Route Optimization for Domain Exploration in Mondstadt, Genshin Impact with Travelling Salesman Problem Approach

Three Gie Gendhis Sekar Ayoe Jatmiko - 13525144
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: threegie.gendhis19@gmail.com , 13525144@std.itb.ac.id

***Abstract*—Genshin Impact is an open world adventure game released in 2020. One of the way to collect the materials in the game is by unlocking and get through the domain that spreads across the nation in the game. Due to the distribution of the domains in Mondstadt, new players tend to struggle to visit all the domains, considering the time consumption it needs too. This paper uses the application of graph theory to find the solution, by modelling the map of Mondstadt with domains as the set of vertices and the route representing the edges of the graph. The Travelling Salesman Problem (TSP) approach is needed to find the shortest route possible, so players could visit every domain exactly once, just in one-go. The resulting route provides a more efficient way to explore Mondstadt's domain.**

Keywords—Graph Theory; Genshin Impact; World Exploration; Route Optimization; Travelling Salesman Problem (TSP)

I. Introduction

Genshin Impact is an open world adventure Role Playing Game (RPG), developed by a Chinese Studio, Mihoyo (now called as Hoyoverse after a rebranding in 2022) and officially released in 2020. Genshin has gained a wide popularity across the world over some factors, such as the storyline, characters, gacha, and engaging combat mechanics that depend on elemental reactions.

In Genshin's world exploration system, players could also grind materials to enhance their character's level or just to obtain primogems (the currency in the game) by entering types of domain provided by the game. Domains are widely spread in the nations in the Genshin verse, and one of the nations is named after Mondstadt. In Mondstadt, there is a total of ** including domain of artifacts, weapon enhancement materials, and battlefield.

Due to the geographical distribution of domains across Mondstadt, new players may encounter difficulties to visit all of the domains to complete their exploration, especially if they lack the resources to do so. This is due to the likelihood that the player has not yet acquired the ability (higher stamina bar) or characters with great exploration skill that helps.

Graph theory can be used to become an alternative to this problem – to optimize exploration time and make sure all domains could get unlocked at one go – by using Travelling Salesman Problem (TSP) approach. By modelling the nation's map to a graph model, the domains are represented as the nodes and the accessible routes are represented as its edges. The type of graph used in this theory is the weighted graph, indicating that each existing edge has a numerical value that will represent the distance between the domains.

Therefore, this paper aims to determine the most efficient route for visiting all domains in Mondstadt using a Travelling Salesman Problem approach. The resulting route can help players optimize exploration time, improve exploration efficiency, and potentially accelerate the collection of in-game resources and rewards, including primogems.

II. Theoretical Foundation

A. Graph Theory

In order to understand the algorithm to the optimization graph theory is needed for further discussions.

1) Graph Definition

A graph G is defined as G = (*V*,*E*) in which *V* is a nonempty set of vertices and *E* is a set of edges connecting the vertices.

A simple graph is a graph G in which each edge connects two different vertices and there is no two edges connected to the same pair of vertices. A complete graph on n vertices (K_n), is a simple graph G in which every pair of its vertices is connected by exactly one edge.

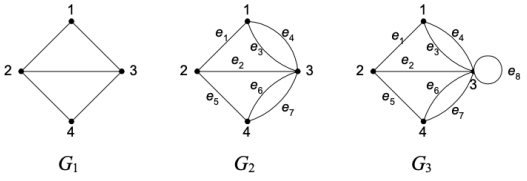

Figure 2.1 (a) Simple graph, (b) Dual graph, (c) Pseudo graph

Source : Gambar 2, Rinaldi Munir, 2026, https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/20-Graf-Bagian1-2026.pdf

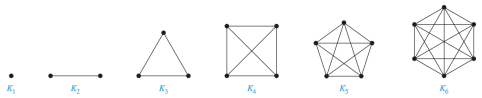

Figure 2.2 Complete graphs
Source : Figure 3, Kenneth H., 2019.

2) Terminologies

Several terminologies needed in order to understand further concepts of graph.

a) Adjacency

Two vertices in a graph G are called adjacent if both two are the endpoints of an edge of G.

b) Incidency

An edge e is called incident with the vertices *u* and *v* if *e* is said to connect *u* and *v*.

c) Weighted Graph

A weighted graph is a graph G which on each of its edges is given a number attached to it. Weighted graphs are often used to model a system, in which the weight of the edges depend on the context it is meant to. The weight may represent prices, distances, or durations of its connecting vertices.

