Advanced Itinerary Optimization in Tokyo Utilizing the A* Algorithm and Graph Theory Principles

Wilson Yusda – 13522019[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522019@mahasiswa.itb.ac.id

***Abstract*—In the dynamic and densely populated city of Tokyo, optimizing travel time is a crucial aspect of itinerary planning for both residents and tourists. This study introduces a groundbreaking approach to itinerary optimization, utilizing the computational efficiency of the A* algorithm, combined with graph theory principles, tailored specifically for Tokyo's complex urban environment. Our methodology is singularly focused on minimizing travel time across the city's extensive network of streets and public transportation systems. The system efficiently navigates through Tokyo's urban landscape, taking into account the intricate web of routes and connections, to deliver time-efficient travel plans. This streamlined approach not only enhances the travel experience by significantly reducing time spent in transit but also sets a new benchmark in the application of advanced algorithms for urban mobility solutions..**

***Keywords*—Itinerary Optimization, A* Algorithm, Graph Theory, Travel Time Efficiency.**

I. INTRODUCTION

In the ever-evolving landscape of urban mobility, the challenge of optimizing travel itineraries in densely populated cities like Tokyo has become increasingly complex. The need for efficient and time-sensitive travel solutions is paramount in such an environment, where every minute counts. This paper delves into an innovative approach to tackle this challenge: Advanced Itinerary Optimization in Tokyo, utilizing the A* Algorithm and Graph Theory Principles. Tokyo, a city known for its intricate and vast transportation network, presents unique challenges and opportunities for advanced computational approaches to itinerary planning.

The A* algorithm, a cornerstone in the field of computer science known for its effectiveness in pathfinding and graph traversal, is adept at finding the most efficient route between two points. When applied to the dense and multifaceted urban grid of Tokyo, this algorithm has the potential to revolutionize how individuals navigate the city. The integration of graph theory further enhances this capability, allowing for a more nuanced understanding and utilization of the complex networks that make up urban landscapes. This combination promises a significant leap in optimizing travel itineraries, particularly in minimizing travel time, which is a critical factor for the fast-paced life of Tokyo's residents and visitors.

This paper aims to explore the application of these advanced computational techniques specifically in the context of Tokyo's unique urban setting. By focusing on the optimization of travel time, the study seeks to offer a solution that is not only theoretically sound but also practically applicable, providing a tool that can significantly improve the daily travel experience of millions. The broader goal is to set a precedent for how cities around the world can leverage technology to address the growing challenges of urban mobility.

II. THEORETICAL BASIS

A. Graph

A graph is defined as a discrete structure consisting of a collection of nodes (vertices) connected through a set of edges. A graph is represented as $G = (V, E)$, where G is the graph, V is a non-empty set of nodes $v1, v2, \dots, vn$, and E is a set of edges $e1, e2, \dots, en$, which connect pairs of nodes in the graph.

In a graph, a pair of nodes can be connected by two different edges, known as multiple edges. There are also edges that start and end at the same node, referred to as loops.

Graphs can be categorized into two types: simple graphs and non-simple graphs. A simple graph is one that does not contain any loops or multiple edges.

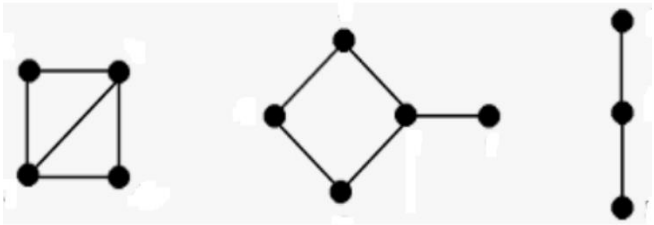


Fig. 1. Simple Graphs (Source: [1])

In contrast, a non-simple graph includes either multiple edges or self-loops. Unsimple graphs can be further divided into two sub-types: multigraphs, which contain multiple edges, and pseudographs, which include self-loops.

Fig. 2. Unsimple Graphs (Source: [1])

Simple graphs are generally easier to process compared to non-simple graphs because they lack repeated edges or edges that connect a node to itself. This simplicity often makes simple

graphs more frequently used in various applications than their non-simple counterparts.

Graphs can be differentiated into two types based on their orientation: directed graphs and undirected graphs. Directed graphs are those in which the edges connecting the nodes have a specific direction, whereas undirected graphs are characterized by edges that connect nodes without any directional orientation.

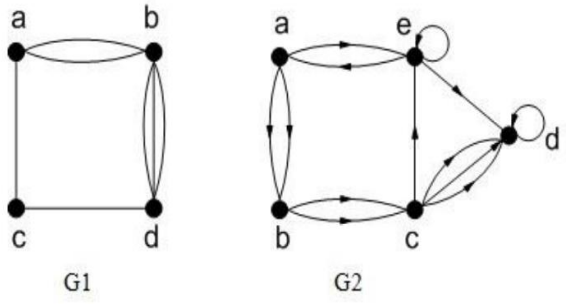


Fig. 3. From left to right, directed graphs, indirected graphs (Source: [1])

Graph theory employs various terms for the analysis of graphs, and the key terminologies include:

