HUTAO-TSP: A Custom Traveling Salesman Problem for Approximating the Optimal Route for Elemental Oculus Collection in Genshin Impact Using Graph Theory

Yavie Azka Putra Araly - 13524077
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: yavieazkaputra@gmail.com , 13524077@std.stei.itb.ac.id

***Abstract*—HUTAO-TSP (*Hamiltonian Undirected Teleportation Aware Open Travelling Salesman Problem) is a* custom graph traversal algorithm designed for scenarios in which each node (e.g., elemental oculus in *Genshin Impact*) must be visited exactly once in an undirected graph, while allowing the traveler to utilize zero-weighted teleportation points accessible from any location in the graph. This algorithm approximates an optimal solution by combining heuristics from the travelling salesman problem, Hamiltonian Path theory, and teleportation-aware path planning, producing efficient traversal routes in a graph augmented with fast-travel edges. We propose a domain-specific heuristic for a variant of the Travelling Salesman Problem (TSP), motivated by the structure of teleportation-enabled traversal in Genshin Impact. While our method does not guarantee optimality, it leverages teleport waypoint proximity to produce lower-cost paths.**

***Keywords*— Approximates, Elemental Oculus, Hamiltonian Path, Teleportation, Travelling Salesman Problem.**

I. Introduction

Genshin Impact is an open-world action role-playing game developed and released by HoYoverse in 2020. The game features an anime-style open world environment and an action-based combat system using elemental reactions and character-switching [1]. This game is set in *Teyvat*, home of seven nations, each represented by 7 different elements and ruled by seven different gods known as *Archons.* The story follows the Traveler — the player character — who, at the start of the game, is separated from their twin sibling after they land in *Teyvat*. Accompanied by Paimon, their guide along their adventure visiting each country in *Teyvat,* the Traveler becomes involved in many conflicts and tries their best to help the people resolve them.

Figure 1. Genshin Impact official poster for the 5.7 version update. Source: https://genshin.hoyoverse.com/

As of the time this paper is written, Genshin Impact has already introduced six countries in *Teyvat* that represent different elements released, which are Mondstadt (*anemo*/wind), Liyue (*geo*/earth), Inazuma (electro/lightning), Sumeru (*dendro*/tree or plants), Fontaine (*hydro*/water), and Natlan (*pyro*/fire). The developer is also working on the next country that represents the 7th element, which is Snezhnaya (*cryo*/ice). Each country in *Teyvat* is also inspired by real-world countries. Mondstadt from Germany, Liyue from China, Inazuma from Japan, Sumeru from the Middle East, Fontaine from France, Natlan from Africa, and Snezhnaya from Russia.

One of the most important aspects of Genshin Impact gameplay is exploring *Teyvat*. In *Teyvat*, players can unlock *Teleport Waypoints,* which allow players to fast travel to the teleportation device that is spread across *Teyvat* from any point to make it easier for players to explore. Players have to collect many items spread across every region while also defeating the monsters. One of the most important items in the game is the Elemental Oculus. This item is crucial for the player's progress. Collecting Elemental Oculus will give players rewards and also increase their stamina for easier exploration and longer sprint and climbing durations.

This paper aims to approximate the most efficient way to collect the Elemental Oculi using a method called HUTAO-

TSP (*Hamiltonian Undirected Teleportation Aware Open Travelling Salesman Problem)*. This method is a combination of Hamiltonian Path, graph theory, and the Travelling Salesman Problem. This method also adds variation to the graph by adding teleportation devices, in this case, Teleportation Waypoints in *Teyvat*.

II. Theoretical Framework

A. Graph

Graph G is defined as a G = (V, E), where V is a non-empty set of vertices and E is an edge set that connects a pair of vertices. There are various types of graphs, such as:

1. Directed & Undirected Graph: A directed graph is A graph that has orientation in its edges, whereas an undirected graph is the opposite.

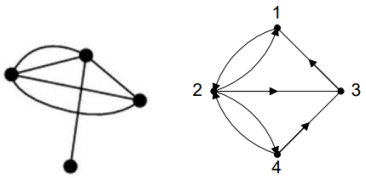

Fig. 2: Undirected graph (left) and directed graph (right). Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

2. Weighted Graph & Unweighted Graph: A weighted graph is a graph that weights every edge on the graph, whereas an unweighted graph is the opposite.

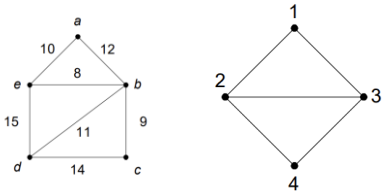

Fig. 3: Weighted graph (left) and unweighted graph (right). Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

3. Simple Graph: A simple graph is a graph that does not contain loops or multiple edges.

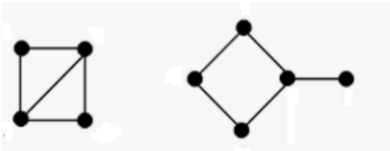

Fig. 4: Simple graph. Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

4. Complete Graph: A complete graph is a graph in which each vertex has an edge to all other vertices.

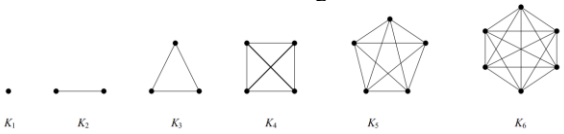

Fig. 5 Undirected graph (left) and directed graph (right). Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

B. Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a classical problem in graph theory. The problem challenge is to find the fastest route for a salesman to visit all of the cities exactly once and return to the first city. In this paper, the "salesman" refers to the Traveler, and the "cities" represent the Elemental Oculus [2].

