cout << "----------------------------------------" << endl;
    for (auto &[src, dest, cost, dist] : costDistancePairs) {
        cout << src->code << " -> " << dest->code
            << "\t" << cost
            << "\t" << fixed << setprecision(2) << dist
            << "\t\t" << (cost / dist) << endl;
    }
}
```

Figure 12. Implementation of Half-Circular Road Problem
Source: https://github.com/yonatan-nyo/road-optimization



```
./bin/main
Route   Cost    Distance (km)   Cost per km
--------------------------------------------
A -> G  60      79.26           0.76
G -> A  60      79.26           0.76
F -> A  50      69.82           0.72
A -> F  50      69.82           0.72
G -> B  50      77.87           0.64
B -> G  50      77.87           0.64
B -> F  40      62.57           0.64
F -> B  40      62.57           0.64
A -> E  40      66.00           0.61
E -> A  40      66.00           0.61
B -> E  30      51.25           0.59
E -> B  30      51.25           0.59
```

Figure 13. Result of Half-Circular Road Problem
Source: Personal Documentation

The cost per kilometer between A and G, as well as G and A, is higher than that of other routes, highlighting the need to construct roads that will effectively connect these two cities or areas

B. Half-Circular Road with an Already Efficient Path Problem

In some cases, a half-circular road may exist between two points, but a more efficient and shorter path is already accessible through other connected nodes. Building a direct road in such situations can create unnecessary redundancy without significantly improving travel efficiency. Therefore, it is more practical to focus on connecting the existing circular path rather than constructing a new one.

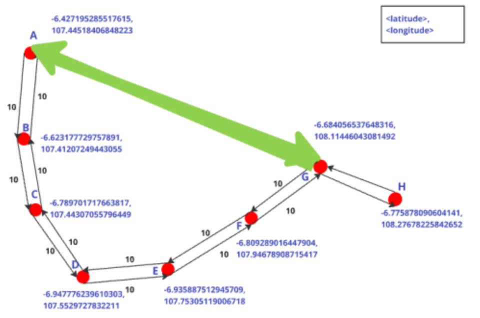
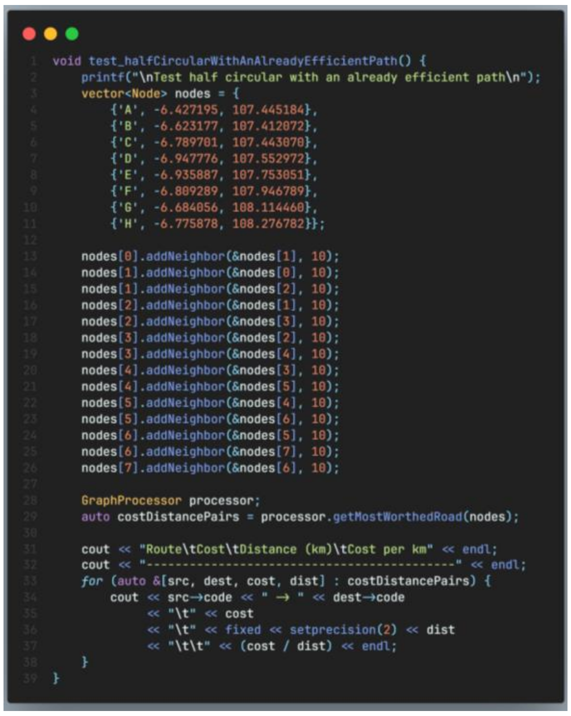
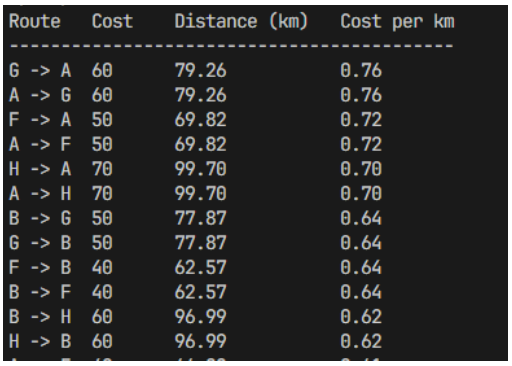


Figure 14. Expectation Result (green) of Half-Circular Road with an Already Efficient Path Problem
Source: Personal Illustration