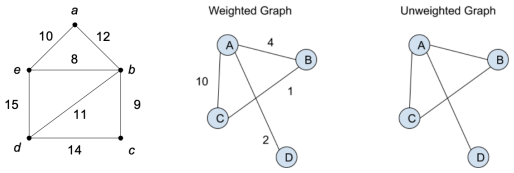

Figure 2.3 Weighted graph and unweighted graph comparison.
Source : Rinaldi Munir, 2026, https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/20-Graf-Bagian1-2026.pdf

d) Path

A path is a sequence of vertices connected by edges, beginning at an initial vertex and ending at a terminal vertex.

e) Cycle / Circuit

A cycle is a path which starts and ends at the same vertice. The length of a cycle in a graph G equals the number of edges it has.

3) Hamiltonian Concept

The Hamiltonian concept is used in representing a graph with special traits on its contained path and cycle. A simple path in a graph G that passes through every vertex once is called a Hamilton path. Otherwise, a Hamilton cycle is a simple circuit in graph G that passes through every vertex exactly once, except for the initial vertex which got passed through two times.

A graph G is called a Hamiltonian graph if it has a hamiltonian cycle, otherwise, it is called a semi-Hamiltonian graph if it only has a Hamiltonian path.

B. Graph Representation

For computation purposes, graph can be represented in several forms, such as matrix and list.

1) Adjacency Matrix

Let n is the number of vertices of graph G. The adjacency matrix of G is a *n* x *n* binary matrix (matrix with its value strictly the elements of domain of {0,1}), with 1 as its *(i,j)*th entry if the two vertices are adjacent, and 0 otherwise.

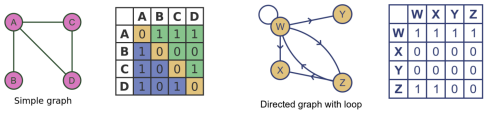

Figure 2.4 Adjacency matrix
Source : Rinaldi Munir, 2026, https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/21-Graf-Bagian2-2026.pdf

2) Incidency Matrix

Let *G = (V,E)* be an undirected graph, with *v* are its vertices and *e* are its edges. Then the incidence matrix labels are the orderings to the numbers of the vertices and edges, which the elements are in the domain of binary numbers {0,1} which 1 represents the condition when a certain edge is incident with a vertice, and 0 if otherwise.

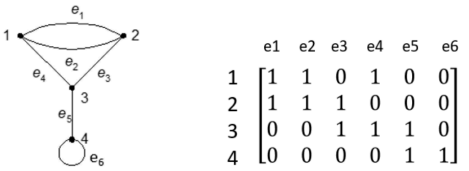

Figure 2.5 Incidency matrix
Source : Rinaldi Munir, 2026, https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/21-Graf-Bagian2-2026.pdf

3) Adjacency List

Adjacency list specify the vertices that are adjacent to each vertex of the graph.

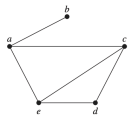

FIGURE 1 A simple graph.

TABLE 1 An Adjacency List for a Simple Graph.

Vertex	Adjacent Vertices
a	b, c, e
b	a
c	a, d, e
d	c, e
e	a, c, d

Figure 2.6 Example of graph representation in adjacency list.
Source : Figure 1, Kenneth H., 2019.

C. Travelling Salesman Problem

Travelling Salesman Problem (TSP) is an optimization solution which uses the application of Hamilton circuits in finding the shortest route possible for visiting checkpoints (in this case, the checkpoints are vertices of a graph *G*). TSP is the optimization solution for finding a Hamilton cycle in a

weighted graph but with the least total weight possible. However, knowing that teleportation is possible in the game, players are able to use teleportation after unlocking the domains. Therefore, TSP implementation in this study uses Hamiltonian path instead of Hamiltonian circuits knowing that it's not necessarily to go back to the starting domain or vertice.

D. Held-Karp Algorithm

The Held-Karp algorithm is a dynamic programming approach which implements the optimization solution in TSP. In short, dynamic programming is an algorithm technique in which abstracting the problems by breaking down the bigger problems into smaller subproblems to find the solution to be stored for further iterations. Compared to the brute-force approach with a time complexity of O(n!), the Held-Karp algorithm improves the efficiency of solving TSP to $O(n^2 2^n)$.