1. Adjacent
   In an undirected graph, two vertices are considered adjacent if connected by an edge. In directed graphs, the initial vertex of an edge is adjacent to the terminal vertex. If an edge forms a loop, the initial and terminal vertices coincide.
2. Incidence
   An edge (u, v) connecting u and v is incident with both vertices.
3. Isolated Vertex
   An isolated vertex is a vertex that does not have any adjacent edges or, in other words, lacks neighboring vertices connected by edges.
4. Null Graph or Empty Graph
   A graph whose edge set is an empty set (Nn)" in English. It refers to a graph that has no edges, indicating a complete absence of connections between vertices in the graph.
5. Degree
   In undirected graphs, a vertex's degree is the count of edges incident with it. If a loop exists at a vertex, it is counted twice. Denoted as $deg(v)$, an isolated vertex has no incident edges, and a graph of only isolated vertices is a null or empty graph.
6. Path
   A sequence of edges starting at a vertex, traversing vertices along the graph's edges. Two vertices are connected if a path exists between them. A graph is connected if every vertex pair is connected. In directed graphs, strong connectivity involves directed paths from u to v and vice versa, while weak connectivity relies on replacing directed edges with undirected ones.
7. Circuit or Cycle
   A path that starts and ends at the same node is called a circuit or cycle.
8. Connected
   Two vertices v1 and v2 are said to be connected if there is a path from v1 to v2.
9. Subgraph
   A subset of a graph's vertices and edges, forming a smaller graph with retained properties and connections.
10. Spanning Subgraph
    A spanning subgraph of a graph is a subgraph that includes all the vertices of the original graph.
11. Cut Set
    The cut-set of a connected graph G is a set of edges that, when removed from G, causes G to become disconnected.
12. Weighted Graph
    A weighted graph is a graph where each of its edges is assigned a weight.

To represent a graph, several models can be used, such as:

1. Adjacency Matrix
   In this representation, a matrix sized according to the number of nodes in the graph is used to store information about the edges within the graph. Each element in the matrix indicates whether there is an edge connecting two nodes, the indices of which correspond to the row and column of that element.

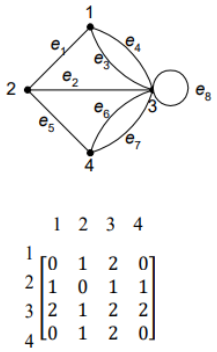


Fig. 4. Adjacency Matrix (Source: [2])

2. Incidence Matrix
   In this matrix, rows are assumed to represent vertices, and columns represent edges. If a vertex, say vertex A, is adjacent to (or incident upon) an edge, say edge E, then this relationship is marked with a 1 in the matrix. Conversely, if there is no adjacency or incidence between the vertex and the edge, it is marked with a 0.

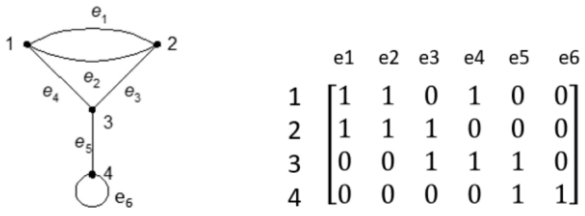
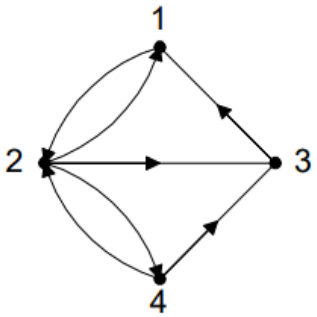


Fig. 5. Incidence Matrix (Source: [2])

3. Adjacency List
   The neighbors of a vertex are represented as a list.



Simpul	Simpul Terminal
1	2
2	1, 3, 4
3	1
4	2, 3

Fig. 6. Adjacency List (Source: [2])

B. A Algorithm*

The A* algorithm, developed by Peter Hart, Nils Nilsson, and Bertram Raphael in 1968, stands as a cornerstone in the field of computer science for pathfinding and graph traversal. Its primary function is to find the shortest path in a weighted graph, a task it accomplishes with remarkable efficiency. This capability has made the A* algorithm an indispensable tool in various applications, ranging from video game development to robotics.

In the context of A* and other search algorithms, a heuristic is a way of estimating the cost to reach the goal from a given node. The heuristic function, often denoted as h(n), is designed to approximate the lowest cost from node n to the goal, without performing an exhaustive search.

In any given subgraph GU and for a chosen set of goals T, consider the function f(n), which represents the real cost of the best possible path that passes through a node n, starting from s and ending at a goal that's best suited for n. We can break down this estimated cost f(n) into two components:

$$f(n) = g(n) + h(n)$$

Here, g(n) is the known cost from the start node to the current node n, and h(n) is the estimated cost from n to the goal. This combined cost function helps A* in deciding which paths are more promising and should be explored first.

The limitation of the A* algorithm lies in its sensitivity to the quality and informativeness of the heuristic function used for guiding the search. While A* is admissible when provided with a lower bound on the cost of reaching the goal, the efficiency of the algorithm is contingent on the accuracy of this heuristic estimate. In scenarios where the available information is insufficient to adequately constrain the set of possible subgraphs at each node, A* may expand more nodes than necessary, compromising its optimality and computational efficiency. The algorithm's performance is thus inherently dependent on the heuristic's ability to provide meaningful guidance in the search for an optimal solution.

