# Campus Gastronomy Tour: Investigating Hamiltonian Cycles on all ITB Ganesha Canteen

Syahrizal Bani Khairan - 13523063
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: syahrizal.khairan@gmail.com , 13523063@std.stei.itb.ac.id

***Abstract*—This paper aims to search for cyclic paths that vists every canteen, cafes, and the likes in the ITB Ganesha campus area using pathfinding algorithms, and examines their properties on graph modification. The graph constituting the canteens as vertices are connected by edges of variable weight that represents the walking distance between vertices. Of particular interest is finding the shortest path that visits all canteen in the campus area. This problem is analogous to the Travelling Salesman Problem. This paper will apply a brute force depth-first-search and Held-Karp algorithm to find such paths.**

***Keywords—travelling salesman problem;***

## I. Introduction

The Institut Teknologi Bandung (ITB) Ganesha campus contains many establishments offering food and beverages. These establishments, which from here on we refer as canteen, may provide drinks, light snacks, to a full meal. These collections of canteen are scattered throughout the area, providing easy access to the general populace and visitors. The distribution of these establishments around the campus presents an opportunity for analysis of their logistical optimization.

To analyze this problem from a computational perspective, the campus layout and its canteens can be formally modeled as a weighted undirected graph. In this model, each canteen is represented as a vertex (or node), and the walkable paths between them are represented as edges. The weight of each edge corresponds to the physical distance between two canteens, creating a network that accurately reflects the travel cost of moving between any two points. The challenge is, therefore, transforms a simple navigation task into a well-defined computational problem: finding the optimal tour of this graph.

This paper will investigate this specific instance of the TSP by first identifying the locations of all canteens on the Ganesha campus. Subsequently, the walking distances between each pair of canteens will be measured as the waling distance. Following the data gathering and graph construction, we will implement and compare two distinct algorithmic approaches to solve this problem: a brute-force depth-first search, which guarantees optimality by exploring every possible path, and the more sophisticated Held-Karp algorithm, a dynamic programming approach that is significantly more efficient for a larger number of vertices.

This paper will briefly explore on commonly used heuristics to solve similar problems. Additionally, we will examine a case of an addition of a node affects the optimal path.

## II. Methodology

### *A. Considered canteen and all their locations*

There are a lot of places where food and beverage can be acquired in the campus area. These ranges from machines to fully fledged cafés. This paper will restrict the sample to several concentrated points, of which the distance between them will be measured.

We exclude vending machines, automated coffee machines and other unmanned vendors from the sample. There a lot of such machines and these do not adequately represent the authentic ITB experience. A minimarket retail chain also opened their branch on campus. For similar reason, the author decides to exclude them as being not unique to the area.

Several points are actually a group of canteens within the same location. This is done to massively simplify calculations; places that are located within the near vicinity of each other are represented as a single node. A place that are roughly within 50 meters apart from another place in a group also belongs to the same group.

With these constraints, 12 points are considered for the context of this paper:

1. Kantin ATM Center
2. Gedung Kuliah Umum Barat (GKUB)
3. Kantin Labtek V
4. Gedung Kuliah Umum Timur (GKUT)
5. Eititu
6. Community Center Barat (CC Barat)
7. Tunas Padi
8. Kantin Lab Biru
9. Kantin CRCS
10. Kantin SBM
11. Kantin Timur
12. Kantin Barrack

Do note that this sample is biased with respect to the author’s knowledge and discretion. Should there be unmentioned places, it is to be understood that the author has no knowledge of their existence. Nevertheless, the provided list should capture the essence of the daily culinary experience for the students.