In Held-Karp, in order to find the shortest path, it is not strictly by finding routes based on order, as looking for the shortest route between A and B, then shortest route to C until it reaches D. Instead, this algorithm will continuously evaluate subsets of visited destinations and computes the minimum cost required to reach a specific destination after visiting all destinations within the subset.

Let *A, B, C,* and *D* be the destinations to get to, where A is the starting point to begin with. The algorithm will start with calculating distance between two destinations, which are *({A,B},B), ({A,C},C),* and *({A,D}, D)*. The notation used to represent the state in the brackets are consecutively the set of checkpoints visited in the state followed by its ending point. Further on, the distances obtained will be stored to calculate the next states in the iterations. In this case, it will continue on calculating distance between 3 checkpoints, which are *({A,B,C},C), ({A,B,D},D), ({A,C,D},D), ({A,C,B},B), ({A,D,B},B),* and *({A,D,C},C)* using the prior information. As a result, this algorithm will produce the minimum-cost of the possible states.

The explanation above shows that the algorithm also uses recursivity to work. This is due to in order to find the total distance, the distance to every checkpoint (to obtain states for further number of checkpoints) are needed. Assume that the algorithm will calculate the total distance of these particular states from the example above. To compute the state *({A,B,C,D},D)*, the algorithm evaluates multiple previously computed states such as *({A,B,C},B)* and *({A,B,C},C)*, and selects the one that yields the minimum total cost.

In general terms, the recursive relation used by the Held-Karp algorithm is given by $T(S,d)=min[T(S-\{d\},i)+w(i,d)]$, where

- $T(S,d)$ = minimum cost required to visit all destinations in subset S and end at destination d,
- $S-\{d\}$ = subset obtained by removing destination d from S,
- i = predecessor destination visited before d, and
- $w(i,d)$ = distance between destinations i and d.

III. METHODOLOGIES

The methodologies used to imply the Held-Karp algorithm consists of data collection and graph modelling. The data collection part is needed to collect the quantitative data of each domain location, in the need of the domain's distance between each other. Then, it will be showed as the weight of the edges in the resulting graph.

A. Data Collection

The data preparations needed to implement the Held-Karp algorithm begins with representing the Mondstadt map in a weighted graph, in which In order to show the more specific data added in this process, table 3.1 as shown below will help to give a more brief description about the point (representing the domain's name) and the *x* and *y* coordinate. In order to do that, the domains are declared in Cartesian coordinates *(x,y)*.

1) Map Visualization

The overall Mondstadt region used in this paper are shown as the figure below, contains a total of fourteen domains name distributed in different locations across many areas in the nation.

Figure 3.1 Mondstadt map
Source : Teyvat Interactive Map, 2026, https://act.hoyolab.com/ys/app/interactive-map/index.html?lang=en-us#/map/

2) Coordinate Acquisition

In order to obtain the distance between domains, each domain has to be assigned in Cartesian coordinates (*x,y*). The software used to obtain the coordinate is GeoGebra Calculator Suite and for simplicity reasons, each domain is renamed after alphabetical names *A* to *N* to label each point representing each fourteen domains.

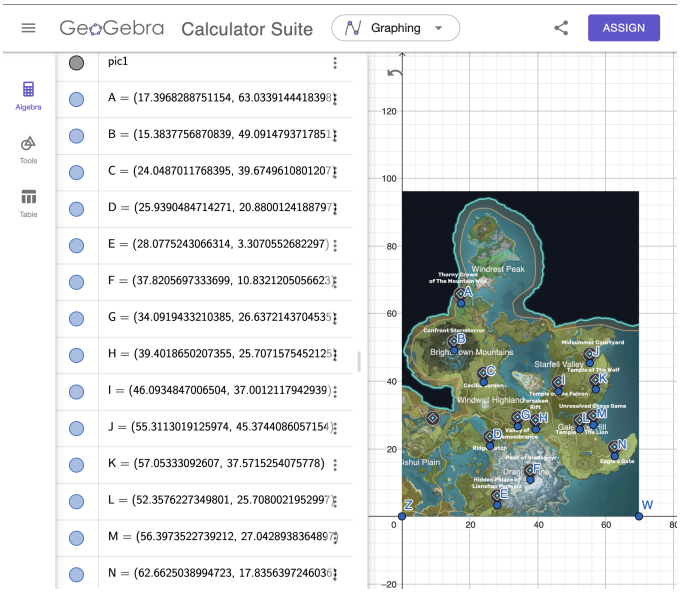

Figure 3.2 Coordinate extraction for Mondstadt map in Geogebra
Source : Author's documentation, 2026.