```
void test_halfCircularWithAnAlreadyEfficientPath() {
    printf("\nTest half circular with an already efficient path\n");
    vector<Node> nodes = {
        {'A', -6.427195, 107.445184},
        {'B', -6.623177, 107.412072},
        {'C', -6.789701, 107.443070},
        {'D', -6.947776, 107.552972},
        {'E', -6.935887, 107.753051},
        {'F', -6.809289, 107.946789},
        {'G', -6.684056, 108.114460},
        {'H', -6.775878, 108.276782}};

    nodes[0].addNeighbor(&nodes[1], 10);
    nodes[1].addNeighbor(&nodes[0], 10);
    nodes[1].addNeighbor(&nodes[2], 10);
    nodes[2].addNeighbor(&nodes[1], 10);
    nodes[2].addNeighbor(&nodes[3], 10);
    nodes[3].addNeighbor(&nodes[2], 10);
    nodes[3].addNeighbor(&nodes[4], 10);
    nodes[4].addNeighbor(&nodes[3], 10);
    nodes[4].addNeighbor(&nodes[5], 10);
    nodes[5].addNeighbor(&nodes[4], 10);
    nodes[5].addNeighbor(&nodes[6], 10);
    nodes[6].addNeighbor(&nodes[5], 10);
    nodes[6].addNeighbor(&nodes[7], 10);
    nodes[7].addNeighbor(&nodes[6], 10);

    GraphProcessor processor;
    auto costDistancePairs = processor.getMostWorthedRoad(nodes);

    cout << "Route\tCost\tDistance (km)\tCost per km" << endl;
    cout << "----------------------------------------------" << endl;
    for (auto &[src, dest, cost, dist] : costDistancePairs) {
        cout << src->code << " -> " << dest->code
             << "\t" << cost
             << "\t" << fixed << setprecision(2) << dist
             << "\t\t" << (cost / dist) << endl;
    }
}
```

Figure 15. Implementation of Half-Circular Road with an Already Efficient Path Problem
Source: https://github.com/yonatan-nyo/road-optimization



```
Route   Cost    Distance (km)   Cost per km
----------------------------------------------
G -> A  60      79.26           0.76
A -> G  60      79.26           0.76
F -> A  50      69.82           0.72
A -> F  50      69.82           0.72
H -> A  70      99.70           0.70
A -> H  70      99.70           0.70
B -> G  50      77.87           0.64
G -> B  50      77.87           0.64
F -> B  40      62.57           0.64
B -> F  40      62.57           0.64
B -> H  60      96.99           0.62
H -> B  60      96.99           0.62
```

Figure 16. Result of Half-Circular Road with an Already Efficient Path Problem
Source: Personal Documentation

The cost per kilometer between A and G, as well as G and A, is higher than that of other routes. However, it is important to note that while the cost per kilometer from H to A is also high, it remains lower than the costs from GA (or AG) and FA (or AF), because the path between G and H is already efficient, without unnecessary detours like the route between A and G.

C. High-Cost Road Problem

Note that the weight of the edges can be adjusted. In the case of high-cost roads, we will simulate a scenario where the worse the traffic congestion (time consumed), the higher the cost.

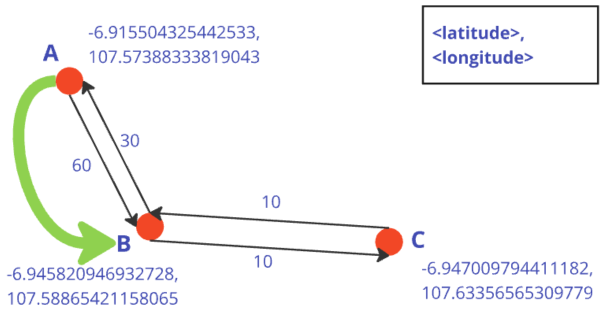
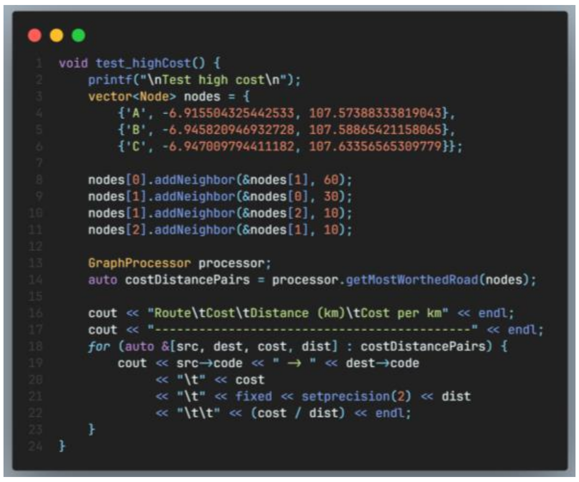
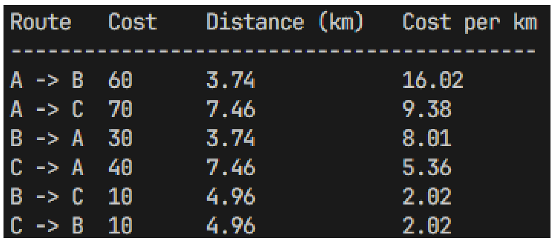


Figure 17. Expectation Result (green) of High-Cost Road Problem
Source: Personal Illustration