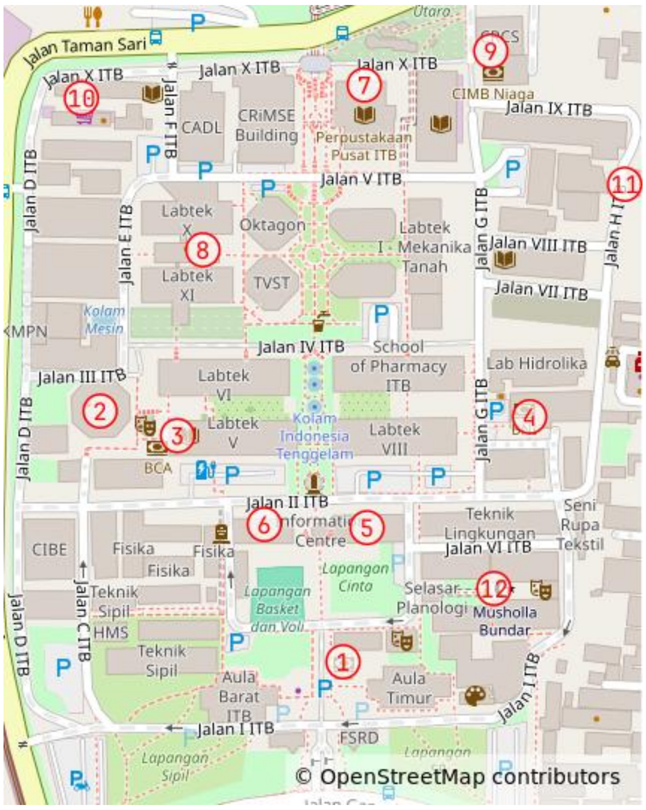


*Fig 1. Marked map showing each locations of canteens as numbered in the previous list.*

## *B. Gathering data on distances between canteens*

To model a tour that visits all canteens, there must be a path that connects each node to the others. Not all path are equal, since pairs of canteens may be further apart. For this reason, we will consider the graph of all canteens and their paths to be a weighted graph.

There are several ways to model the distance between canteens. One may plot the nodes into a map and simply measure the euclidean distances as they are located on the map. This approach does not reflect accurately the actual distance travelled for a person that travels on foot. One may refine the calculated path to follow through roads as seen on available maps. However, it is questionable how accurate this approach can model pedestrian paths. In this paper, the weight of a path is measured as the distance travelled when walked on foot.

Due to limited resource, it is not currently feasible to gather data on the distance between all possible pairs of canteens. In the data collection process, distance between canteens may not be measured directly if there is an intermediate location between the two. For a particular canteen, only the path to the several nearest canteens are measured. This turned out to be a commonly used heuristic when approximating an exact solution to similar problems.

Between a pair of nodes, it is possible that there are multiple paths. In this case, we will only consider an edge that have the shortest distance among all paths that connect the same pair of nodes. The data collection process attempts to include only reasonable paths.

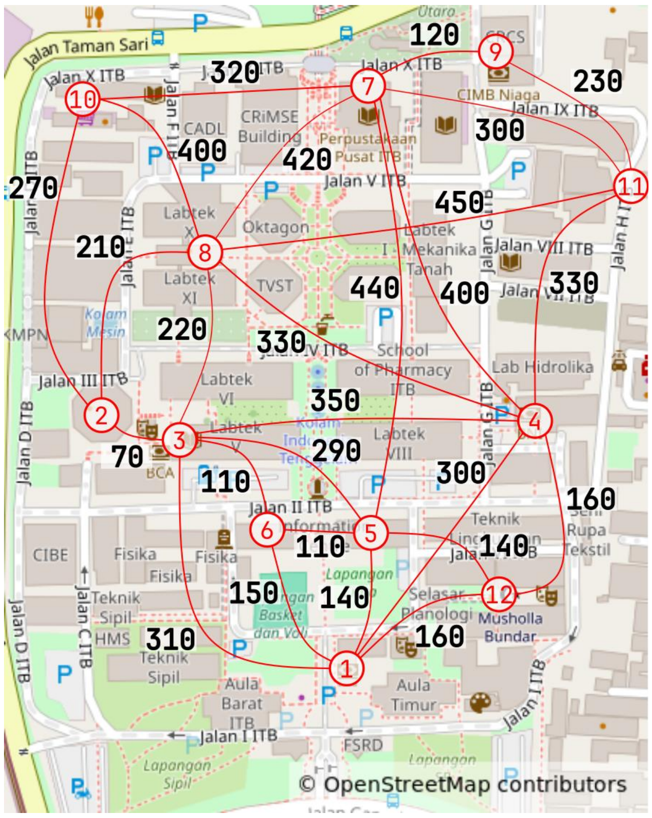


*Fig 2. Labelled weighted graph of the canteens. Weight of each edge is measured as walking distance between inciding nodes in meters. Edges are not drawn to reflect actual pedestrian path*

## *C. Algorithm to search for the most optimal Hamiltonian tour on all canteens*

Based on the previous problem statements, the task of finding a cycle that visits all canteens with the smallest total distance is analogous to the Travelling Salesman Problem. There is a few exact algorithms that solve it. Exact algorithms include a brute-force method and the Held-Karp algorithm which employs dynamic programming techniques. This paper will employ both algorithms, to demonstrate the effects of tabulation on time complexity.