Table 3.1 as shown below will give a summary about the correspondence between the named points and the domain's name, also its corresponding Cartesian coordinates.

Point	Domain's Name	*x*	*y*
A	Thorny Crown of The Mountain Wild	17.40	63.03
B	Confront Stormterror	15.38	49.09
C	Cecilia Garden	24.05	39.67
D	Ridge Watch	25.94	20.88
E	Hidden Place of Lianshan Formula	28.08	3.31
F	Peak of Vindagnyr	37.82	10.83
G	Valley of Remembrance	34.09	26.64
H	Forsaken Rift	39.40	25.71
I	Temple of The Falcon	46.09	37.00
J	Midsummer Courtyard	55.31	45.37
K	Temple of The Wolf	57.05	37.57
L	Temple of The Lion	52.36	25.71
M	Unresolved Chess Game	56.40	27.04
N	Eagle's Gate	62.66	17.84

Table 3.1 Mondstadt's domain in *x* and *y* coordinate

3) Distance Calculation

After obtaining each domain's coordinate, the distance between every domain could possibly be obtained by using the Euclidean formula. The 2D Euclidean distance formula calculate the distance between two points in a straight line as shown below

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

where

- d is the distance between two domains,
- (*x1,y1*) is the coordinate of the first domain,
- (*x2,y2*) is the coordinate of the second domain.

For more effective and better time efficiency, the Euclidian distance between certain domains into another 13 domains are calculated in a Python program shown below.

```
import math

# Domain coordinates
A = [17.40, 63.03]
B = [15.38, 49.09]
C = [24.05, 39.67]
D = [25.94, 20.88]
E = [28.08, 3.31]
F = [37.82, 10.83]
G = [34.09, 26.64]
H = [39.40, 25.71]
I = [46.09, 37.00]
J = [55.31, 45.37]
K = [57.05, 37.57]
L = [52.36, 25.71]
M = [56.40, 27.04]
N = [62.66, 17.84]

labels = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
'K', 'L', 'M', 'N']
points = [A, B, C, D, E, F, G, H, I, J, K, L, M, N]

n = len(points)

dist_matrix = []
for i in range(n):
    row = [0.0] * n
    dist_matrix.append(row)


for i in range(n):
    for j in range(n):
```

```
        if i != j:
                    dist_matrix[i][j]  = math.dist(points[i],
points[j])

if __name__ == "__main__":
    for i in range(n):
        for j in range(n):
            if i != j:
                point_i = labels[i]
                point_j = labels[j]
                distance = dist_matrix[i][j]
                    print("Distance", point_i, "->", point_j,
":", round(distance, 2))
```

```
PROBLEMS   OUTPUT   ···                Filter (e.g. text, !excludeTex...

[Running] python3 -u "/Users/gie/Documents/matdis/distance.py"
Distance A -> B : 14.09
Distance A -> C : 24.29
Distance A -> D : 43.01
Distance A -> E : 60.67
Distance A -> F : 56.05
Distance A -> G : 40.03
Distance A -> H : 43.32
Distance A -> I : 38.74
Distance A -> J : 41.82
Distance A -> K : 47.12
Distance A -> L : 51.14
Distance A -> M : 53.07
Distance A -> N : 63.96
```

Figure 3.3 Result of the python program to calculate Euclidean distance for domain *A* to another 13 domains.
Source : Author's documentation