III. METHODOLOGY

A. Limitations

When applying advanced itinerary optimization techniques in Tokyo utilizing the A* Algorithm and Graph Theory principles, several limitations must be acknowledged and addressed:

1. The algorithm focuses solely on travel duration by car due to the complexities associated with incorporating multiple modes of transportation.
2. Data subjectivity is a concern, as the information is sourced from a single provider and may not represent diverse user perspectives.
3. Predictive analysis is reliant on API data, which can be inaccurate in unforeseen circumstances or sudden changes in conditions.

B. Data Sample

In this study, the dataset was developed by extracting itinerary place options from Planetyze.com. For each selected location, the corresponding longitude and latitude coordinates were carefully recorded. These geographical coordinates were then plotted in Google Maps to provide a visual representation of each destination and its spatial relationship to others.

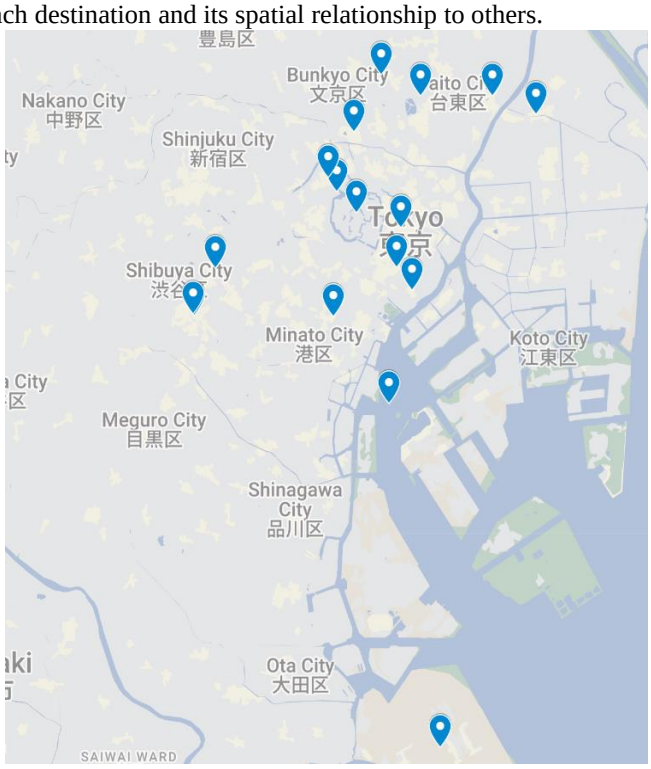


Fig. 7. Datasets in Map Visualization (Source: [3])

For each corresponding point on the map gathered from the source, a careful selection of places was made to be used in this study. These locations, chosen based on their relevance and representation in the dataset, form the basis of our analysis, providing a foundation for examining route efficiency and travel dynamics within the research scope:

Node	Locations	(Latitude,Longitude)
0	Haneda Airport	(35.54939, 139.77983)
1	Hachikō Memorial	(35.65905, 139.70062)

	Statue	
2	Tokyo Skytree	(35.71006, 139.8107)
3	Tokyo Tower	(35.65858, 139.74543)
4	Sensō-ji	(35.71476, 139.79665)
5	Tsukiji Outer Market	(35.66532, 139.77088)
6	Imperial Palace	(35.68517, 139.75279)
7	Nezu Shrine	(35.72013, 139.76076)
8	Art Aquarium Museum	(35.67136, 139.76573)
9	Harajuku Street	(35.67089, 139.7077)
10	Ueno Park	(35.71475, 139.77343)
11	Tokyo Dome	(35.70563, 139.75189)
12	Shibuya Scramble Crossing	(35.65948, 139.70055)
13	Chidorigafuchi Moat	(35.69049, 139.74637)
14	Rainbow Bridge	(35.63656, 139.76314)
15	Yasukuni Jinja	(35.69413, 139.74384)
16	Tokyo Station	(35.68123, 139.76712)

Tabel. 1.Tokyo Tourist Destination Dataset with Latitude and Longitude (Source:[3])

In this dataset configuration, it is assumed that the itinerary planning begins at Haneda Airport, serving as the initial point of arrival for travelers entering Tokyo. The journey is planned to conclude at Tokyo Station, symbolizing a common transit hub for travelers who typically continue their journey to other cities after visiting Tokyo. However, if a traveler wishes to customize their journey, the dataset allows for the modification of both the start and end locations.

C. Problem Modeling

In this problem-solving approach, we begin by adding features to the dataset, utilizing the Graphhopper API to determine the travel time between each pre-selected location. The use of travel time enhances accuracy, as relying solely on distance may overlook various on-site factors. To increase speed, caching is enabled using pickle. If 'graph.pkl' already exists, the system will not create a new graph, thereby ensuring efficiency by reusing existing data.

```python
def initialize_or_load_graph(coordinates, locations_name, api_key, graph_file_path):
    if os.path.exists(graph_file_path):
        with open(graph_file_path, 'rb') as file:
            graph = pickle.load(file)
    else:
        graph = {}
        for i in range(len(coordinates)):
            for j in range(i + 1, len(coordinates)):
                node1, node2 = locations_name[i], locations_name[j]
                coord1, coord2 = coordinates[i], coordinates[j]
                travel_time = get_travel_time(coord1, coord2)
                if node1 not in graph:
                    graph[node1] = {}
                if node2 not in graph:
                    graph[node2] = {}
                graph[node1][node2] = travel_time
                graph[node2][node1] = travel_time
        with open(graph_file_path, 'wb') as file:
            pickle.dump(graph, file)
    return graph
```