```
void test_highCost() {
    printf("\nTest high cost\n");
    vector<Node> nodes = {
        {'A', -6.915504325442533, 107.57388333819043},
        {'B', -6.945820946932728, 107.58865421158065},
        {'C', -6.947009794411182, 107.63356565309779}};

    nodes[0].addNeighbor(&nodes[1], 60);
    nodes[1].addNeighbor(&nodes[0], 30);
    nodes[1].addNeighbor(&nodes[2], 10);
    nodes[2].addNeighbor(&nodes[1], 10);

    GraphProcessor processor;
    auto costDistancePairs = processor.getMostWorthedRoad(nodes);

    cout << "Route\tCost\tDistance (km)\tCost per km" << endl;
    cout << "----------------------------------------------" << endl;
    for (auto &[src, dest, cost, dist] : costDistancePairs) {
        cout << src->code << " -> " << dest->code
             << "\t" << cost
             << "\t" << fixed << setprecision(2) << dist
             << "\t\t" << (cost / dist) << endl;
    }
}
```

Figure 18. Implementation of High-Cost Road Problem
Source: https://github.com/yonatan-nyo/road-optimization



```
Route   Cost    Distance (km)   Cost per km
----------------------------------------------
A -> B  60      3.74            16.02
A -> C  70      7.46            9.38
B -> A  30      3.74            8.01
C -> A  40      7.46            5.36
B -> C  10      4.96            2.02
C -> B  10      4.96            2.02
```

Figure 19. Result of High-Cost Road Problem
Source: Personal Documentation

The cost per kilometer between A and B is the highest, as the cost is significantly inflated over a short distance due to heavy traffic congestion. Therefore, constructing a new road from A to B would be the best solution.

D. Circling Road Problem

