Optimizing Deep-Space Observation Routes Using Shortest Hamiltonian Paths with Priority Constraints

Kenzo Yo - 13525016
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: kenzoyo123@gmail.com , 13525016@std.stei.itb.ac.id

***Abstract*—Observation opportunities for deep space objects are inherently limited by factors such as atmospheric conditions, observation window, and geospatial locations. Moreover, space telescopes are limited in energy, requiring an efficient observation route. This paper models the observation route as a complete graph, where vertices represent the objects and edges the distance between them. This allows the shortest Hamiltonian path, which is the path to visit every vertex or object once, to be found. Along with that, there is an added priority constraint to model the scenario where astronomers would find some deep space objects more interesting or of higher priority. Using Godot to visualize the graph and Held-Karp Algorithm to find the shortest path, along with the deep sky objects of the Centaurus constellation, this paper finds that the total angular distance used to traverse the observation route can be reduced by over 60% compared to randomly generated routes. As a result, by reducing the angular distance traveled, this can potentially make deep space observation more time and energy efficient.**

Keywords—Deep Space Objects; Observation Route; Graph; Shortest Hamiltonian Path; Optimization

I. Introduction

Deep space observation has always been an important component of Astronomy, allowing humans to better understand about space, stars, and in general the universe they live in. There are numerous Deep Space Objects (DSOs) in the sky, each having some unique characteristics, such as quasars, metallic stars, and so much more. However, the amount of time that can be used to observe DSOs is limited by multiple factors.

In space, telescopes have a limited window of time for observing a certain object, before being occluded by celestial objects like the Sun, Moon, and even the Earth. On top of that, they also must reposition themselves for the next observation object, all while having limited amounts of energy. On the ground, observation times are usually limited by their geospatial location and atmospheric condition, making certain DSOs only appear for a limited amount of time each night. Because of that, it's important to optimize the observation route for multiple DSOs to conserve both time and energy so that a single observation session could be as efficient as possible.

To optimize DSOs observation route, we can model it as a graph, where each object represents the vertices and the distance between each of them represents the graph's edges. This then allows the finding of the shortest Hamiltonian path, with an added priority constraint as some DSOs might have a higher observation priority.

This paper applies graphs and their theories to an Astronomy-related problem. By modeling this problem as finding the shortest Hamiltonian path, this paper aims to optimize deep space observation routes with an added priority constraint.

II. Theoretical Framework

A. Graph Definition

In discrete math, a graph is an abstraction of discrete objects and their connections to each other. In a graph, each vertex or node represents an object, while each edge or side represents a connection between two objects. A graph is also defined as the tuple (ordered pair) $G = (V, E)$, where V is a nonempty set of vertices and E is a set of edges. The set of edges E can be empty.

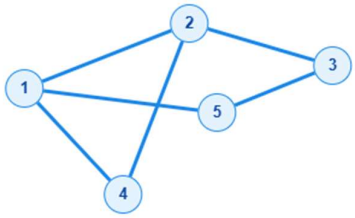


Fig. 1. A simple graph with 5 vertices. (Graph created using https://graphonline.top/en/)

B. Graph Terminologies

1) Simple graph: A graph is simple if no two same vertices are connected by more than one edge and no vertices have an edge that connects to itself (called a loop).

2) Multigraph: A graph is defined as multigraph if there exists either a loop or more than one edge that connects the same pair of vertices.

3) Weighted graph: A graph that has a weight on every of its edges.

4) Unweighted graph: A graph that doesn't have a weight on any of its edges.

5) Degree: The number of connections a vertex has, a loop represents two degrees, but otherwise every edge is exactly one degree.

6) Complete graph: A simple graph where every vertex is connected to every other vertex (other than itself). A complete graph made of n vertices is denoted as K_n and contains exactly $n(n-1)/2$ number of edges

7) Path: A path is a sequence of distinct vertices and edges where two consecutive vertices are joined by an edge.

8) Cycle: A cycle is a path that begins and ends on the same vertex.

9) Hamiltonian Path: A path is defined as Hamiltonian if it visits every vertex in a graph exactly once. A graph that contains a Hamiltonian path is called semi-Hamiltonian graph. A simple complete graph with n vertices contains exactly $(n-1)!/2$ distinct number of Hamiltonian paths [1].

C. Shortest Hamiltonian Path Algorithm: Held-Karp Algorithm

Finding the shortest Hamiltonian path is somewhat like finding the existence of a Hamiltonian path in a graph, also known as the Hamiltonian path problem, which is considered a NP-Hard problem. Because of their similarity, finding the shortest Hamiltonian path could also be considered NP-Hard.