Table 3.2 gives a summary about the distances obtained for each domain after the execution of the program.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
A	0	14.09	24.29	43.01	60.67	56.05	40.03	43.32	38.74	41.82	47.12	51.14	53.07	63.96
B	14.09	0	12.80	30.12	47.51	44.36	29.22	33.52	33.00	40.10	43.23	43.75	46.57	56.67
C	24.29	12.80	0	18.88	36.58	31.96	16.45	20.75	22.20	31.78	33.07	31.56	34.73	44.35
D	43.01	30.12	18.88	0	17.70	15.56	9.98	14.30	25.80	38.24	35.30	26.86	31.08	36.85
E	60.67	47.51	36.58	17.70	0	12.31	24.09	25.10	38.20	50.11	44.87	33.03	36.95	37.51
F	56.05	44.36	31.96	15.56	12.31	0	16.24	14.96	27.45	38.72	32.94	20.80	24.66	25.81
G	40.03	29.22	16.45	9.98	24.09	16.24	0	5.39	15.85	28.30	25.43	18.29	22.31	29.89
H	43.32	33.52	20.75	14.30	25.10	14.96	5.39	0	13.12	25.29	21.26	12.96	17.05	24.56
I	38.74	33.00	22.20	25.80	38.20	27.45	15.85	13.12	0	12.45	10.97	12.91	14.34	25.33
J	41.82	40.10	31.78	38.24	50.11	38.72	28.30	25.29	12.45	0	7.99	19.88	18.36	28.49
K	47.12	43.23	33.07	35.30	44.87	32.94	25.43	21.26	10.97	7.99	0	12.75	10.55	20.51
L	51.14	43.75	31.56	26.86	33.03	20.80	18.29	12.96	12.91	19.88	12.75	0	4.25	12.96
M	53.07	46.57	34.73	31.08	36.95	24.66	22.31	17.05	14.34	18.36	10.55	4.25	0	11.13
N	63.96	56.67	44.35	36.85	37.51	25.81	29.89	24.56	25.33	28.49	20.51	12.96	11.13	0

Table 3.2 Euclidean distance between each fourteen domains.

B. Graph Modelling

From the data that has been obtained, a resulting weighted graph that represents domains locations and its distance with one another is a complete graph which is a K_{14} (*n,* the number of vertices are based on the number of domains existing in Mondstadt), as shown in Figure 3.4.

C. Held-Karp Algorithm Implementation

Now that the distances have been obtained, the Held-Karp Algorithm can be executed. Based on the geographical landscape of Mondstadt, the nearest domain from the Mondstadt city is domain *I* (Temple of The Falcon), being located near the city's gate.. Therefore, it is more precise to assume that the starting vertex begins at domain *I,* since at the beginning of the game (the prologue of Genshin Impact) players are strictly directed to the center of Mondstadt city before gaining the permission to access further exploration of the nation.

As already been discussed in the Theoretical Foundation, the graph implementation considered in this study is a Hamiltonian path rather than a Hamiltonian cycle. This is due to the free will of the players to teleport back to the initial domain (domain *I*) right after reaching the final domain.

To automate the optimization process, the Held-Karp algorithm is written in a Python program as shown below.

```
from distance import dist_matrix, labels, n

from itertools import combinations



start_vertex = labels.index('I')



def held_karp(edge_weight, num_vertices, start_vertex):

    min_tour_cost = {}

    predecessor = {}

    initial_set = frozenset([start_vertex])

    min_tour_cost[(initial_set, start_vertex)] = 0

    other_vertices = [v for v in range(num_vertices) if v !=
     start_vertex]
```

The program lines above are the initialization of the overall program. The min_tour_cost variable stores the computed values of *T(S,d)*, while *predecessor* records the predecessor destination *i* for each state, which will later be used to recalculate the optimal path. The base case corresponds to *T({I},I)* = 0, since no distance has been traveled when only the starting domain or vertices that have been visited.

```
    for subset_size in range(1, len(other_vertices) + 1):

        for vertex_combo in combinations(other_vertices,
         subset_size):
```