Both of the mentioned algorithms run in exponential time with the brute force method being superexponential. For more than a dozen of nodes, the brute force method runs in an unreasoable time which will be shown in the results section. In practice, algorithms that employ heuristics quickly yield solution close to the exact optimal solution [1].

The brute-force algorithm has a time complexity of $O(n!)$. The brute-force algorithm is equivalent to generating all permutations of nodes, testing for the distance of a path in the order of each permutation. The Held-Karp algorithm, which uses memoization, has a time complexity of $O(n^2 2^n)$. [2]

To search for the optimal tour, there need to be a slight modification on the graph. Due to the limited data, not all pairs of nodes have an edge that coincides with them; the graph is not complete. The Travelling Salesman Problem traditionally assume a complete graph. To addresss this problem, one may (1) assign an arbitrarily large weighted edges to complete the graph, or (2) synthetically construct an edge by searching the incomplete graph for the shortest path between unconnected nodes. We will see that neither choice should affect the resulting optimal tour.

## III. Implementation

Two exact algorithms will be employed to solve the Travelling Salesman Problem. The first is a brute force method by a complete depth-first search of the graph. The search will go on to all nodes, keeping track of nodes that are already in the path. Also note that the graph is represented as a symmetric square matrix.

```
def tsp_brute_force(graph, start, visited=None,
path=None, current_cost=0, best_path=None,
min_cost=float('inf')) -> tuple[list[int], int]:
    if visited is None:
        visited = {start}
    if path is None:
        path = [start]
        if len(visited) == len(graph):
        # Return to the starting node to complete the
tour
        final_cost = current_cost +
graph[start][path[0]]
        if final_cost < min_cost:
            return path + [path[0]], final_cost
        return best_path, min_cost

    for neighbor in range(len(graph)):
        if neighbor not in visited:
            visited.add(neighbor)
            path.append(neighbor)

            best_path, min_cost = tsp_brute_force(
                graph,
                neighbor,
                visited,
                path,
                current_cost + graph[start][neighbor],
                best_path,
                min_cost
            )
            path.pop()
            visited.remove(neighbor)

    return best_path, min_cost
```

The second, more efficient exact algorithm is the Held-Karp algorithm, a classic dynamic programming approach. Instead of building and evaluating one complete path at a time, the Held-Karp algorithm solves the problem by iteratively building up solutions to smaller subproblems. It defines a subproblem as finding the shortest path from a starting node to a different destination node k, visiting a specific subset of intermediate nodes S. It reuses stored result of subproblems to prevent unneeded computation.

```
def tsp_held_karp(graph, start=0):
    """
    Solves the Traveling Salesman Problem using Held-
Karp, a DP algorithm, and reconstructs the path.
    """
    n = len(graph)

    # Memo table
    # memo stores the minimum cost for a given state
(mask, pos)
    memo = {}

    # path_memo stores the next city to visit to
achieve that minimum cost
    path_memo = {}

    def held_karp(mask, pos):
    # Base case: if all cities are visited, return to
the starting node
        if mask == (1 << n) - 1:
            return graph[pos][start]

        # If this subproblem is already solved, return
the stored result
        if (mask, pos) in memo:
            return memo[(mask, pos)]

        min_cost = float('inf')
        best_next_city = -1

        # Iterate over all possible next cities
        for next_city in range(n):
            # If the city has not been visited yet
            if not (mask & (1 << next_city)):
                # Calculate the cost of going to the
next city and then solving the rest of the tour
                new_cost = graph[pos][next_city] +
held_karp(mask | (1 << next_city), next_city)

                if new_cost < min_cost:
                    min_cost = new_cost
                    best_next_city = next_city

        memo[(mask, pos)] = min_cost
        path_memo[(mask, pos)] = best_next_city

        return min_cost

    # Calculate the minimum cost of the tour starting
from start
    min_tour_cost = held_karp(1 << start, start)

    path = []
    current_mask = 1 << start
    current_node = start

    # Follow the path_memo to rebuild the tour
    for _ in range(n - 1):
        path.append(current_node)
        next_node = path_memo[(current_mask,
current_node)]
        current_mask |= (1 << next_node)
        current_node = next_node

    path.append(current_node)
    path.append(start)

   return path, min_tour_cost
```

Additionally, since the data collected does not result in a complete graph, a method to synthetically construct incomplete edges is needed. This code is used to fill in the edges to make a complete graph. The weight of an edge that coincides a pair tof nodes is equal to the distance of the shortest path between them in the original graph.