Finding such a path can be done in two ways, one is through brute force that will yield a time complexity of $O(n!)$ as there exists a factorial number of Hamiltonian paths. The other way is through Held-Karp Algorithm which, by using dynamic programming, yields an exponential time complexity of $O(n^2 2^n)$. This paper will use the Held-Karp Algorithm, as it allows a faster computation of the shortest Hamiltonian path, especially when the number of vertices exceeds 10, allowing a greater flexibility before the algorithm eventually takes too long to compute a path.

While the Held-Karp Algorithm is made for finding the shortest Hamiltonian cycle, or more specifically to solve the Traveling Salesman Problem, the algorithm can be changed slightly towards the end to stop it from adding the starting point vertex as the final vertex, making it solve for the shortest Hamiltonian path instead.

The Held-Karp Algorithm mainly uses dynamic programming. The official definition of this algorithm is as follows.

Let us have $S \subseteq \{2, 3, 4, \ldots, n\}$, which means S is a subset of the set of vertices $\{2, 3, 4, \ldots, n\}$.

Let $l \in$ S and $C(S, l)$ as the minimum cost of starting from one city, visiting all other cities, then stopping at the city l. To find the minimum cost, a recursion relation is used.

$$C(\{l\}, l) = a_{1l}, \text{ for any } l. \quad (1)$$

$$C(S, l) = \min_{m \in (s-l)} [C(S - l, m) + a_{ml}] \quad (2)$$

Where (1) represents the base case when $n(S) = 1$. The term a_{1l} represents the weight from vertex 1 to vertex l.

The recurrence relation in (2) calculates the minimum cost of visiting all the vertices in S and stopping at l. This is done by considering every possible predecessor vertex $m \in (s - \{l\})$, calculating the cost of traversing m to l (shown with term a_{ml}), then selecting the minimum of all the costs. When this is applied repeatedly to larger subsets, the shortest Hamiltonian cycle can be found.

Before reconstructing the optimal cycle, let us define e as the minimum cost of the cycle:

$$e = \min_{l \in \{2, 3, \ldots, n\}} [C(\{2, 3, \ldots, n\}, l) + a_{l1}] \quad (3)$$

The reconstruction of the most optimal cycle is done by backtracking. Mathematically, they can be represented by two equations:

$$e = C(\{2, 3, \ldots, n\}, i_n) + a_{in1} \quad (4)$$

$$C(\{i_2, i_3, \ldots, i_{p+1}\}, i_{p+1}) = C(\{i_2, i_3, \ldots, i_p\}, i_p\}) + a_{ip\,ip+1} \quad (5)$$

Where for equation (5), $2 \leq p \leq n - 1$

To reconstruct the optimum cycle, the equation at (4) first finds the vertex i_n that is equal to e. In other words, it finds the vertex that yields the minimum cycle cost e. Next, (5) finds the predecessor of the i_n vertex by checking whether it satisfies equality in (5). This is repeated until i_2 is reached [2].

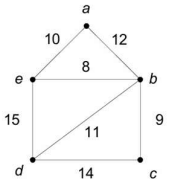
D. Adjacency Matrix

An adjacency matrix is defined as a two-dimensional matrix used to represent a graph. For a graph with n vertices, the adjacency matrix would be a $n \times n$ matrix where each element a_{ij} represents the adjacency relationship of vertices i and j.

For a simple, unweighted graph, a_{ij} would have a value of 1 if vertices i and j are connected by an edge, and 0 otherwise.

Meanwhile, for a simple, weighted graph, the value of element a_{ij} would be the weight of the edge that connects vertices i and j if they are connected, otherwise they would have a value of zero or infinity.

This matrix representation of a graph was selected because the Held-Karp Algorithm requires an adjacency matrix filled with distance or weight between vertices.



	a	b	c	d	e
a	∞	12	∞	∞	10
b	12	∞	9	11	8
c	∞	9	∞	14	∞
d	∞	11	14	∞	15
e	10	8	∞	15	∞

Fig. 2. A weighted simple graph (left) and its adjacency matrix (right). (Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/21-Graf-Bagian2-2026.pdf 6.pdf, accessed on 18/06/2026).

III. Methodology

A. Graph Representation of Deep Space Objects

To find the shortest Hamiltonian path of an observation route, we first need to represent the DSOs as a graph. Each DSO will be represented by a vertex, and to ensure Hamiltonian paths exist in the graph, a complete graph is created by connecting each vertex with every other vertex.