TSP is an NP-hard problem, meaning that no known algorithm can solve it efficiently in polynomial time for large inputs. It requires exponential time and memory to find the exact solution. There are many approaches for solving TSP, including brute force, greedy, dynamic programming, and heuristic methods. But as stated before, these algorithms have exponential time complexity. Take a look at this example graph:

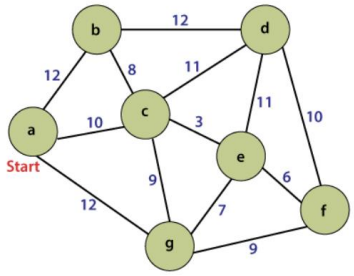

Fig. 6: Weighted Graph for TSP. Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf

This is an example of the Traveling Salesman Problem. The goal of the problem is to find the minimum weight for visiting all nodes and returning to the starting node. The brute force approach tried every possible path and calculated their total distances, resulting in a time complexity of $O(n!)$. On the other side, the dynamic programming approach, such as the Held-Karp Algorithm, has $O(n^2 * 2^n)$ Time complexity. For example, finding the route for collecting 30 Elemental Oculus by using brute force takes $50! = 9.3 \times 10^{157}$ times of computation for the computer to solve it, and $50^2 * 2^{50} = 2{,}814{,}749{,}767{,}106{,}560{,}000$ Time of computation for the computer to solve it using the Held-Karp Algorithm. Assuming a computer could handle 1,000,000,000 computations per second, brute forcing will take an incredibly long time, more than the age of the universe. On the other hand, the Held-Karp Algorithm will take around. 32.578,12 Days, or roughly 89 years, to find the solution.

After knowing that computing every possible path is nearly impossible using modern computing technology, we can still use approximation algorithms to significantly decrease the time of computation, yet still give a reasonable solution. We can apply approximation algorithms—such as those based on the Minimum Spanning Tree (MST),—to obtain a reasonable

answer without taking years to find the exact solution. In this paper, we will use the Minimum Spanning Tree to approximate the most efficient solution for the graph.

C. Minimum Spanning Tree

A spanning tree is a connected, undirected subgraph that includes all the vertices of the original graph. A Minimum Spanning Tree (MST) is a spanning tree of a graph with the minimum possible weight among all possible spanning trees of a graph. There can be multiple MSTs for a graph if different spanning trees yield the same minimum total weight.

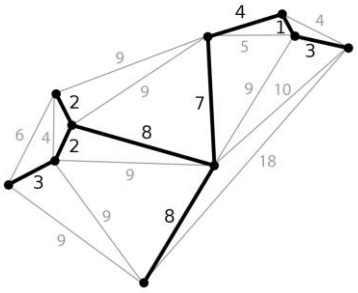

Fig 7: Minimum Spanning Tree from a weighted graph. Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf

There are several algorithms to find the MST for a given graph, including Kruskal's Algorithm, Prim's Algorithm, and Boruvka's Algorithm. Many problems in different topics can be solved using MST, such as network design, image processing, biology, and social network analysis. In this paper, we will use Prim's Algorithm to find the MST that connects all Elemental Oculi in the map.

D. Hamiltonian Path and Hamiltonian Circuit

A Hamiltonian Path is a path that visits every vertex in a graph exactly once without returning to the starting vertex [2]. A Hamiltonian Circuit is a circuit in a graph that visits every vertex exactly once and returns to the starting vertex. The difference between the Hamiltonian Cycle and Traveling Salesman Problem lies in their goals: the Hamiltonian Cycle problem asks whether such a cycle exists, while TSP seeks the cycle with the minimum total weight that visits each vertex exactly once. Therefore, TSP will be used to find the minimum weight of a Hamiltonian Circuit.

E. Depth First Search

Depth First Search is of graph traversal algorithm that we travel all adjacent vertices one by one. After there are no nodes to visit left, we go back to the previous branch that has at least an adjacent node that has not been visited yet. This Algorithm is similar to a tree, where we first completely traverse a subtree and then move to the next subtree. The key difference is that DFS allows traversal within a graph that has cycles [3].

F. Elemental Oculus

Fig. 8: Elemental Oculus represents each element. (From left to right: Pyroculus, Geoculus, Dendroculus, Anemoculus, Hydroculus, Electroculus). Source: https://x.com/Archon7gnosis/status/1813540629320007739

Oculi in Genshin Impact are collectible items that can be found across *Teyvat*. They are scattered throughout the world and are used to level up their corresponding *Statue of the Seven* [4]. Upgrading the *Statue of the Seven* grants players various rewards, such as *Mora* (the in-game currency), *primogems* (a premium currency used for character and weapon wishes), and stamina. Increased stamina allows players to walk, climb, and swim for longer durations, which significantly eases exploration, especially for new players.