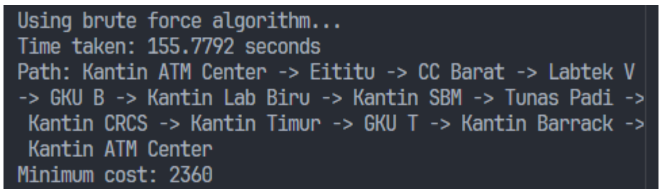
```
def construct_incomplete_edge(graph: list[list[int]]) -
> None:
    """
    Constructs incomplete edges for the graph by
searching for shortest path.
    """

    def find_shortest_path_cost(start: int, end: int) -
> int:
        # UCS algorithm
        queue = [(0, start)]  # (cost, node)
        visited = dict()

        while queue:
            current_cost, current_node =
heapq.heappop(queue)

            if current_node in visited and
visited[current_node] <= current_cost:
                continue

            visited[current_node] = current_cost

            if current_node == end:
                return current_cost

            for neighbor, edge_cost in
enumerate(graph[current_node]):
                if edge_cost > 0:
                    new_cost = current_cost + edge_cost
                    if neighbor not in visited or
new_cost < visited[neighbor]:
                        heapq.heappush(queue,
(new_cost, neighbor))

        return visited.get(end, float('inf'))

    for i in range(len(graph)):
        for j in range(i + 1, len(graph)):
            if graph[i][j] == -1 and graph[j][i] == -1:
                cost = find_shortest_path_cost(i, j)
                graph[i][j] = cost
                graph[j][i] = cost

    return
```

All code and data is available in a GitHub repository linked in the appendix.

## IV. Result and Analysys

### A. *Shortest tour that visits all canteens*

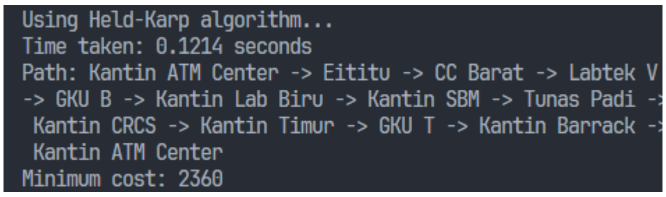
First we compute the optimal tour using the brute force algorithm. Incomplete edges in the graph is filled with weight 10000.



```
Using brute force algorithm...
Time taken: 155.7792 seconds
Path: Kantin ATM Center -> Eititu -> CC Barat -> Labtek V
-> GKU B -> Kantin Lab Biru -> Kantin SBM -> Tunas Padi ->
 Kantin CRCS -> Kantin Timur -> GKU T -> Kantin Barrack ->
 Kantin ATM Center
Minimum cost: 2360
```

*Fig 3. Result using a brute-force algorithm.*

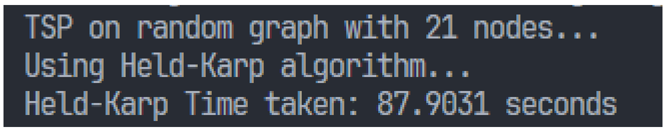
Then, with the same input using the Held-Karp algorithm:



```
Using Held-Karp algorithm...
Time taken: 0.1214 seconds
Path: Kantin ATM Center -> Eititu -> CC Barat -> Labtek V
-> GKU B -> Kantin Lab Biru -> Kantin SBM -> Tunas Padi -
 Kantin CRCS -> Kantin Timur -> GKU T -> Kantin Barrack -
 Kantin ATM Center
Minimum cost: 2360
```

*Fig. Result using Held-Karp algorithm which agrees with the brute-force algorithm.*

Both of the algorithms used above always produce the same result. But it can be seen that Held-Karp massively outperforms the brute-force method even though both of them are exponential algorithms. These algorithms are still impractical for large graph however. For comparison, the Held-Karp algorithm crosses one minute of computation time on a graph of 21 nodes.



```
TSP on random graph with 21 nodes...
Using Held-Karp algorithm...
Held-Karp Time taken: 87.9031 seconds
```

*Fig 4. The time taken for Held-Karp algorithm to find an optimal cycle. The brute-force algorithm took over one minute for a graph with 12 nodes.*

Previously the complete graph is generated by fiiling missing edges with a predetermined weight. By choosing an extremely large weight, it discourages the search process to include such edges. If a hamiltonian circuit exist for the original graph, the optimal path will never include the filled in edges.