In a road network designed as a two-way circular path connecting neighboring nodes, where the cost represents road distance and is uniform across all connections, the most inefficient connections occur between pairs of nodes that are directly opposite each other on the circle.

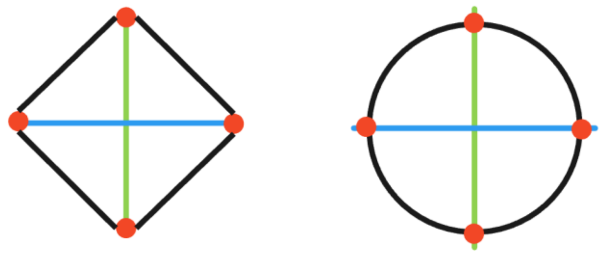


Figure 20. Expectation Result (green or blue) of Circling Road Problem
Source: Personal Illustration

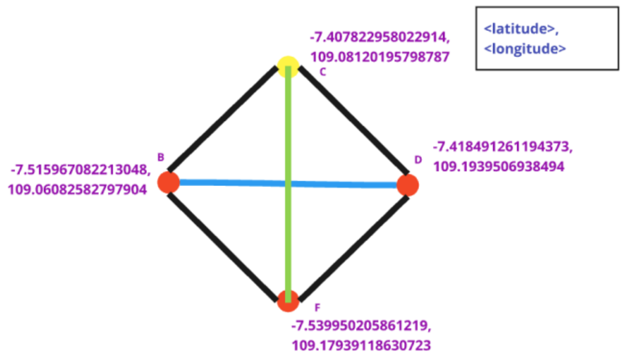
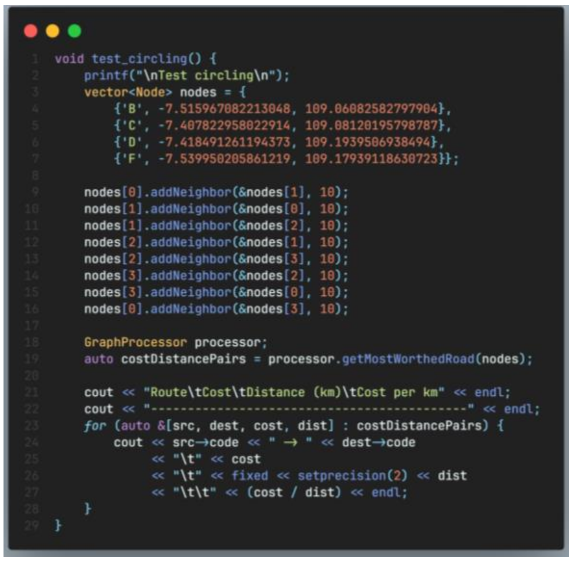
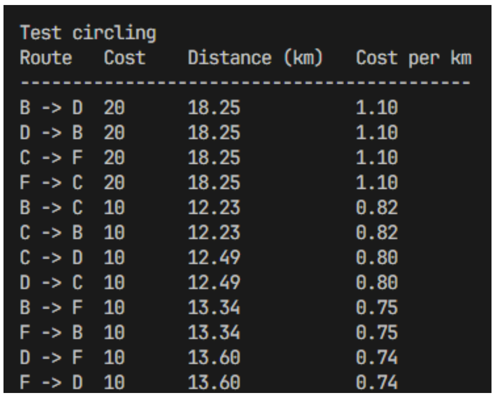


Figure 21. Expectation Result (green or blue) of Circling Road Problem (with coordinates)
Source: Personal Illustration



```
void test_circling() {
    printf("\nTest circling\n");
    vector<Node> nodes = {
        {'B', -7.515967082213048, 109.06082582797904},
        {'C', -7.407822958022914, 109.08120195798787},
        {'D', -7.418491261194373, 109.1939506938494},
        {'F', -7.539950205861219, 109.17939118630723}};

    nodes[0].addNeighbor(&nodes[1], 10);
    nodes[1].addNeighbor(&nodes[0], 10);
    nodes[1].addNeighbor(&nodes[2], 10);
    nodes[2].addNeighbor(&nodes[1], 10);
    nodes[2].addNeighbor(&nodes[3], 10);
    nodes[3].addNeighbor(&nodes[2], 10);
    nodes[3].addNeighbor(&nodes[0], 10);
    nodes[0].addNeighbor(&nodes[3], 10);

    GraphProcessor processor;
    auto costDistancePairs = processor.getMostWorthedRoad(nodes);

    cout << "Route\tCost\tDistance (km)\tCost per km" << endl;
    cout << "-----------------------------------------" << endl;
    for (auto &[src, dest, cost, dist] : costDistancePairs) {
        cout << src->code << " -> " << dest->code
             << "\t" << cost
             << "\t" << fixed << setprecision(2) << dist
             << "\t\t" << (cost / dist) << endl;
    }
}
```

Figure 22. Implementation of Circling Road Problem
Source: https://github.com/yonatan-nyo/road-optimization