Currently, there are 6 types of Elemental Oculus in Genshin Impact. Each type represents its region and element. *Anemoculus, Geoculus, Electroculus, Dendroculus, Hydroculus, and Pyroculus* have already been added to the game; each can be found in their respective region. Finding Elemental Oculus needs time, effort, and patience. Here's the data about the number of Elemental Oculus in each region:

Table 1: Number of Elemental Oculus in Genshin Impact at 5.0 version. Source: https://genshin-impact.fandom.com/wiki/Oculus

Region	Oculi
Mondstadt	66
Liyue	131
Inazuma	181
Sumeru	271
Fontaine	271
Natlan	222

There are already in total of 1.142 Elemental Oculus in the game that need to be found across the map. In May 2021, Hoyoverse, Genshin Impact, released the *Teyvat Interactive Map*.

This interactive map allows players to track every item in the game and find their location on the map. But sadly, the map does not integrate with the game. It means that players have to open the map while playing the game. But this interactive map is really helpful for players who do not have much time exploring the map. For example, one of the subregions in Fontaine, New Fontaine Research Institute, has around 27 Hydroculus in its area. Considering the Teyvat map is already really wide and having thousands of Elemental Oculus (and the number is increasing), finding a way to collect them efficiently is quite a helpful idea.

Fig. 9: Teyvat Interactive Map. Source: https://act.hoyolab.com/ys/app/interactive-map

G. Teleport Waypoints and Statue of The Seven

Fig. 10: Statue of the Seven in Fontaine (left) and Teleport Waypoint (right). Source: in-game screenshot by author.

Teleport Waypoints are structures that players can find and unlock across *Teyvat*. Statues of the Seven are structures found all around *Teyvat,* representing their country by their corresponding *Archon*. The difference between Teleport Waypoints and Statues of the Seven is in their function besides teleporting. Unlike the Teleport Waypoints, Statues of The Seven have more purposes than just as a teleportation device. They can unlock areas, heal and revive fallen characters, change the Traveler's elemental power depending on which statues the Traveler interacts with, and also offer rewards in exchange for oculi.

III. Elemental Oculus Distribution Modelling

A. Getting Coordinates for Each Elemental Oculus

The first thing we have to do for modeling Elemental Oculus distribution is to get the data of the coordinates for every Elemental Oculus. Let's take a look at the previous example, the Elemental Oculus distribution in New Fontaine Research Institute, one of the subregions in Fontaine.

Fig. 11: Distribution of Hydroculus in New Fontaine Research Institute. Source: https://act.hoyolab.com/ys/app/interactive-map/index.html

As mentioned before, there are in total of 27 Hydroculus in the subregions, 9 teleport waypoints, and 1 Statue of the Seven. There are many ways to mark their location. In this paper, we use graphic design software, such as Canva, to mark the object's location. By marking every object, we get a group of vertices, as shown in the figure below:

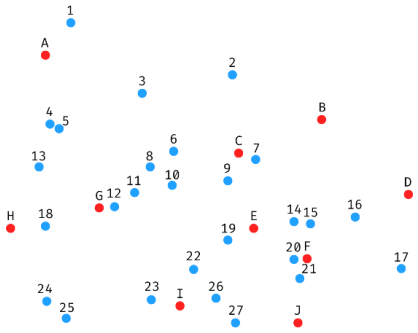

Fig. 12: Elemental Oculus (marked with blue color) and Teleport Waypoints distribution (marked with red color). Source: author.

After marking the object's location, we have to get its coordinates. There are many tools available to do this. In this paper, we are using Python with the matplotlib library to let us get the coordinates for each object.

```
import matplotlib.pyplot as plt

img = plt.imread('marked_node.png')
coords = []

def onclick(event):
    x, y = int(event.xdata), int(event.ydata)
    coords.append((x, y))
    print(f"Clicked: ({x}, {y})")
    if len(coords) == 37:
        plt.close()

plt.imshow(img)
plt.title("Click on the nodes (order matters)")
cid = plt.gcf().canvas.mpl_connect('button_press_event', onclick)
plt.show()

# Save coordinates
with open("node_coordinates.txt", "w") as f:
    for i, (x, y) in enumerate(coords):
        f.write(f"{i},{x},{y}\n")
```

Source code 1: Python source code for getting every object's coordinates by clicking it. Source: author.

This code allows us to get the coordinates for every object in the map by simply clicking on it. After running the program and clicking every object on the figure, we get the coordinates for each Elemental Oculus and Teleport Waypoints (Statue of the Seven is categorized as Teleport Waypoints) as follows:

Table 2: Hydroculus and Teleport Waypoints coordinates from Figure 7. Idx 0 – 26 are Elemental Oculus, Idx 27 – 36 are Teleport Waypoints. Source: Author

Idx	x	y	Idx	x	y	Idx	x	y
0	211	63	13	868	644	26	698	939
1	685	219	14	916	652	27	140	161
2	421	269	15	1047	631	28	954	353

3	157	361	16	1184	781	29	704	446
4	178	371	17	136	656	30	1203	569
5	513	440	18	677	702	31	752	665
6	754	463	19	868	758	32	902	758
7	444	486	20	881	816	33	296	606
8	673	525	21	571	783	34	36	667
9	504	540	22	450	875	35	533	893
10	396	567	23	147	877	36	879	948
11	344	606	24	196	931			
12	117	486	25	635	868			

B. Connecting every Elemental Oculus and Teleport Waypoints with a Minimum Spanning Tree

As mentioned before, Minimum Spanning Tree (MST) is a connected, undirected subgraph that includes all the vertices of the original graph. Using MST can give us the minimum weight to connect every node in a graph with minimum weight. There are several algorithms to find the MST from a graph, such as Prim's Algorithm and Kruskal's Algorithm. In this paper, we are using Prim's Algorithm.