With this information, the Hamiltonian tour that visits all canteens with the shortest distance with respect to the available data is as follows

Kantin ATM Center – Eititu – CC Barat – Labtek V – GKU B – Lab Biru – Kantin SBM – Tunas Padi – Kantin CRCS – Kantin Timur – GKU T – Kantin Barrack - Kantin ATM Center
Distance: 2360 meters

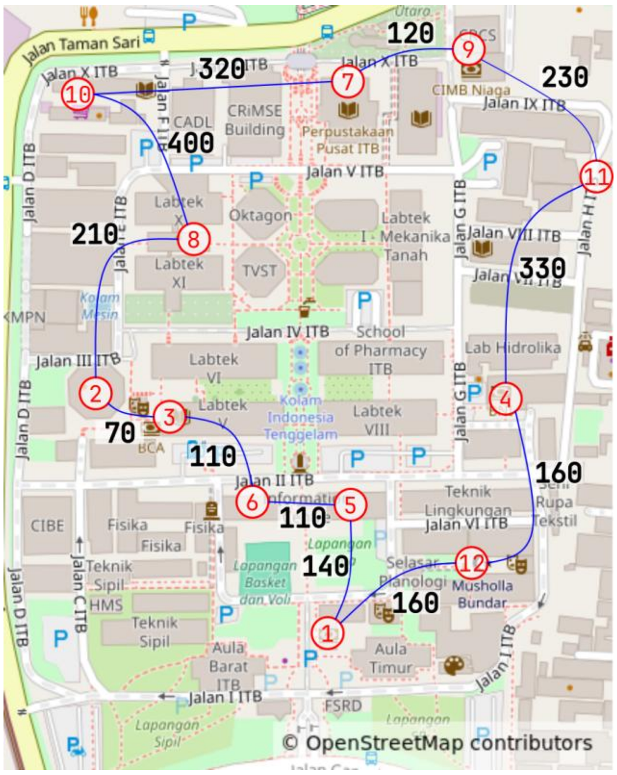
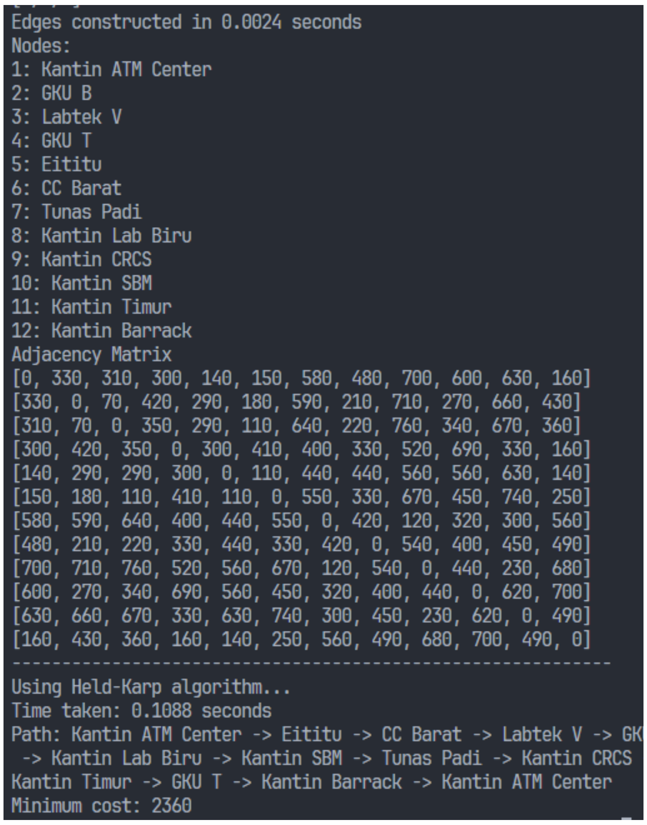


*Fig 5. The optimal tour that visits all canteens with the shortest distance*

### *B. Other method of filling in incomplete edges*

Now we propose another solution to complete missing edges in the graph. For every pair of nodes in a graph, provided that the graph is connected, there is a path between them that has minimal cost. Using this fact, we may construct an artificial edge connecting every missing pair of nodes as having the same weight as the optimal cost.