Fig. 8. Snapshot of initialize_or_load_graph function (Source: Primary)

```python
def get_travel_time(coord1, coord2):
    endpoint = f'https://graphhopper.com/api/1/route?point={coord1[0]},{coord1[1]}&point={coord2[0]},{coord2[1]}&vehicle=car&key={api_key}'
    response = requests.get(endpoint)
    data = response.json()
    if 'paths' in data and len(data['paths']) > 0:
        travel_time_milliseconds = data['paths'][0]['time']
        travel_time_hours = travel_time_milliseconds/3600000
        with open('tex.json', 'a') as json_file:
            json_file.seek(0, 2)
            if json_file.tell() > 0:
                json_file.write(',')
            json.dump(data, json_file, indent=2)
            json_file.write('\n')
        return travel_time_hours
    else:
        print(f'Error retrieving travel duration from {coord1} to {coord2}')
        return float('inf')
```

Fig. 9. Snapshot of get_travel_time function (Source: Primary)

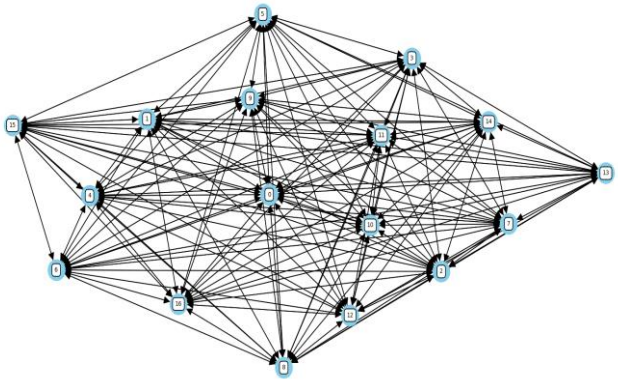


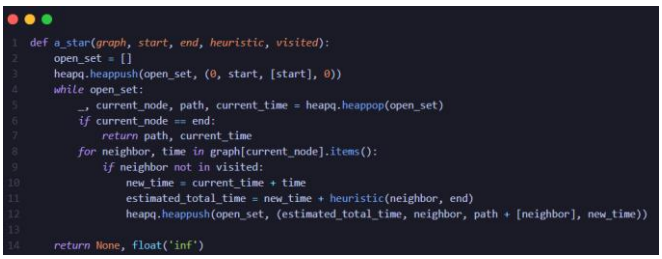
Fig. 10. Graph Visualization of the datasets (Source: Primary)

	Haneda Airport	Hachikō Memorial Statue	Tokyo Skytree	Tokyo Tower	Sensō-ji	Tsukiji Outer Market	Imperial Palace	Nezu Shrine	Art Aquarium museum	Harajuku Street	Ueno Park	Tokyo Dome	Shibuya Scramble Crossing
Hachikō Memorial Statue	0.593457	0.000000	0.294262	0.141043	0.271704	0.182390	0.138479	0.258549	0.142870	0.068580	0.249877	0.179907	0.011647
Tokyo Skytree	0.756747	0.294262	0.000000	0.249886	0.075831	0.194163	0.188715	0.194419	0.172184	0.283846	0.151608	0.191860	0.287129
Tokyo Tower	0.606710	0.141043	0.249886	0.000000	0.231228	0.118098	0.122795	0.238023	0.096801	0.158342	0.199985	0.164222	0.149032
Sensō-ji	0.734188	0.271704	0.075831	0.231228	0.000000	0.155757	0.169820	0.141703	0.134338	0.264951	0.103386	0.149866	0.268234
Tsukiji Outer Market	0.585216	0.182390	0.194163	0.118098	0.155757	0.000000	0.110958	0.187586	0.031478	0.158712	0.138901	0.146064	0.161995
Imperial Palace	0.651916	0.138479	0.188715	0.122795	0.169820	0.110958	0.000000	0.177277	0.086191	0.136516	0.166404	0.098634	0.135947
Nezu Shrine	0.742111	0.258549	0.194419	0.238023	0.141703	0.187586	0.177277	0.000000	0.171153	0.267783	0.082342	0.087321	0.267214
Art Aquarium museum	0.595672	0.142870	0.172184	0.096801	0.134338	0.031478	0.086191	0.171153	0.000000	0.153374	0.138905	0.138234	0.156657

Fig. 11. Snippet of Graph Data with pandas (Source: Primary)

For the graph generated in the previous process, it is subsequently processed by a function called find_route, which requires the parameters graph and start location. Users have the flexibility to decide whether they wish to conclude their itinerary at a specific place from the list or opt to generate the most efficient route without considering the specific end location of the trip.

To begin with, it's essential to look at the main algorithm supporting this program. Here, the A* algorithm is implemented, complete with its heuristic features, forming the backbone of our system.