Prim's Algorithm is used for finding the MST for a graph. It starts with an empty spanning tree. The algorithm works as follows:

Step 1: Choose a minimum edge from graph G and insert it into a set T.

Step 2: Choose an edge (u, v) such that (u, v) is adjacent to T, but does not form a circuit with T.

Step 3: Repeat steps 1 and 2 for (n–2) times [2].

This algorithm has *O(n*n)* time complexity and is guaranteed to find the MST from a connected, weighted graph. This algorithm is also efficient for finding the MST from a sparse graph, like the Elemental Oculus distribution.

We can also use Python code for finding the MST from a graph automatically by using network, numpy, and matplotlib libraries.

```
# Compute MST for EO nodes using Prim's algorithm
mst = nx.minimum_spanning_tree(G.subgraph
        (range(len(eo_coords))),
        weight='weight', algorithm='prim')

# Plot all nodes, highlight EO vs TW
pos = nx.get_node_attributes(G, 'pos')
node_types = nx.get_node_attributes(G, 'type')

# Split EO and TW for color coding
eo_nodes = [n for n in G.nodes if node_types[n] == 'eo']
tw_nodes = [n for n in G.nodes if node_types[n] == 'tw']

plt.figure(figsize=(12, 12))

# Draw EO and TW nodes, and MST edges
nx.draw_networkx_nodes(G, pos, nodelist=eo_nodes, node_color='skyblue',
label='Elemental Oculus')
nx.draw_networkx_nodes(G,            pos,            nodelist=tw_nodes,
node_color='mediumpurple', label='Teleport Waypoint')
nx.draw_networkx_edges(mst, pos, edge_color='black')
```

Source code 2: Python code for finding the MST from a graph. Source: Author.

After running the program, we get the MST as follows:

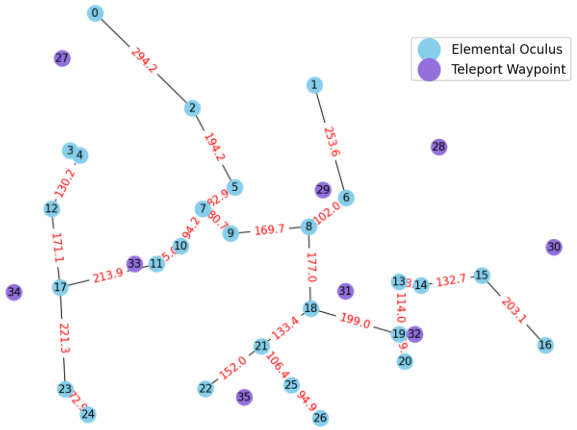

Fig. 13: MST generated from Elemental Oculus distribution in Figure 7.

Table 3: Some not visible nodes' weight.

X	Y	Distance(X,Y)
3	4	23.26
11	10	65.00
7	9	80.72
7	5	82.93
23	24	72.92
19	20	59.44
13	14	48.66

By using this MST, we will determine the most efficient way to visit all nodes by also utilizing the Teleport Waypoint.

C. Connecting Teleport Waypoints to MST

As mentioned before, we have already made the MST for Elemental Oculus distribution. After that, we can connect the Teleport Waypoints to every node in the graph.

As the name "Teleport Waypoints" suggests, this structure allows players to teleport to it without walking from every place in the map. This also occurs in our graph, making it a directed path that connects to all 27 nodes with 0 weight. But since it is obvious that every node is connected with every Teleport Waypoint, we don't need to draw the edge. Instead, we need to compute the distance from a Teleport Waypoint to some nodes, because we will utilize these teleport waypoints to leverage the cost for visiting every node.

IV. Problem Formulation and HUTAO-TSP Concepts

A. Problem Statement

HUTAO-TSP is used to approximate the most efficient way to visit all nodes with a new feature, Teleport Waypoint. In this paper, we will use it to collect every Elemental Oculus by just visiting them once. Even though this approach looks like it only works with a teleportation-aware graph, generally, for any TSP problem, HUTAO-TSP can still be used to leverage the traveling cost for visiting every node in the graph.

First, we need to find the MST of a graph. We already did it before. We can see the MST that we made as a binary tree, since every node has a maximum of 3 degrees. After forming the binary tree, there are 2 steps in HUTAO-TSP —As the

name Hamiltonian Undirected Teleportation Aware Open TSP—Creating a Hamiltonian path and utilizing Teleportation Waypoints.

B. Creating Hamiltonian Path

Before creating the Hamiltonian Path, we need to find the initial traversal path for the MST. As shown in Figure 7, every node in the MST has a maximum of 3 degrees. We also don't want to visit an edge twice, making our MST a binary tree. We can create a binary tree based on the MST with Python.

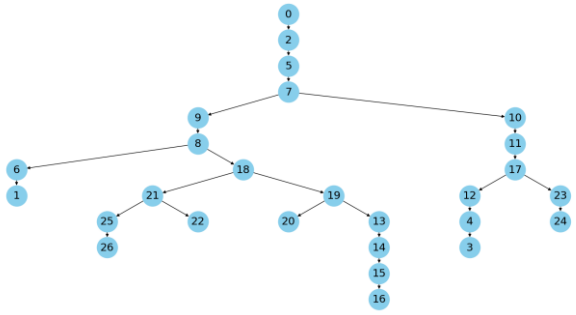

Fig. 14: Binary tree formed from the MST in Figure 7. Source: Author

But as we know, traversing a binary tree is not efficient because it does many backtracks. Let's take an example with traversing the binary tree we have made in Figure 8 with pre-order. First, we start at the root node, 0. We can travel the tree naturally until reaching node 1. The path will be [0 – 2 – 5 7 – 9 – 8 – 6 – 1]. But after reaching node 1, we need to do backtracking to nodes 6 and 8, and continue to node 18. This is not an efficient way to visit every node.