```
Test circling
Route   Cost    Distance (km)   Cost per km
------------------------------------------
B -> D  20      18.25           1.10
D -> B  20      18.25           1.10
C -> F  20      18.25           1.10
F -> C  20      18.25           1.10
B -> C  10      12.23           0.82
C -> B  10      12.23           0.82
C -> D  10      12.49           0.80
D -> C  10      12.49           0.80
B -> F  10      13.34           0.75
F -> B  10      13.34           0.75
D -> F  10      13.60           0.74
F -> D  10      13.60           0.74
```

Figure 23. Result of Circling Road Problem
Source: Personal Documentation

E. Circling Road Problem with External Nodes

In a road network designed as a circular two-way path connecting neighboring nodes, with additional nodes branching outward, the travel cost is assumed to be proportional to the road distance and remains consistent across all connections. The most inefficient connections occur between pairs of nodes that are the furthest apart along the circle. It is important to note that external nodes do not influence the most inefficient connections. So, we need to investigate further to identify which connections have the greatest impact on improving efficiency.

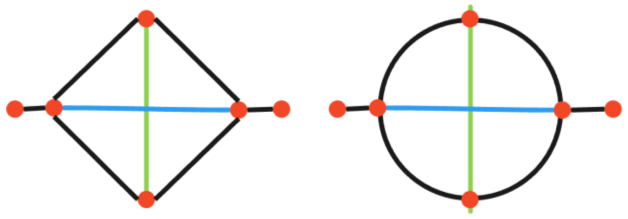


Figure 23. Expectation Result (green or blue) of Circling Road Problem with External Nodes
Source: Personal Illustration

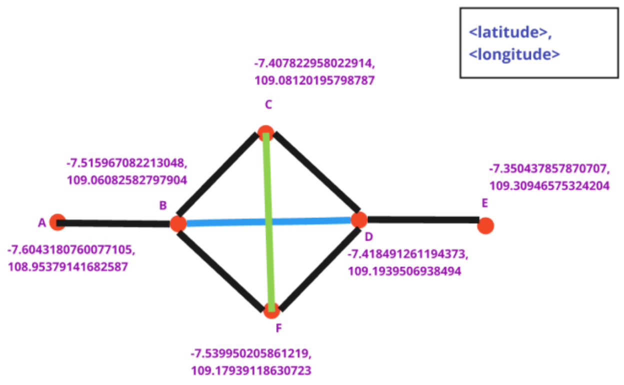
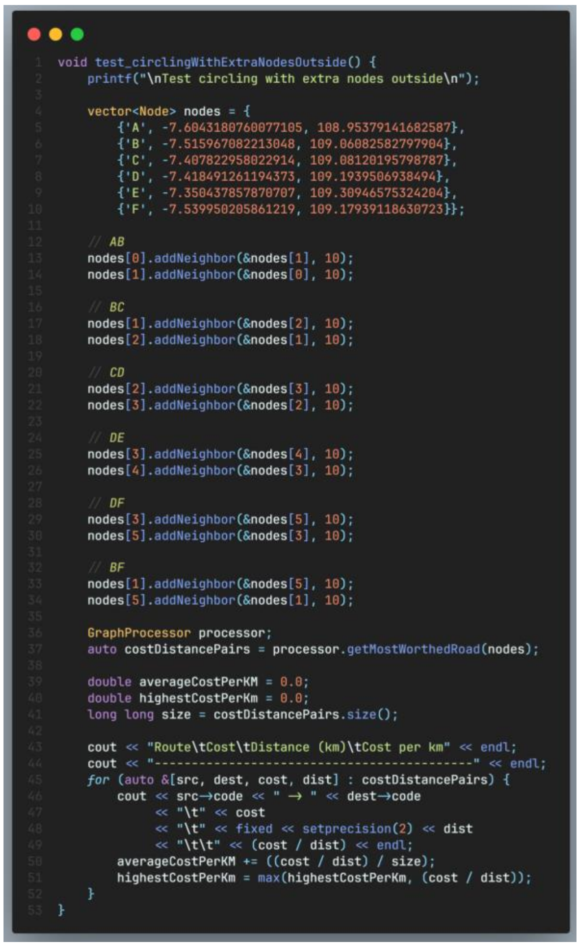
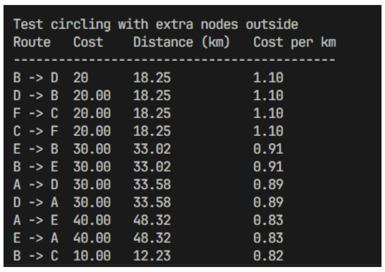


Figure 24. Expectation Result (green or blue) of Circling Road Problem with External Nodes (with coordinates)
Source: Personal Illustration