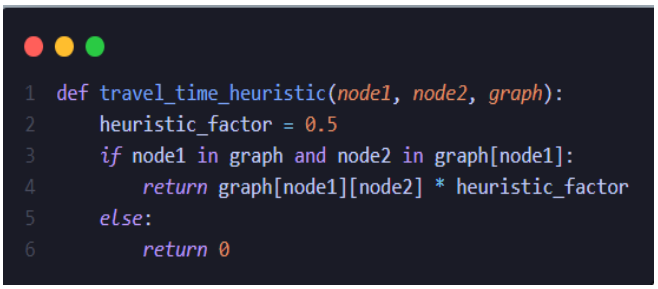


```python
def a_star(graph, start, end, heuristic, visited):
    open_set = []
    heapq.heappush(open_set, (0, start, [start], 0))
    while open_set:
        _, current_node, path, current_time = heapq.heappop(open_set)
        if current_node == end:
            return path, current_time
        for neighbor, time in graph[current_node].items():
            if neighbor not in visited:
                new_time = current_time + time
                estimated_total_time = new_time + heuristic(neighbor, end)
                heapq.heappush(open_set, (estimated_total_time, neighbor, path + [neighbor], new_time))

    return None, float('inf')
```

Fig. 12. Snapshot of a_star function (Source: Primary)

When this function is called, it initializes an 'open set' – a collection of nodes that are to be explored, starting with the start node. The function then enters a loop, continually exploring nodes until it either finds the end node or exhausts all possible paths. Within this loop, the function selects a node from the open set that has the lowest estimated total cost, which is a combination of the actual travel time to reach that node and an estimated time (heuristic) to reach the end node from there. This aligns with the heuristic theory we discussed earlier, where the heuristic function h(n) is designed to estimate the cost from the current node to the end node. If the selected node is the end node, the function concludes by returning the path to this node and the total travel time. Otherwise, it proceeds to examine the node's neighbors, calculating the travel time for each and adding them to the open set for further exploration. This process ensures that the algorithm always prioritizes nodes that are more likely to lead to the quickest path to the destination. The loop continues until the algorithm either finds the most efficient route to the end node or determines that no such route exists.

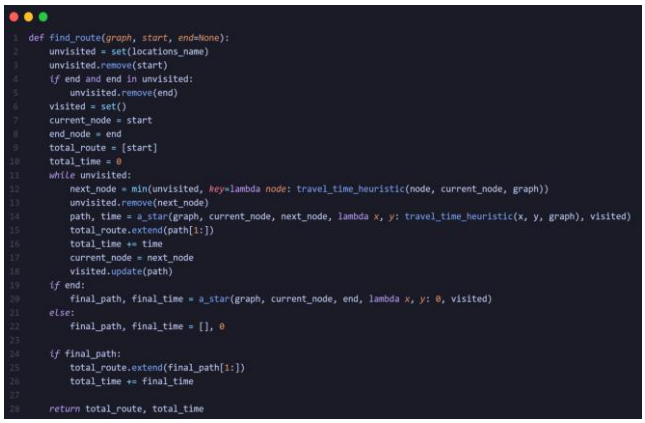


```python
def travel_time_heuristic(node1, node2, graph):
    heuristic_factor = 0.5
    if node1 in graph and node2 in graph[node1]:
        return graph[node1][node2] * heuristic_factor
    else:
        return 0
```

Fig. 13. Snapshot of travel_time_heuristic function (Source: Primary)

In conjunction with the a_star function and the find_route function, the travel_time_heuristic function calculates the distance between two nodes, utilizing the heuristic factor, which plays a crucial role in the efficiency of the algorithm. The heuristic factor's importance lies in its ability to estimate the cost (in this case, time) from the current node to the destination node. This estimation guides the algorithm in selecting the most promising path forward, balancing between the known path and potential future paths. The "estimated total time" is thus a sum of two components: the actual time already spent traveling from the start node to the current node, and the heuristic estimate of the time required to reach the end node from the current node. This method enables the A* algorithm to make informed decisions at each step, optimizing the route for the shortest possible total travel time.

The core of our calculations is centered around the find_route function. This function takes a graph and a starting location as inputs, along with an optional end location.



```python
def find_route(graph, start, end=None):
    unvisited = set(locations_name)
    unvisited.remove(start)
    if end and end in unvisited:
        unvisited.remove(end)
    visited = set()
    current_node = start
    end_node = end
    total_route = [start]
    total_time = 0
    while unvisited:
        next_node = min(unvisited, key=lambda node: travel_time_heuristic(node, current_node, graph))
        unvisited.remove(next_node)
        path, time = a_star(graph, current_node, next_node, lambda x, y: travel_time_heuristic(x, y, graph), visited)
        total_route.extend(path[1:])
        total_time += time
        current_node = next_node
        visited.update(path)
    if end:
        final_path, final_time = a_star(graph, current_node, end, lambda x, y: 0, visited)
    else:
        final_path, final_time = [], 0

    if final_path:
        total_route.extend(final_path[1:])
        total_time += final_time

    return total_route, total_time
```

Fig. 14. Snapshot of find_route function (Source: Primary)

The find_route function serves as the primary component for calculating the total travel time and determining the most efficient route. This function is versatile, adapting to user requests which can vary between selecting a specific endpoint for the journey or opting to discover the most efficient route without a predetermined end location.

Initially, it creates a set of all locations (unvisited) and removes the starting location, and if provided, the ending location. The algorithm then iteratively finds the next best node to visit from the remaining unvisited nodes, using the travel_time_heuristic function to determine this. This heuristic guides the selection by estimating the travel time from the current node to each unvisited node, choosing the one with the minimum estimated time. Once a next node is chosen, the a_star function is called to find the most efficient path to this next node, updating the total route and time accordingly.