This problem can be solved by creating a Hamiltonian Path from the binary tree by adding some edges to the tree. We know creating a Hamiltonian Path from a tree will make the tree no longer a tree, that's why this added edge is called a 'helper edge'. For every leaf in the MST, we can create a directed edge called a 'helper edge' that allows a heuristic jump from a leaf to another node. This will create a Hamiltonian Path in the MST and allow us to go to other nodes without doing backtracking.

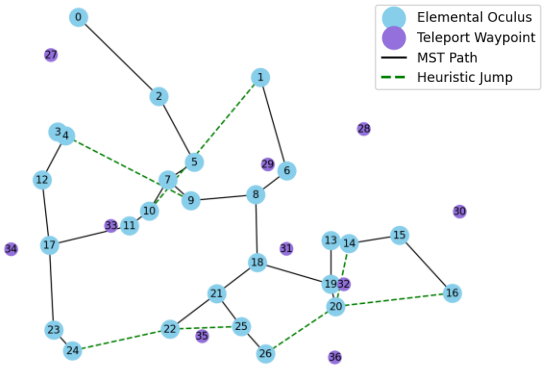

Fig. 15: Example of adding a helper edge from the MST. Source: Author.

The picture above is an example of using a helper edge that allows a heuristic jump from a leaf to another node. The idea is to find the closest unvisited node to be visited next. That's why the helper edge will not always look like above, since it looks for the closest unvisited node to connect with the current leaf. As we travel the tree, every time we reach a leaf, the distance from the previous root of a subtree will be calculated and compared to the heuristic jump distance. If the heuristic jump gives less weight, we will choose to jump rather than do backtracking. For example, if we travel the tree in order, starting from node 0, the path will be [0 – 2 – 5– 7 – 9 – 8 – 6 – 1]. Node 1 is a leaf. So, we need to do backtracking from node 1 to node 8 and continue to node 18. The path will be [1 – 6 – 8 – 18]. This takes 253.6 + 102.0 + 177.0 = 532.6 weight to reach the next unvisited node. On the other side, using a helper edge will shorten the path. From node 1, the closest unvisited node is 10 with a distance of 452.35. This will make travelling the tree will be more efficient.

C. Utilizing Teleport Waypoints

Teleport waypoints are also crucial in HUTAO-TSP, since T and A in HUTAO-TSP stand for "Travelling Aware". Teleport waypoints are used to significantly increase the efficiency of collecting Elemental Oculus. As mentioned before, players are allowed to teleport to every Teleport Waypoint from every place in the map.

Teleport waypoints can be used to avoid backtracking. For example, let's take a look back at the tree. If we travel the tree with pre-order, the path will be [0 – 2 – 5– 7 – 9 – 8 – 6 – 1]. Note that node 8 is the root of a subtree with 9 and 18 are its children. Using pre-order traversal will always make us travel the left child first, which is 8. But this path is not efficient. As we know, there is a teleport waypoint 29 that has less distance to node 6 rather than from node 8 itself. On the other hand, visiting node 18 directly from node 8 is also not efficient, since we can visit 18 from teleport waypoint 31. That's why teleport waypoints are really important here.

V. The HUTAO-TSP Algorithm: A Two-Phase Heuristic Framework

The core idea of how HUTAO-TSP works lies in how it determines the visitation sequence. As we know, there are several tree and graph traversal algorithms. In HUTAO-TSP, we employ a modified DFS to choose which branch to visit next, called "Branch Prioritization". This key heuristic of HUTAO-TSP is applied at any junction (a node with multiple unvisited child branches) to decide the order of exploration.

The algorithm is structured into two primary phases:

1. Phase 1: Optimal sequence determination

In this phase, HUTAO-TSP will do traversal within the MST that has been made before using modified DFS. This phase prioritizes the global structure of the route.

2. Phase 2: Final path construction

Using the sequence from phase 1 as a blueprint, a detailed and more efficient path is constructed in this phase by making a discrete "Teleport-or-Walk" decision each time we try to reach another node.

A. *Formal problem statement and MST construction*

Given a complete, weighted, and undirected graph G = (V, E), where the vertex set V is partitioned into Elemental Oculus (EO) nodes and Teleport Waypoint (TW) nodes. Each EO and TW has its coordinates, making the edge set E contains all pairs of EO nodes weighted by their Euclidean Distance. Creating an MST will give an efficient structural backbone for traversal, denoted `T_mst` (Fig. 7).

B. *Phase 1: Sequence determination via Teleportation-Aware DFS*

This phase will find an `oculus_visit_sequence,` which will be the path for collecting every Elemental Oculus using a modified DFS algorithm.