```cpp
void test_circlingWithExtraNodesOutside() {
    printf("\nTest circling with extra nodes outside\n");

    vector<Node> nodes = {
        {'A', -7.6043180760077105, 108.95379141682587},
        {'B', -7.515967082213048, 109.06082582797904},
        {'C', -7.407822958022914, 109.08120195798787},
        {'D', -7.418491261194373, 109.1939506938494},
        {'E', -7.350437857870707, 109.30946575324204},
        {'F', -7.539950205861219, 109.17939118630723}};

    // AB
    nodes[0].addNeighbor(&nodes[1], 10);
    nodes[1].addNeighbor(&nodes[0], 10);

    // BC
    nodes[1].addNeighbor(&nodes[2], 10);
    nodes[2].addNeighbor(&nodes[1], 10);

    // CD
    nodes[2].addNeighbor(&nodes[3], 10);
    nodes[3].addNeighbor(&nodes[2], 10);

    // DE
    nodes[3].addNeighbor(&nodes[4], 10);
    nodes[4].addNeighbor(&nodes[3], 10);

    // DF
    nodes[3].addNeighbor(&nodes[5], 10);
    nodes[5].addNeighbor(&nodes[3], 10);

    // BF
    nodes[1].addNeighbor(&nodes[5], 10);
    nodes[5].addNeighbor(&nodes[1], 10);

    GraphProcessor processor;
    auto costDistancePairs = processor.getMostWorthedRoad(nodes);

    double averageCostPerKM = 0.0;
    double highestCostPerKm = 0.0;
    long long size = costDistancePairs.size();

    cout << "Route\tCost\tDistance (km)\tCost per km" << endl;
    cout << "-------------------------------------------------" << endl;
    for (auto &[src, dest, cost, dist] : costDistancePairs) {
        cout << src->code << " -> " << dest->code
             << "\t" << cost
             << "\t" << fixed << setprecision(2) << dist
             << "\t\t" << (cost / dist) << endl;
        averageCostPerKM += ((cost / dist) / size);
        highestCostPerKm = max(highestCostPerKm, (cost / dist));
    }
}
```

Figure 25. Implementation of Circling Road Problem with External Nodes
Source: https://github.com/yonatan-nyo/road-optimization

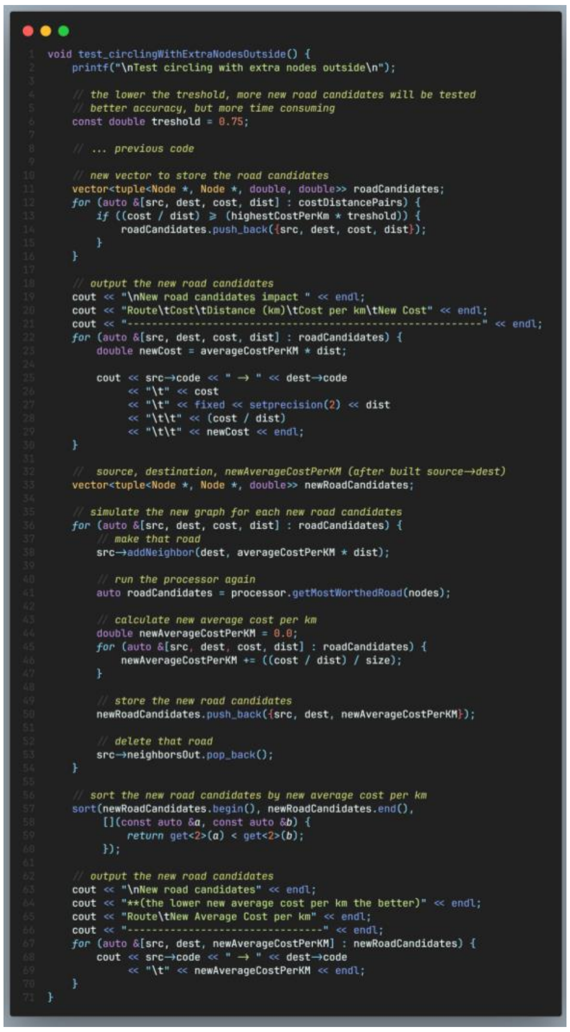