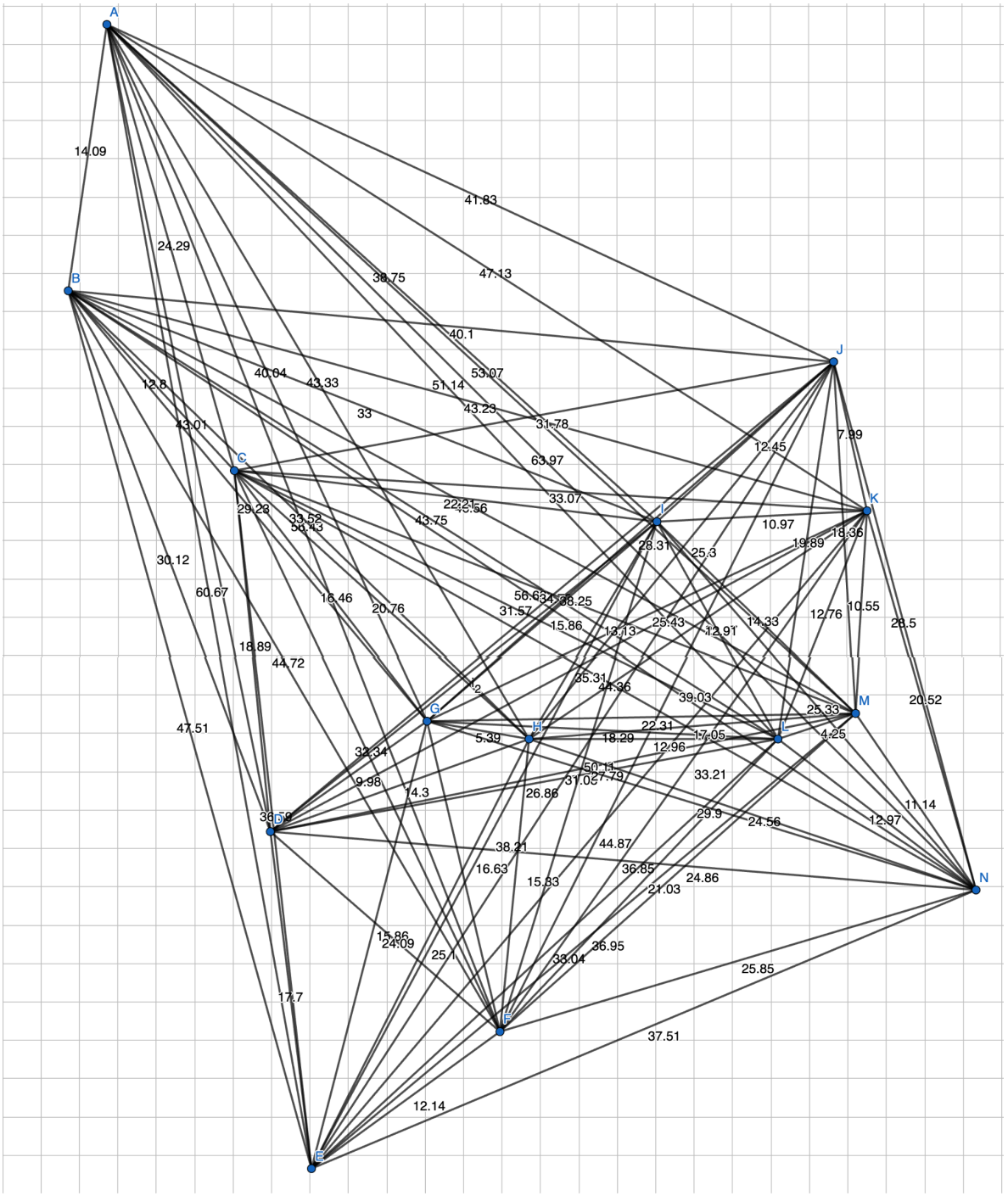

Figure 3.4 The resulting weighted graph
Source : Author's documentation, 2026.

```
            visited_set = frozenset(vertex_combo) |
            {start_vertex}

            for current_vertex in vertex_combo:

                previous_visited_set = visited_set -
            {current_vertex}

                best_cost = None

                best_predecessor = None

                predecessor_candidates = [start_vertex] + [

                v for v in vertex_combo if v !=
                current_vertex]

                for candidate in predecessor_candidates:

                    if (previous_visited_set, candidate) not
                     in min_tour_cost:

                        continue

                    cost_so_far =    min_tour_cost
                    [(previous_visited_set,candidate)]

                    edge_cost =
                     edge_weight[candidate][current_vertex]

                    total_cost = cost_so_far + edge_cost

                    if best_cost is None or total_cost <
                     best_cost:

                        best_cost = total_cost

                        best_predecessor = candidate

                if best_predecessor is not None:

                    min_tour_cost[(visited_set,
                     current_vertex)] = best_cost

                    predecessor[(visited_set,
                     current_vertex)] = best_predecessor
```