- **Input**
  An MST, in this case `T_mst, is` rooted at the designated `source_node`. This can be achieved by creating a binary tree (Fig. 8) or simply by choosing a leaf from `T_mst`.
- **Data structures**
  - A stack for managing DFS traversal, initially filled with `source_node.`
  - A `visited_oculi` set to save Oculus that has been collected
  - An `oculus_visit_sequence` list as the output. This will give the path for visiting every Elemental Oculus, without utilizing the Teleport Waypoints.
- **Algorithm**

1. While the stack is not empty, pop a node, designated `current_node`.
2. If `current_node` is visited, which means it is in `visited_oculi`, continue to next iteration
3. Add `current_node` to `visited_oculi` and `oculus_visit_sequence`, `current_node` will not be visited next.
4. Find every unvisited child of `current_node`.
5. Junction Heuristic: If `current_node` has more than 1 child, apply the "Branch Prioritization" Heuristic:
   a. For each child, calculate its teleport inefficiency score. This score is defined as the mean Euclidean distance from every node in that child's respective subtree to its nearest Teleport Waypoint. A higher score signifies greater geographic isolation from the teleport network.
   b. Sort the children in descending order based on this score.
6. Push the sorted children onto the stack. Due to the LIFO nature of the stack, the branch with the highest inefficiency score is processed next.

- **Output:** The `oculus_visit_sequence` list, which represents the strategic order of visitation.

C. *Phase 2: Final Path Construction with Teleport-or-Walk Decisions*

This final phase constructs the explicit travel path, including waypoints, and calculates the total traversal cost. This will also give a step-by-step exploration guide to visit the next Elemental Oculus by directly walking toward it or using nearby Teleport Waypoints.

- **Input:**
  - The `oculus_visit_sequence`, which has been created from phase 1.
  - `T_mst` for teleport waypoint mapping.
  - Pre-computed distance between each Elemental Oculus and its nearest Teleport Waypoint.
- **Data Structures:**
  - A `final_path` list, initialized with the first node from the sequence.
  - A `total_cost` variable, initialized to 0.0.
- **Algorithm:**

1. Iterate through the `oculus_visit_sequence` from the first to the second-to-last node. Each pair in the iteration represents a travel segment from a `from_node` to a `to_node`.
2. For each segment, perform a Teleport-or-Walk Decision:
   a. Calculate `walk_cost`: The direct edge weight between `from_node` and `to_node` in G.
   b. Calculate `teleport_cost`: This is the cost to instantly teleport from the player's current location to the Teleport Waypoints nearest the destination, and then walk to the destination. This cost is simply the pre-computed distance from to_node to its nearest Teleport Waypoints (`nearest_tw_data[to_node]['dist']`).
3. Path Assembly:
   a. If `walk_cost` ≤ `teleport_cost`, the direct walk is chosen. `walk_cost` is added to `total_cost`, and `to_node` is appended to `final_path`. The instruction to reach the next node is to walk.

- Otherwise, teleportation is chosen. `teleport_cost` is added to `total_cost`.
- **Output:** The `final_path`, a complete, actionable route containing both Elemental Oculus and Teleport Waypoints nodes, and the `total_cost` of this route.

```
# ----- HUTAO - TSP Final Code ----- #

# Step 1: Create MST (source code 2)
# Step 2: Pre-compute the distance between every node with its closest
teleport waypoint using Euclidean distance
# --- STEP 3: PHASE 1 - DETERMINE OCULUS VISIT SEQUENCE VIA DFS ---
print("--- Phase 1: Determining Optimal Oculus Sequence via Teleport-
Aware DFS ---")

# Pre-computation for efficiency
nearest_tw_data = precompute_nearest_tw_data(G, len(eo_coords),
len(tw_coords), tw_start_index)

# Data structures for the DFS traversal
source_node = 0
T = nx.dfs_tree(mst, source=source_node) # Directed tree for
parent/child logic
```

```
stack = [source_node]
visited_oculi = set()
oculus_visit_sequence = [] # The ordered list of Oculi to visit

While stack:
    current_node = stack.pop()
    if current_node in visited_oculi:
        continue

    visited_oculi.add(current_node)
    oculus_visit_sequence.append(current_node)

    children = [child for child in T.neighbors(current_node) if child
not in visited_oculi]
    If children:
        if len(children) > 1:
            scored_children = []
            For a child among children:
                score = get_branch_teleport_score(child, T,
nearest_tw_data)
                scored_children.append((child, score))
            scored_children.sort(key=lambda x: x[1], reverse=True)
            children = [child for child, score in scored_children]

        For a child among children:
            stack.append(child)

print("Optimal Visit Sequence Found:", oculus_visit_sequence)


# --- STEP 4: PHASE 2 - BUILD FINAL PATH WITH TELEPORT DECISIONS ---
print("\n--- Phase 2: Building Final Path with Teleport-or-Walk
Decisions ---")

final_path = [oculus_visit_sequence[0]] # Start the path with the first
oculus
total_cost = 0.0
path_description = [f"Start at Node {final_path[0]}."]

# Iterate through the sequence to decide the travel method between each
pair
for i in range(len(oculus_visit_sequence) - 1):
    from_node = oculus_visit_sequence[i]
    to_node = oculus_visit_sequence[i+1]

    # Option A: Walk directly
    walk_cost = G[from_node][to_node]['weight']

    # Option B: Use the teleport network
    tw_data_from = nearest_tw_data[from_node]
    tw_data_to = nearest_tw_data[to_node]
    teleport_cost = tw_data_from['dist'] + tw_data_to['dist']

    if walk_cost <= teleport_cost:
        # Decision: Walking is better or equal
        total_cost += walk_cost
        path_description.append(f"Travel from {from_node} -> {to_node}.
Method: Walk (Cost: {walk_cost:.1f})")
        final_path.append(to_node)
    Else:
        # Decision: Teleporting is better
        total_cost += teleport_cost
        tw_from = tw_data_from['id']
        tw_to = tw_data_to['id']
        path_description.append(f"Travel from {from_node} -> {to_node}.
Method: Teleport via TW {tw_from} & {tw_to} (Cost:
{teleport_cost:.1f})")

        # Add the waypoints to the path, preventing duplicate
consecutive waypoints
        if final_path[-1] != tw_from:
            final_path.append(tw_from)
        if final_path[-1] != tw_to:
            final_path.append(tw_to)
        final_path.append(to_node)


# --- STEP 5: RESULTS AND VISUALIZATION ---
print("\n--- Traversal Complete ---")
for step in path_description:
    print(step)

print("\n--- Final Path (including Teleport Waypoints) ---")
path_with_labels = [f"TW-{n}" if n >= tw_start_index else str(n) for n

 in final_path]
print(" -> ".join(path_with_labels))
print(f"\nTotal Estimated Cost: {total_cost:.2f}")

#Draw graph
```