To convert the DSOs into a graph, we will use a simple program made using the Godot game engine, and consequently, the programming language used by Godot, GDScript. This is done to make visualization easier and more flexible, as each vertex can easily be represented with a node (Godot's building block), which can have various properties such as names, colors, shapes, and much more.

Here, the equatorial coordinate system is used for the position of each DSO. This is because the equatorial coordinate system is a widely used coordinate system for DSOs and using it would allow an easier integration of DSOs into the program.

Each DSO will have their data stored in a .csv file, with four columns: the name of the object, right ascension (RA), declination, and priority value of [0, 1). This is then parsed and put into an array for further processing.

Before putting the vertices on screen, we first must convert the celestial coordinate system to the cartesian one. The conversion is done directly, without any usage of trigonometry, as this is intended to be a simple visualization of the DSOs.

$$x = 9600 - RA / 360.0 \times 9600 \quad (6)$$

$$y = (1.0 - (declination + 90.0) / 180.0) \times 5400 \quad (7)$$

The values of 9600 and 5400 at (6) and (7) each represent the screen width and height of a desktop, originally 1920 and 1080, then scaled by 5 to make the graph less crowded. Notice that we subtracted 9600 by the conversion at (6). This is to flip the DSOs, as otherwise it would appear mirrored. Then, at the y-value calculation (7), notice that we add 90 to the declination in order normalize negative declination values. Here, we subtract 1 by the normalized declination value because Godot's coordinate system is flipped, with y-coordinate increasing downward.

After the direct conversion of the celestial coordinates to a cartesian one, we can then put each vertex into the screen. Next, we can make a program that connects each vertex with other vertex to create a complete simple graph.

To calculate the distance between each vertex or DSO, we use the angular distance formula, which goes as follows [3].

$$\cos d = \sin \phi_1 \sin \phi_2 + \cos \phi_1 \cos \phi_2 \cos (L_1 - L_2) \quad (8)$$

Then, to get d or the angular distance, we just need to apply the inverse cosine of the whole equation. We get the result in units of radians.

As an example of representing the DSOs into a graph, we can input some of the stars of the Orion constellation into the program and generate a complete graph from it.

TABLE I. Orion Constellation Input Data

obj_name	right_ascension	declination	priority
Betelgeuse	05:55:10.305	07:24:25.426	0
Rigel	05:14:32.272	-08:12:05.91	0
Bellatrix	05:25:07.900	06:20:59	0
Saiph	05:47:45.4	-09:40:11	0
Alnitak	05:40:45.5	-01:56:34	0
Alnilam	05:36:12.8	-01:12:06.9	0
Mintaka	05:32:00.4	-00:17:57	0

Fig. 3. Input data table for the few brightest stars of Orion. (Source for right ascension and declination: SIMBAD Astronomical Database [4]).

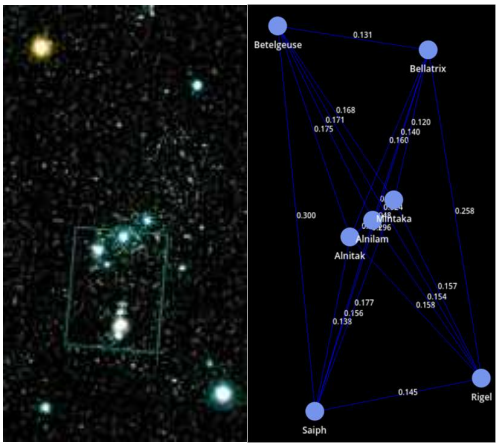


Fig. 4. The Orion constellation. The left image is an astrophotographed picture, while the right image is the complete graph representation using the coordinates provided by TABLE I. (Source: left image by Akira Fujii, https://science.nasa.gov/asset/hubble/orion-constellation/, accessed on 19/06/2026. The right image is by the Author)

B. Priority Based Constraint: Edge Weight Adjustment

In this paper, the priority of a DSO is defined as a value $p \in [0,1)$. There are two ways to implement a priority constraint, the first is to split a vertex that has a priority value into two new vertices, v_1 and v_2, connected by an edge with a weight equal to the priority value. The other way is to simply multiply all edges connected to the vertex with priority by its priority value.

Vertex splitting introduces more edges and vertices for every prioritized one, which can increase the time needed to find the shortest Hamiltonian path, considering the Held-Karp Algorithm has an exponential time complexity. Furthermore, vertex splitting, which will increase or subtract a cost based on the vertex's priority value, wouldn't have much of an effect over long angular distances or edges with a large cost.

The second implementation, while simpler, more naturally balances observation priority against the angular distances between DSOs, making it more consistent with a priority-based observation route, as a higher priority object should generally be scheduled earlier than a lower priority one.