With this method, the Held-Karp algorithm yield the following result



```
Edges constructed in 0.0024 seconds
Nodes:
1: Kantin ATM Center
2: GKU B
3: Labtek V
4: GKU T
5: Eititu
6: CC Barat
7: Tunas Padi
8: Kantin Lab Biru
9: Kantin CRCS
10: Kantin SBM
11: Kantin Timur
12: Kantin Barrack
Adjacency Matrix
[0, 330, 310, 300, 140, 150, 580, 480, 700, 600, 630, 160]
[330, 0, 70, 420, 290, 180, 590, 210, 710, 270, 660, 430]
[310, 70, 0, 350, 290, 110, 640, 220, 760, 340, 670, 360]
[300, 420, 350, 0, 300, 410, 400, 330, 520, 690, 330, 160]
[140, 290, 290, 300, 0, 110, 440, 440, 560, 560, 630, 140]
[150, 180, 110, 410, 110, 0, 550, 330, 670, 450, 740, 250]
[580, 590, 640, 400, 440, 550, 0, 420, 120, 320, 300, 560]
[480, 210, 220, 330, 440, 330, 420, 0, 540, 400, 450, 490]
[700, 710, 760, 520, 560, 670, 120, 540, 0, 440, 230, 680]
[600, 270, 340, 690, 560, 450, 320, 400, 440, 0, 620, 700]
[630, 660, 670, 330, 630, 740, 300, 450, 230, 620, 0, 490]
[160, 430, 360, 160, 140, 250, 560, 490, 680, 700, 490, 0]
-----------------------------------------------------------
Using Held-Karp algorithm...
Time taken: 0.1088 seconds
Path: Kantin ATM Center -> Eititu -> CC Barat -> Labtek V -> GK
 -> Kantin Lab Biru -> Kantin SBM -> Tunas Padi -> Kantin CRCS
Kantin Timur -> GKU T -> Kantin Barrack -> Kantin ATM Center
Minimum cost: 2360
```

*Fig 6. Adjacency matrix when shortest-path distance method is applied.*

which is the same path produced with the previous method. There is no evidence to assert that the results always agree.

## V. Discussion and Reflection

The algorithms successfully identified the shortest possible tour visiting all 12 selected canteen locations on the ITB Ganesha campus, which was found to be 2360 meters. The optimal path follows the sequence: Kantin ATM Center ⟶ Eititu ⟶ CC Barat ⟶ Labtek V ⟶ GKU B ⟶ Lab Biru ⟶ Kantin SBM ⟶ Tunas Padi ⟶ Kantin CRCS ⟶ Kantin Timur ⟶ GKU T ⟶ Kantin Barrack ⟶ Kantin ATM Center. This provides a practical, efficient route for anyone wishing to undertake a complete "gastronomy tour" of the campus.

A key finding from a computational standpoint is the dramatic difference in performance between the brute-force and Held-Karp algorithms. While both are exact algorithms that yielded the same correct result , the brute-force method's superexponential time complexity resulted in a runtime of over one minute for just 12 nodes. In contrast, the Held-Karp algorithm, which is also exponential, found the solution almost instantly. This empirically demonstrates the effect of algorithmic improvements to the runtime of exponential algorithms.

## VI. Appendix

All code and the collected data is available in the following repository:

https://github.com/rizalkhairan/canteen-tour

## ACKNOWLEDGMENT

The completion of this paper would not have been possible without the support and encouragement of many people. The author extends his deepest gratitude to Dr. Ir. Rinaldi Munir, M.T., whose inspirational teaching provided the foundational knowledge for this project and whose invaluable guidance and constant support were essential to its completion. Special thanks are also extended to Rafi Ramadhan of STI'23 for his willingness to contribute his invaluable time for a part of the data collection process.

## REFERENCES

[1] C. Rego, D. Gamboa, F. Glover dan C. Osterman, "Traveling salesman problem heuristics: leading methods, implementations and latest advances," *European Journal of Operational Research,* 2011.

[2] M. Held dan R. M. Karp, "A Dynamic Programming Approach to Sequencing Problems," *Journal of the Society for Industrial and Applied Mathematics,* 1962.

## 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, 1 Juni 2025

[illegible]

Syahrizal Bani Khairan, 13523063