Source code 3: The HUTAO-TSP Algorithm. Source: Author.

We will try to approximate the most efficient path based on the MST that is shown in Fig. 7 and Fig. 8. Since we already created the MST and binary tree, we can directly jump to phase 1.

Initially, the stack is now [0]. The while loop begins. 0 is now popped from the stack and added to the sequence, making `oculus_visit_sequence` now [0]. Next, the only unvisited child of 0 is 2. Push 2 to the stack, and repeat the step above. Do the same thing with nodes 5 and 7.

When reaching node 7, this is where the critical decision starts. Pop 7 from the stack and add it to the output sequence. `oculus_visit_sequence` is now [0, 2, 5, 7]. Since node 7 has more than 1 child, the "Branch Prioritization Heuristic" will try to find which node to visit first.

a) Score Branch 9: The algorithm looks at the entire subtree connected to Node 9 (nodes 9, 8, 18, 6, 1, 21, 22, 25, 26). It calculates the average distance from all these nodes to their nearest Teleport Waypoint. This branch is large and spread out. Let's say the code calculates its Teleport Inefficiency Score to be **145.8.**

b) Score Branch 10: It does the same for the subtree at Node 10 (nodes 10, 11, 17, 12, 3, 4, 23, 24). This branch is more compact and generally closer to waypoints. Let's say its score is **98.5**.

c) The Decision**:** Since 145.8 > 98.5, the algorithm concludes that Branch 9 is the "harder," more "teleport-inefficient branch. Therefore, it must be explored first.

d) Update the Stack**:** To explore the highest-priority branch next, it gets pushed onto the stack last. The stack is now [...10, 9]. The next node to be popped will be 9.

After the algorithm finishes adding every node to `oculus_visit_sequence`, the algorithm will decide the final path for collecting every Elemental Oculus. `oculus_visit_sequence` will be [0, 2, 5, 7, …], `final_path` will be initialized by [0], and `total_cost` will be 0.0. Next, starting from node 0, the algorithm will decide to visit a node by directly walking or using the Teleport Waypoint. The nearest Teleport Waypoints from node 2 are 27 with a distance of 301.04, while direct walking costs 294.2. The algorithm will choose to walk to node 2 directly since it costs less than teleporting. `final_path` will be [0, 2] and `final_cost` will be 294.2. Next, from node 2 to node 5, there is Teleport Waypoint 29, which is the nearest Teleport Waypoint from node 5 and costs 82.93. The distance between node 2 and node 5 is 191.2, which means the algorithm will choose to do teleportation to 29, and then walk to node 5 since `teleport_cost` < `walk_cost`. The `final_path` now is [0, 2, 29, 5]. This will continue until every node is visited, or until we are successfully collecting all Elemental Oculus.