Because of this, the latter or the second implementation is used instead. Mathematically, the second implementation can be modeled as:

$$w(i, j) = d(i, j) \times (1 - p_i) \tag{9}$$

Here, $w(i, j)$ is the modified edge weight between vertex i and j after multiplying the edge weight or angular distance $d(i, j)$ by $1 - p_i$. The subtraction of 1 with p_i is done to reduce the angular distance. This operation is done for every vertex that has a priority value, and for every edge that connects it to vertex j. Note that, the upper bound for p_i cannot be equal to one as it will cause the modified edge weight to be zero.

To more accurately model the priority constraint, this paper makes the highest priority DSO as the starting point of observation. This is because an astronomer might want to start an observation from a DSO with the highest priority.

C. *Finding the Shortest Hamiltonian Path*

The Held-Karp Algorithm is originally made to find the shortest Hamiltonian cycle. However, this paper requires the shortest Hamiltonian path instead. Because of that, this paper uses a modified version of the Held-Karp Algorithm that finds the Hamiltonian path, which is done by omitting the final step that connects the last visited vertex to back to the starting vertex.

The algorithm is usually done in two steps, and the implementation here uses the iterative version of the algorithm, which is adapted from [5]. The first step computes the minimum cost required to visit every subset of vertices. To achieve this, we first define a dynamic programming table of size $2^n \times n$, where n is the number of vertices. Each state of this table then represents a subset of visited vertices and the current ending vertex. Next, we also define a parent table that stores the predecessor vertex associated with the minimum cost of each state. This is done to easily reconstruct the final shortest Hamiltonian path later.

Bit masking is then used to represent the subsets of vertices. Each subset is represented by an n-length binary number, where a bit value of 1 represents a visited vertex, and 0 if otherwise. For example, the bit 00110_2 indicates that vertex 1 and 2 have been visited.

The second part of the algorithm reconstructs the shortest Hamiltonian path. We first find the minimum-cost state corresponding to the subset containing all vertices and the associated end vertex with it. This process is analogous to that of equation (3), except that the return edge to starting vertex is omitted. Then, by using the parent table we defined earlier, we iteratively find the predecessor to the end vertex. Once finished, we reverse the array, as it is appended from the end vertex at first. Afterwards, the algorithm is finished, and we get the shortest Hamiltonian path and the total cost of it.

```
for mask in range(1, nSubsets):
    if not (mask & 1):
        continue
    for j in range(1, n):
        if not (mask & (1 << j)):
            continue
        var prevMask = mask ^ (1 << j)

        for k in range(n):
            if prevMask & (1 << k):
                var cost = dp[prevMask][k] + dist_adj_matrix.buffer[k][j]
                if cost < dp[mask][j]:
                    dp[mask][j] = cost
                    parent[mask][j] = k
```

Fig. 5. GDScript code for the first part of the algorithm, which computes the minimum cost to visit every subset of vertices. (Source: Author).

```
var fullMask = (1 << n) - 1
var answer = INF
var endNode = -1

for j in range(n):
    if dp[fullMask][j] < answer:
        answer = dp[fullMask][j]
        endNode = j

var path: Array = []
var mask = fullMask
var prev
var curr = endNode
while curr != null:
    path.append(curr)
    prev = parent[mask][curr]
    mask ^= (1 << curr)
    curr = prev

path.reverse()
```

Fig. 6. GDScript code for the second part of the algorithm, which reconstructs the shortest Hamiltonian path. (Source: Author).

D. *Evaluating the Efficacy of Shortest Hamiltonian Path*

To evaluate the efficacy of finding the shortest Hamiltonian path, we can do a comparison test. Such a test will be done by first picking 1000 random Hamiltonian paths of DSOs observation route, then averaging them out and comparing it to the cost of the shortest Hamiltonian path. We pick the value of 1000 as a balance between computational cost and the reliability of the improvement score. Additionally, to compute the extreme ends, we will also find the best and worst cost from these 1000 random paths, by finding the minimum and maximum angular distance consecutively.

With this evaluation method in mind, we can calculate the average cost from randomly picking an observation route, analogous to what an astronomer might do when planning an observation session.

The comparison formula is a relatively simple and direct one, and is shown below:

$$I = (R - S) / R \times 100\% \tag{9}$$

Where I represent the improvement score, R is the random average path cost, and S is the shortest path score.

To compare the score between graphs or DSOs that have a nonzero priority value, an observation path is generated by repeatedly choosing the unvisited DSO with the highest priority value. After all the prioritized DSOs have been selected, the remaining DSOs are picked in random order. This process is repeated 1000 times, then averaged out and compared to the shortest Hamiltonian path with priority constraints, using formula (9).