Finally, to enhance the user experience, we include an additional feature that visualizes the route calculated by the find_route function. This feature opens a new tab displaying a map that highlights the chosen destinations and their respective routes, offering users a clear and interactive overview of their journey.

```python
def display_route_on_map(route, coordinates, locations_name):
    map_center = coordinates[locations_name.index(route[0])]
    my_map = folium.Map(location=map_center, zoom_start=13)
    route_coords = []
    for i, location in enumerate(route):
        index = locations_name.index(location)
        coord = coordinates[index]
        route_coords.append(coord)
        if i == 0 :
            marker_color = 'red'
        elif i == len(route) - 1:
            marker_color = 'green'
        else:
            marker_color = 'blue'
        popup_text = f"{i + 1}. {location}"
        folium.Marker(
            location=coord,
            popup=popup_text,
            icon=folium.Icon(color=marker_color)
        ).add_to(my_map)
    plugins.AntPath(locations=route_coords, color='blue', weight=5, delay=1000).add_to(my_map)
    my_map.save('route_map.html')
    import webbrowser
    webbrowser.open('route_map.html', new=2)
```

Fig. 15. Snapshot of display_route_on_map function (Source: Primary)

In the map visualization provided by our function, the starting point (Haneda Airport) is marked with a red marker, while the

end point (Tokyo Station or any selected end point for the most efficient route) is highlighted with a green marker. Each intermediate stop along the route is denoted by blue markers. Additionally, the function utilizes the coordinates from the dataset to accurately represent the actual positions of these points. The path connecting these locations is clearly traced with a striped blue line, offering an interactive and dynamic view on an HTML-based website, which is a more effective and user-friendly approach compared to static images.

IV. ALGORITHM AND METHOD ANALYSIS

To further substantiate our research, testing is conducted, followed by analysis. In this section, two test cases will be presented. The first case involves the user selecting Tokyo Station as the final stop. In the second test case, the user designates Tokyo Station as a key spot and seeks the most efficient route without specifying an end destination.

Initially, we conduct a test on the route where Tokyo Station is set as our final destination. To accomplish this, enter the word 'final' as the input on the provided interface.

```
Do you want to make Tokyo Station the final destination or a stop on the route? Enter 'final' for final destination or '
stop' for a stop on the route: final
Route:
[Haneda Airport] -> [Rainbow Bridge] -> [Tsukiji Outer Market] -> [Art Aquarium museum] -> [Imperial Palace] -> [Chidori
gafuchi Moat] -> [Yasukuni Jinja] -> [Tokyo Dome] -> [Nezu Shrine] -> [Ueno Park] -> [Sensō-ji] -> [Tokyo Skytree] -> [T
okyo Tower] -> [Hachikō Memorial Statue] -> [Shibuya Scramble Crossing] -> [Harajuku Street] -> [Tokyo Station]
Total Time (hours): 1.9041563888888893
Do you want to display the route map? (yes/no): yes
```

Fig. 16. Snapshot of Test Case 1 Terminal Input (Source: Primary)

As demonstrated in the code snippet, the program is designed to find the most efficient route from Haneda Airport, which is the starting point, to Tokyo Station, set as our final destination. Multiple tests have been conducted to confirm that this indeed represents the quickest route in terms of travel time. To further assess the effectiveness of our algorithm, we intentionally included nodes that are in close proximity to each other and observed how they are connected in the route. Additionally, we have incorporated a route visualization feature. This not only aids in better understanding the efficiency of the proposed route but also simplifies navigation for the users, providing them with a more intuitive travel experience. As seen above, the total time needed to complete the most efficient route is 1.90 hours with the route in the order of :

[Haneda Airport] -> [Rainbow Bridge] -> [Tsukiji Outer Market] -> [Art Aquarium museum] -> [Imperial Palace] -> [Chidorigafuchi Moat] -> [Yasukuni Jinja] -> [Tokyo Dome] -> [Nezu Shrine] -> [Ueno Park] -> [Sensō-ji] -> [Tokyo Skytree] -> [Tokyo Tower] -> [Hachikō Memorial Statue] -> [Shibuya Scramble Crossing] -> [Harajuku Street] -> [Tokyo Station]

The effectiveness of the route and the success of our research are substantiated by anticipated viewpoints. Firstly, the absence of repeated nodes signifies that no location is visited more than once. Additionally, the program prioritizes minimizing loops, as evident in the visualization below where the program endeavors to reduce loop occurrences.