This block implements the recursive relation $T(S,d)=\min[T(S-\{d\},i)+w(i,d)]$ directly. The variable *visited_set* corresponds to S, *current_vertex* corresponds to d, and the *candidate* variable iterates over every possible predecessor i. As illustrated in the four destination example in the Theoretical Foundation, computing a state such as $T(\{A,B,C,D\},D)$ requires evaluating previously computed states such as $T(\{A,B,C\},B)$, and $T(\{A,B,C\},C)$ selecting the *predecessor* that yields the minimum total cost. The same principle is applied here across all fourteen domains, with the combinations function systematically generating every subset S of increasing size, ensuring that smaller subproblems are always solved before the larger ones that depend on them.

```
    all_vertices = frozenset(range(num_vertices))

    best_tour_cost = None
    best_final_vertex = None

    for final_vertex in other_vertices:

        if (all_vertices, final_vertex) not in min_tour_cost:

            continue

        total_tour_cost = min_tour_cost[(all_vertices,
         final_vertex)]

        if best_tour_cost is None or total_tour_cost <
         best_tour_cost:

            best_tour_cost = total_tour_cost

            best_final_vertex = final_vertex
```

After all subsets have been processed, the algorithm determines the optimal final destination by comparing $T(V,d)$ across all candidate vertices d, where V denotes the complete set of fourteen domains. As this study models the problem as a Hamiltonian path rather than a Hamiltonian cycle, no return cost to the start vertex is added at this stage.

```
    tour = []

    current_visited_set = all_vertices

    current_vertex = best_final_vertex

    while current_vertex != start_vertex:

        tour.append(current_vertex)

        prev_vertex = predecessor[(current_visited_set,
         current_vertex)]

        current_visited_set = current_visited_set -
        {current_vertex}

        current_vertex = prev_vertex

    tour.append(start_vertex)

    tour.reverse()

    return best_tour_cost, tour


tour_cost,    tour_vertices    =    held_karp(dist_matrix,    n,
start_vertex)

tour_labels = [labels[v] for v in tour_vertices]


print("Result : ")

print(f"Start vertex: {labels[start_vertex]}")

print(f"Optimal path: {' -> '.join(tour_labels)}")

print(f"Total path cost (distance): {tour_cost:.2f}")
```

Since the dynamic programming procedure only yields the minimum cost rather than the corresponding sequence of destinations, the path is reconstructed by tracing the predecessor dictionary backward from best_final_vertex until the start vertex is reached, after which the resulting sequence is reversed to reflect the correct travel order.

IV. Result

The shortest path obtained by implementing the Held-Karp algorithm is sequentially by visiting vertices in the following order: *I, J, K, M, N, L, H, G, F, E, D, C, B, A*. By mapping the vertices to the actual domain's name, the most optimal route begins at Temple of The Falcon, then by visiting Midsummer Courtyard, Temple of The Wolf, Unresolved Chess Game, Eagle's Gate, Temple of The Lion, Forsaken Rift, Valley of Remembrance, Peak of Vindagnyr, Hidden Place of Lianshan Formula, Ridge Watch, Cecilia Garden, Confront Stormterror, and finally ending at Thorny Crown of The Mountain Wild. The total distance of the optimal route is at exact 165.46 units. Figure 4.2 illustrates the resulting weighted graph and the resulting edges that form the TSP route have been highlighted in orange.

```
Result :
Start vertex: I
Optimal path: I -> J -> K -> M -> N -> L -> H -> G -> F -> E -> D -> C -> B -> A
Total path cost (distance): 165.46

[Done] exited with code=0 in 0.218 seconds
```

Figure 4.1 Result of the program execution.
Source : Author's documentation.