```
Test circling with extra nodes outside
Route   Cost    Distance (km)   Cost per km
-------------------------------------------
B -> D  20      18.25           1.10
D -> B  20.00   18.25           1.10
F -> C  20.00   18.25           1.10
C -> F  20.00   18.25           1.10
E -> B  30.00   33.02           0.91
B -> E  30.00   33.02           0.91
A -> D  30.00   33.58           0.89
D -> A  30.00   33.58           0.89
A -> E  40.00   48.32           0.83
E -> A  40.00   48.32           0.83
B -> C  10.00   12.23           0.82
```

Figure 26. Result of Circling Road Problem with External Nodes
Source: Personal Documentation

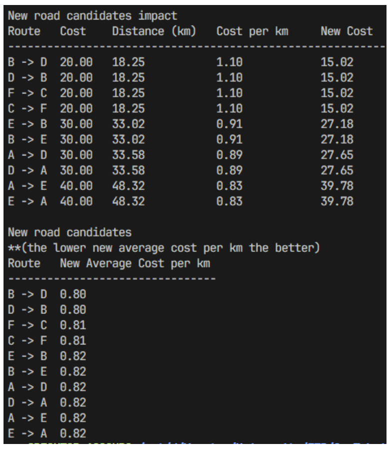
The results indicate that the least efficient connections occur between pairs of nodes that are the furthest apart along the circle. However, it is understood that some connections are more efficient than others. Therefore, it is necessary to assess which connections have a greater impact on reducing the average cost per kilometer.

By setting a threshold to identify potential candidate connections, we can calculate the average cost per kilometer after adding each connection. The connection that results in the lowest new average cost per kilometer is the most optimal.



```cpp
void test_circlingWithExtraNodesOutside() {
    printf("\nTest circling with extra nodes outside\n");

    // the lower the treshold, more new road candidates will be tested
    // better accuracy, but more time consuming
    const double treshold = 0.75;

    // ... previous code

    // new vector to store the road candidates
    vector<tuple<Node *, Node *, double, double>> roadCandidates;
    for (auto &[src, dest, cost, dist] : costDistancePairs) {
        if ((cost / dist) >= (highestCostPerKm * treshold)) {
            roadCandidates.push_back({src, dest, cost, dist});
        }
    }

    // output the new road candidates
    cout << "\nNew road candidates impact " << endl;
    cout << "Route\tCost\tDistance (km)\tCost per km\tNew Cost" << endl;
    cout << "------------------------------------------------------------" << endl;
    for (auto &[src, dest, cost, dist] : roadCandidates) {
        double newCost = averageCostPerKM * dist;

        cout << src->code << " -> " << dest->code
             << "\t" << cost
             << "\t" << fixed << setprecision(2) << dist
             << "\t\t" << (cost / dist)
             << "\t\t" << newCost << endl;
    }

    //  source, destination, newAverageCostPerKM (after built source->dest)
    vector<tuple<Node *, Node *, double>> newRoadCandidates;

    // simulate the new graph for each new road candidates
    for (auto &[src, dest, cost, dist] : roadCandidates) {
        // make that road
        src->addNeighbor(dest, averageCostPerKM * dist);

        // run the processor again
        auto roadCandidates = processor.getMostWorthedRoad(nodes);