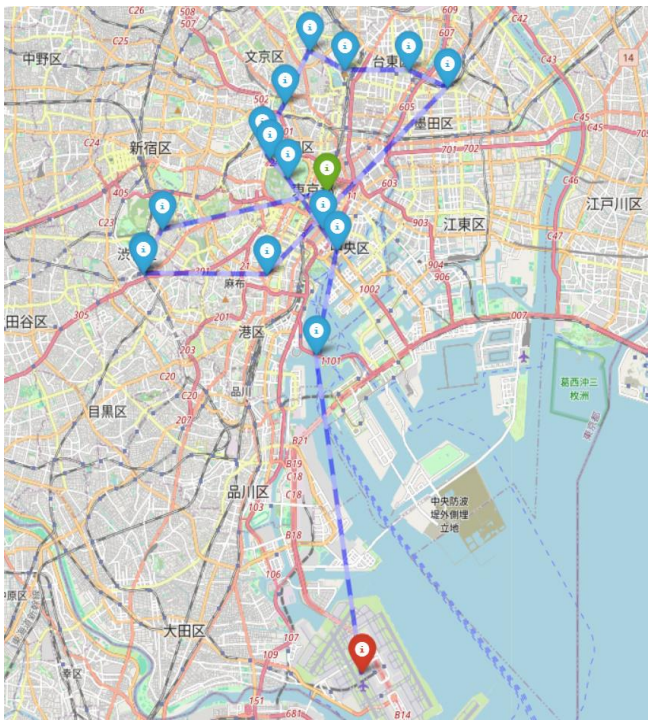


Fig. 17. Map Visualization of Results Route For Test Case 1 (Source: Primary)

Fig. 18. Closer Map Inspection on Test Case 1 (Source: Primary)

The image presented above, though initially appearing somewhat ambiguous, effectively demonstrates the program's objective to minimize travel time as much as possible. This is evident in the way the program strategically avoids unnecessary loops within the Japanese terrain. While some looping may be observed, these are primarily attributed to specific route planning requirements and the terrain's characteristics. To further illustrate this point, consider the proximity between the Hachikō Memorial Statue and the Shibuya Scramble Crossing. The presence of a direct edge between these nodes in the program's output substantiates that the algorithm efficiently connects close locations. This example is representative of the program's approach to every node in the dataset, showcasing the effective application of the A* algorithm across all points.

Now, let's shift our focus to scenarios where users seek a comprehensive exploration of Japan, desiring to experience the best of all itineraries without a specific end destination in mind. In such cases, the program meticulously calculates the most optimal routes covering all planned stops. This approach is also rooted in the A* algorithm, where for each node, the program aims to find the most efficient route based on travel time and the

applied heuristic. This ensures that users experience a well-rounded journey, seamlessly connecting the highlights of Japan in a manner that prioritizes both time efficiency and the richness of the travel experience. To accomplish this, enter the word 'stop' as the input on the provided interface

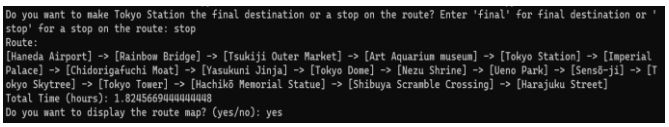


Fig. 19. Snapshot of Test Case 2 Terminal Input (Source: Primary)

As observed, the program generates a faster travel time when the end destination is left undefined, as compared to situations where the end point is predetermined. This makes intuitive sense, considering that a specified end destination is often more about personal preference than route efficiency. However, when the focus is shifted to maximizing route effectiveness, the program excels in delivering the most efficient path, thereby saving travel time for the user. This approach allows travelers to spend less time in transit and more time enjoying the various attractions included in their tour. We can now determine that the most efficient route for our dataset concludes at Harajuku Street. This route has been proven to be the most time-effective, with a total travel time of 1.825 hours with the route in the order of: [Haneda Airport] -> [Rainbow Bridge] -> [Tsukiji Outer Market] -> [Art Aquarium museum] -> [Tokyo Station] -> [Imperial Palace] -> [Chidorigafuchi Moat] -> [Yasukuni Jinja] -> [Tokyo Dome] -> [Nezu Shrine] -> [Ueno Park] -> [Sensō-ji] -> [Tokyo Skytree] -> [Tokyo Tower] -> [Hachikō Memorial Statue] -> [Shibuya Scramble Crossing] -> [Harajuku Street]

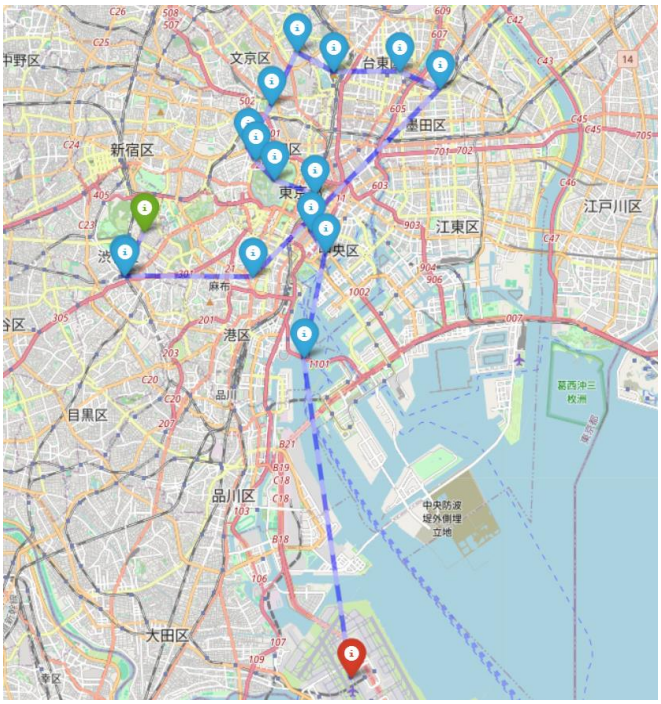


Fig. 20. Map Visualization of Results Route For Test Case 2 (Source: Primary)