After running the code, we will get this output, giving us the efficient path for collecting all the Elemental Oculus.

```
--- Phase 1: Determining Optimal Oculus Sequence via Teleport-Aware DFS
---
Optimal Visit Sequence Found: [0, 2, 5, 7, 9, 8, 18, 19, 20, 13, 14,
15, 16, 21, 22, 25, 26, 6, 1, 10, 11, 17, 12, 4, 3, 23, 24]

--- Phase 2: Building Final Path with Teleport-or-Walk Decisions ---
--- Traversal Complete ---
Start at Node 0.
```

```
Travel from 0 -> 2. Method: Walk (Cost: 294.2)
Travel from 2 -> 5. Method: Teleport to TW 29 (Cost: 191.1)
Travel from 5 -> 7. Method: Walk (Cost: 82.9)
Travel from 7 -> 9. Method: Walk (Cost: 80.7)
Travel from 9 -> 8. Method: Teleport to TW 29 (Cost: 84.9)
Travel from 8 -> 18. Method: Teleport to TW 31 (Cost: 83.6)
Travel from 18 -> 19. Method: Teleport to TW 32 (Cost: 34.0)
Travel from 19 -> 20. Method: Walk (Cost: 59.4)
Travel from 20 -> 13. Method: Teleport to TW 31 (Cost: 117.9)
Travel from 13 -> 14. Method: Walk (Cost: 48.7)
Travel from 14 -> 15. Method: Walk (Cost: 132.7)
Travel from 15 -> 16. Method: Walk (Cost: 203.1)
Travel from 16 -> 21. Method: Teleport to TW 35 (Cost: 116.4)
Travel from 21 -> 22. Method: Teleport to TW 35 (Cost: 84.9)
Travel from 22 -> 25. Method: Teleport to TW 35 (Cost: 105.0)
Travel from 25 -> 26. Method: Walk (Cost: 94.9)
Travel from 26 -> 6. Method: Teleport to TW 29 (Cost: 52.8)
Travel from 6 -> 1. Method: Teleport to TW 29 (Cost: 227.8)
Travel from 1 -> 10. Method: Teleport to TW 33 (Cost: 107.3)
Travel from 10 -> 11. Method: Teleport to TW 33 (Cost: 48.0)
Travel from 11 -> 17. Method: Teleport to TW 34 (Cost: 100.6)
Travel from 17 -> 12. Method: Walk (Cost: 171.1)
Travel from 12 -> 4. Method: Walk (Cost: 130.2)
Travel from 4 -> 3. Method: Walk (Cost: 23.3)
Travel from 3 -> 23. Method: Teleport to TW 34 (Cost: 237.5)
Travel from 23 -> 24. Method: Walk (Cost: 72.9)

--- Final Path (including Teleport Waypoints) ---
0 -> 2 -> TW-29 -> 5 -> 7 -> 9 -> TW-29 -> 8 -> TW-31 -> 18 -> TW-32 ->
19 -> 20 -> TW-31 -> 13 -> 14 -> 15 -> 16 -> TW-35 -> 21 -> TW-35 -> 22
-> TW-35 -> 25 -> 26 -> TW-29 -> 6 -> TW-29 -> 1 -> TW-33 -> 10 -> TW-
33 -> 11 -> TW-34 -> 17 -> 12 -> 4 -> 3 -> TW-34 -> 23 -> 24

Total Estimated Cost: 2985.95
```

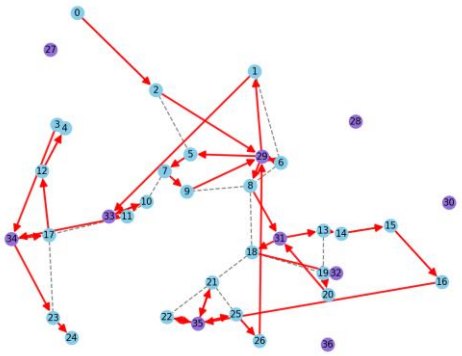

Fig. 16: Final path for collecting Elemental Oculus.

After running the code, we can get the approximated efficient path, including which Teleport Waypoint we should use and the final cost for collecting the Elemental Oculus, which is 2895.95. Combining all the steps needed to reach this solution, this algorithm takes *O(N*N + N*M)* time complexity, where N is the Elemental Oculus and M is the Teleport Waypoints.

VI. Conclusion

The challenges of collecting Elemental Oculus present a fascinating variant of the Traveling Salesman Problem. Given that exact solutions for the TSP are computationally intractable for non-trivial instances (NP-Hard), this paper introduces HUTAO-TSP (Hamiltonian Undirected Teleportation Aware Open TSP), a novel heuristic framework designed to approximate a highly efficient collection route in polynomial time.

As a heuristic model, HUTAO-TSP does not guarantee a mathematically optimal path. The primary limitations of the current model stem from its simplification of the game world into a 2D Euclidean space, while some in-game world has various terrains. But it successfully approximates the efficient path for collecting Elemental Oculus. Further research could extend this work in several promising directions, such as terrain and obstacle integration, three-dimensional pathfinding, dynamic heuristic, and generalization to other domains.

APPENDIX

The full source code for this paper can be found in this GitHub repository:

https://github.com/YavieAzka/hutao-tsp

Acknowledgment *(Heading 5)*

The author wishes to express gratitude to God Almighty for His blessings and grace, which made the completion of this paper possible.

The author extends his deepest appreciation to Dr. Ir. Rinaldi Munir, M.T, as the lecturer for the IF1220 Discrete Mathematics, for the invaluable guidance, constructive feedback, and profound knowledge shared throughout the semester and during the development of this work.

References

[1] Genshin Impact Wiki, "Genshin Impact Wiki," Fandom. [Online]. Available: https://genshin-impact.fandom.com/wiki/Genshin_Impact_Wiki. [Accessed: Jun. 10, 2025].

[2] R. Munir, "IF2120 Matematika Diskrit – Semester II Tahun 2024/2025," Institut Teknologi Bandung, [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025-2/matdis24-25-2.htm#SlideKuliah. [Accessed: Jun. 11, 2025].

[3] "Depth First Search or DFS for a Graph," *GeeksforGeeks*, Last Updated: 29 Mar. 2025. [Online]. Available: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/. [Accessed: Jun. 19, 2025].

[4] "Oculus," *Genshin Impact Wiki*, Fandom. [Online]. Available: https://genshin-impact.fandom.com/wiki/Oculus. [Accessed: Jun. 19, 2025].

[5] "Teleport Waypoint," *Genshin Impact Wiki*, Fandom. [Online]. Available: https://genshin-impact.fandom.com/wiki/Teleport_Waypoint. [Accessed: Jun. 11, 2025].

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

Ttd

Yavie Azka Putra Araly

13524077