        // calculate new average cost per km
        double newAverageCostPerKM = 0.0;
        for (auto &[src, dest, cost, dist] : roadCandidates) {
            newAverageCostPerKM += ((cost / dist) / size);
        }

        // store the new road candidates
        newRoadCandidates.push_back({src, dest, newAverageCostPerKM});

        // delete that road
        src->neighborsOut.pop_back();
    }

    // sort the new road candidates by new average cost per km
    sort(newRoadCandidates.begin(), newRoadCandidates.end(),
         [](const auto &a, const auto &b) {
             return get<2>(a) < get<2>(b);
         });

    // output the new road candidates
    cout << "\nNew road candidates" << endl;
    cout << "**(the lower new average cost per km the better)" << endl;
    cout << "Route\tNew Average Cost per km" << endl;
    cout << "-------------------------------" << endl;
    for (auto &[src, dest, newAverageCostPerKM] : newRoadCandidates) {
        cout << src->code << " -> " << dest->code
             << "\t" << newAverageCostPerKM << endl;
    }
}
```

Figure 27. Implementation of Circling Road Problem with External Nodes with Candidates
Source: https://github.com/yonatan-nyo/road-optimization



```
New road candidates impact
Route   Cost    Distance (km)   Cost per km     New Cost
-----------------------------------------------------------
B -> D  20.00   18.25           1.10            15.02
D -> B  20.00   18.25           1.10            15.02
F -> C  20.00   18.25           1.10            15.02
C -> F  20.00   18.25           1.10            15.02
E -> B  30.00   33.02           0.91            27.18
B -> E  30.00   33.02           0.91            27.18
A -> D  30.00   33.58           0.89            27.65
D -> A  30.00   33.58           0.89            27.65
A -> E  40.00   48.32           0.83            39.78
E -> A  40.00   48.32           0.83            39.78

New road candidates
**(the lower new average cost per km the better)
Route   New Average Cost per km
-------------------------------
B -> D  0.80
D -> B  0.80
F -> C  0.81
C -> F  0.81
E -> B  0.82
B -> E  0.82
A -> D  0.82
D -> A  0.82
A -> E  0.82
E -> A  0.82
```

Figure 28. Result of Circling Road Problem with External Nodes with Candidates
Source: https://github.com/yonatan-nyo/road-optimization

Therefore, the connections between B and D are more significant than the others, as connecting B and D (either BD or DB) will help reduce the cost for AE and EA, as shown in **Figure 24**.

V. Conclusion

Improving road conditions in the real world depends on various factors such as distance, time-spent, lifespan, build cost, and population density. By quantifying these factors into non-negative costs and applying them to the edges of the graph, we can identify the most inefficient roads. This analysis helps us determine whether we need to enlarge existing roads, address specific issues causing inefficiency, or construct entirely new roads to enhance connectivity and traffic flow, as discussed in Section IV.

VI. Appendix

a. YouTube video explaining the concept: https://youtube.com/shorts/DyoCbo89MWs
b. GitHub repository for the project: https://github.com/yonatan-nyo/road-optimization

VII. Acknowledgment

The author would like to express sincere gratitude to God Almighty for the guidance and ease in writing this paper. Special thanks are also extended to Ir. Rila Mandala, M.Eng., Ph.D., for his role as the lecturer in the IF1220 Discrete Mathematics course. The author is deeply appreciative of the unwavering support from family and friends throughout the completion of this paper. Additionally, the author would like to thank Dr. Ir. Rinaldi Munir, M.T., for publishing the lecture materials on the website, which were instrumental in the research process. Finally, the author wishes to acknowledge Google Maps for its invaluable assistance in providing the maps used for testing in this paper.

References

[1] Munir, Rinaldi. 2023. "Graf (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf (Diakses pada 22 Desember 2024)
[2] Munir, Rinaldi. 2023. "Graf (Bagian 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf (Diakses pada 22 Desember 2024)
[3] Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs." https://www.cs.yale.edu/homes/lans/readings/routing/dijkstra-routing-1959.pdf (Diakses pada 27 Desember 2024)

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, 27 Desember 2024

[illegible]

Yonatan Edward Njoto
13523036