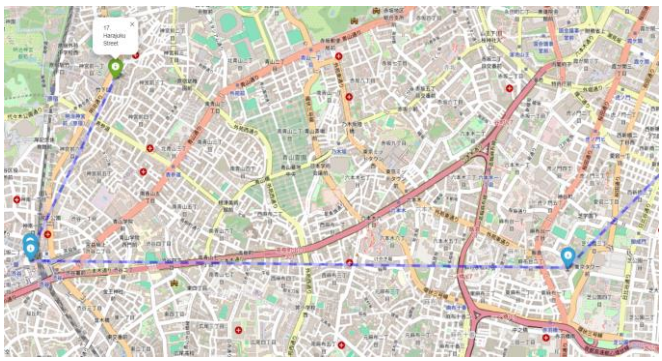


Fig. 21. Closer Map Inspection on Test Case 2 (Source: Primary)

We can also clearly observe that the route adapts accordingly while still striving to minimize unnecessary loops. This demonstrates the program's efficiency in route optimization. Furthermore, our standing hypothesis which imply that closely situated nodes would be connected sequentially is also validated in these scenarios. This consistency in the program's behavior further reinforces the reliability and effectiveness of our routing algorithm, showcasing its capability to adapt to different user preferences while maintaining optimal travel efficiency.

As we draw this testing and analysis phase to a close, it's important to reflect on the insights gained and the potential areas for further development. Throughout our rigorous examination of the program, we've uncovered valuable data that not only validates the effectiveness of the A* algorithm in practical scenarios but also highlights areas where improvements can be made.

Our analysis also brought to light certain limitations. These include the need for greater adaptability in dynamic environments and the potential for integrating real-time data to enhance route accuracy. Future iterations of the program could benefit from incorporating live traffic updates, weather conditions, and user preferences to provide even more tailored and responsive route suggestions.

However, our testing and analysis have laid a solid foundation for future enhancements. By leveraging the A* algorithm, we've demonstrated that it's possible to significantly improve how we navigate our world. This isn't just about reaching a destination; it's about doing so in the most efficient manner possible, ensuring that every journey is not only well-planned but also a delightful and smooth experience. The insights gained from this project pave the way for more advanced and user-friendly travel planning tools, making every trip an optimized and enjoyable adventure.

V. CONCLUSION

In conclusion, our project 'Advanced Itinerary Optimization in Tokyo Utilizing the A* Algorithm and Graph Theory Principles' has not only showcased the robust capabilities of these advanced computational methods in enhancing urban travel planning but also illuminated a pathway for future enhancements. While the current system adeptly navigates Tokyo's complex network of destinations, delivering optimized routes and significantly improving travel efficiency, we acknowledge certain limitations that present opportunities for further development.

One of the main opportunities for improvement is to better understand what users really want. This means considering things like how they prefer to travel, what they like to do, and how much time they have. This involves tailoring our system to better align with individual needs, thereby offering a more customized travel experience. Plus, we know that having better and more data is key to making our system more accurate and trustworthy. So, we'll focus on expanding and improving our dataset to get even better results.

Looking ahead, the potential to enhance and expand our itinerary optimization system using the A* algorithm is vast. Our aim is to integrate user feedback and the latest technological advances to better understand urban travel dynamics. This will allow us to refine our system, making it not just a tool for finding efficient routes, but one that fully considers each individual's travel preferences and the unique character of different cityscapes. We're committed to evolving beyond basic route optimization, envisioning a system that enriches every aspect of travel planning, turning every journey into an efficient, yet deeply personal and unforgettable experience.

VI. Appendix

The complete A* algorithm and many other functions can be found below.
https://github.com/Razark-Y/RouteFinder-for-Travel-Itinerary-in-Tokyo

VII. Acknowledgment

The author expresses deep gratitude to God Almighty for His grace and guidance, which have been instrumental in the successful completion of this paper titled “Advanced Itinerary Optimization in Tokyo Utilizing the A* Algorithm and Graph Theory Principles.” The author would like to thank everyone involved in the preparation of this paper, namely:

1. Dr. Nur Ulfa Maulidevi, S.T., M.Sc., Dr. Rinaldi Munir, M.T., and Dr. Fariska Zahra Zakhralativa Ruskanda, S.T., M.T., the lecturers of the IF2120 Discrete Mathematics course, for their guidance and knowledge imparted during the lectures,
2. The author's parents, for their constant encouragement and support,
3. Friends and peers, for their support and contributions in the drafting and refining of this paper.

Finally, the author thanks the readers. Apologies are extended for any errors in writing or content. It is the author's sincere hope that this paper proves beneficial to its readers.

References

[1] R. Munir, “Graf Bagian 1,” IF2120 Matematika Diskrit. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/19-Graf-Bagian1-2023.pdf [Accessed 9 December 2023]

[2] R. Munir, “Graf Bagian 2,” IF2120 Matematika Diskrit. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/20-Graf-Bagian2-2023.pdf [Accessed 9 December 2023]

[3] Planetyze, “The best 50 places to visit in Tokyo,” [Online]. Available: https://planetyze.com/en/japan/tokyo/blog/the-best-50-places-to-visit-in-tokyo [Accessed 9 December 2023]

[4] Hart, Nilsson dkk. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions of Systems Science and Cybernetics, Vol. 4.

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, 11 Desember 2023

Wilson Yusda 13522019