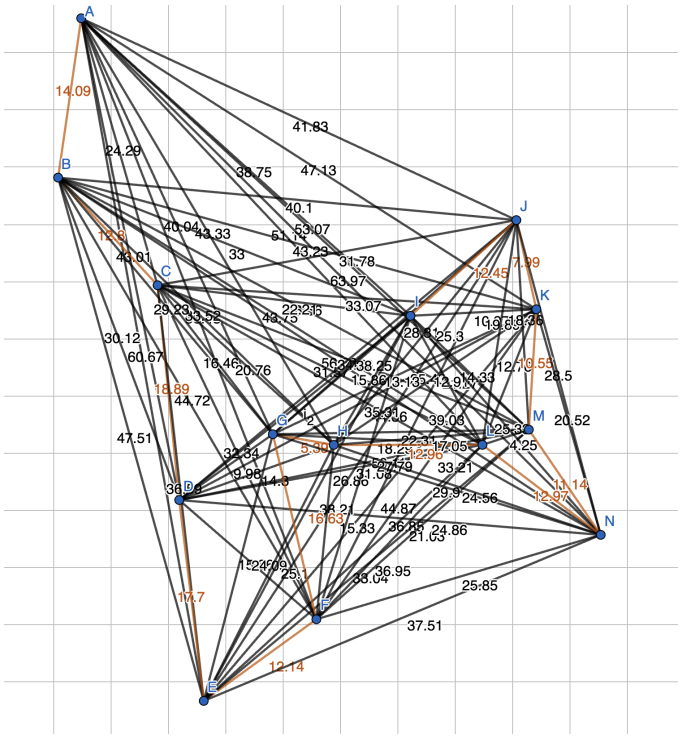

Figure 4.2 Shortest route obtained by the Held-Karp algorithm for Mondstadt's domain exploration.
Source : Author's documentation.

V. Conclusion

In conclusion, this study has successfully creates a model of Genshin Impact Mondstadt's map in a weighted graph in order to find the most optimal route to explore domains in Mondstadt. It is already proven that it can be obtained by implementing The Salesman Problem (TSP) approach with the help of the Held-Karp algorithm.

These findings demonstrate that mathematical concepts, especially about graph theory, could help in finding solutions for open-world games like Genshin Impact especially in a matter of time efficiency.

For further development of this study, the author recommend some of these points that can be used for future use on the same matter, which are

- The considerations of the geographical Mondstadt landscape in details that may be an obstacle, such as cliffs, mountains, or water ecosystem that may obstruct the accessibility of the players to reach the domains, and
- The impact of the player's chosen *main character* (MC) which is either Aether or Lumine to the time spent to visit all domains.

Appendix

The programs used in this paper can be fully accessed on the following GitHub link GitHub Link

Paper explanation video could be accessed on the following link Youtube Link

Acknowledgment

The author would like to express sincere gratitude to everyone who has provided support and assistance for the author in the process of writing this paper. First of all, by thanking Allah, The Almighty God for giving the writer a healthy body and mind so this paper could be finished just in time. Of course, the author would also want to send the gratitude for Prof. Dr. Ir. Rinaldi Munir, M.T., as the author's lecturer in IF1220 Discrete Mathematics in STEI ITB, because without his lectures, the author of this paper would not get this far in understanding the concepts needed to write this paper and overall discrete mathematics concepts to keep on pursuing the author's degree in ITB. Finally, the author is also grateful to the family and friends accompany during the process of this paper.

References

[1] Munir, Rinaldi. Graf (Bagian 1). Informatika STEI ITB 2026. [Online]. Available in https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/20-Graf-Bagian1-2026.pdf . Accessed on June 16, 2026.

[2] Munir, Rinaldi. Graf (Bagian 3). Informatika STEI ITB 2026. [Online]. Available in https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/22-Graf-Bagian3-2026.pdf Accessed on June 16, 2026.

[3] Kenneth H. R. (2019). Discrete Mathematics and Its Applications Eighth Edition. McGraw-Hill Education.

[4] Munir, Rinaldi. Graf (Bagian 2). Informatika STEI ITB 2026. [Online]. Available in https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/21-Graf-Bagian2-2026.pdf Accessed on June 16, 2026.

[5] Hoyolab. Teyvat Interactive Map. [Online]. https://act.hoyolab.com/ys/app/interactive-map/index.html?lang=en-us#/map/ Accessed on June 17, 2026.

[6] GeeksforGeeks. Euclidean Distance. [Online]. Available in https://www.geeksforgeeks.org/maths/euclidean-distance/ Accessed on June 17, 2026.

Bandung, June 18, 2026

Three Gie Gendhis Sekar Ayoe Jatmiko
13525144

STATEMENT

I hereby declare this paper is the original work of the author, which means that the paper is not an adaptation nor a translation of other existing papers and not a plagiarism.