IV. Results

A. Deep Space Objects Input Data

Let us pick the constellation of Centaurus and its stars, as well as a few interesting DSO such as a star cluster. The input data will be presented as a .csv file for the Godot program to process. We will also make a comparison of the shortest path generated with and without prioritized objects.

TABLE II. Centaurus Constellation Input Data (no Prioritized Objects)

obj_name	right_ascension	declination	priority
θ Centauri	14:06:41.32	-36:22:07.3	0
α Centauri	14:39:40.90	-60:50:06.5	0
β Centauri	14:03:49.44	-60:22:22.7	0
γ Centauri	12:41:31.20	-48:57:35.6	0
ε Centauri	13:39:53.27	-53:27:58.9	0
η Centauri	14:35:30.45	-42:09:27.9	0
ζ Centauri	13:55:32.43	-47:17:17.8	0
δ Centauri	12:08:21.54	-50:43:20.7	0
ι Centauri	13:20:36.07	-36:42:43.5	0
λ Centauri	11:35:46.93	-63:01:11.4	0
κ Centauri	14:59:09.70	-42:06:14.9	0
ν Centauri	13:49:30.30	-41:41:15.6	0
μ Centauri	13:49:37.01	-42:28:25.3	0
π Centauri	11:21:00.44	-54:29:27.7	0
d Centauri	13:31:02.67	-39:24:26.2	0
ω Centauri	13:26:47.28	-47:28:46.1	0
σ Centauri	12:28:02.41	-50:13:50.2	0

Fig. 7. Input data table for the Centaurus constellation, with no prioritized object (all DSOs have 0 priority). (Source: SIMBAD Astronomical database [4])

TABLE III. Centaurus Constellation Input Data (With Prioritized Objects)

obj_name	right_ascension	declination	priority
θ Centauri	14:06:41.32	-36:22:07.3	0
α Centauri	14:39:40.90	-60:50:06.5	0.3
β Centauri	14:03:49.44	-60:22:22.7	0
γ Centauri	12:41:31.20	-48:57:35.6	0
ε Centauri	13:39:53.27	-53:27:58.9	0
η Centauri	14:35:30.45	-42:09:27.9	0
ζ Centauri	13:55:32.43	-47:17:17.8	0
δ Centauri	12:08:21.54	-50:43:20.7	0
ι Centauri	13:20:36.07	-36:42:43.5	0
λ Centauri	11:35:46.93	-63:01:11.4	0
κ Centauri	14:59:09.70	-42:06:14.9	0
ν Centauri	13:49:30.30	-41:41:15.6	0.4
μ Centauri	13:49:37.01	-42:28:25.3	0
π Centauri	11:21:00.44	-54:29:27.7	0
d Centauri	13:31:02.67	-39:24:26.2	0
ω Centauri	13:26:47.28	-47:28:46.1	0.8
σ Centauri	12:28:02.41	-50:13:50.2	0.7

Fig. 8. The input data for centaurus constellation, with a few prioritized object, such as ω Centauri, which is a large globular star cluster. (Source: SIMBAD Astronomical Database [4])

B. Graph Visualization And Shortest Path Generation

After inputting the data, the Godot program processes them and first generates a K_{17} complete graph. Afterwards, we can make the program find the shortest path, which will be shown as a Hamiltonian path with its edges being drawn with the color red.

In the nonprioritized DSOs observation route graph, the starting point of the Hamiltonian path will always be the first vertex defined in the table, which in this case is θ Centauri. Meanwhile, in the prioritized DSOs observation route graph, the starting point is the DSO with the highest priority value, which is ω Centauri.

The shortest Hamiltonian path produced from the nonprioritized DSOs input data is as such: {θ Centauri, ι Centauri, d Centauri, ν Centauri, μ Centauri, η Centauri, κ Centauri, α Centauri, β Centauri, ε Centauri, ζ Centauri, ω Centauri, γ Centauri, σ Centauri, δ Centauri, π Centauri, λ Centauri}. The total cost of this Hamiltonian path is 1.80 radians.

Meanwhile, the shortest Hamiltonian path, using the prioritized DSOs input data, is: {ω Centauri, γ Centauri, σ Centauri, δ Centauri, π Centauri, λ Centauri, α Centauri, β Centauri, ε Centauri, ζ Centauri, μ Centauri, ν Centauri, d Centauri, ι Centauri, θ Centauri, η Centauri, κ Centauri}. The total cost of this Hamiltonian path is slightly larger, which is 1.84 radians.

Notice that the existence of a prioritized object (where p_i or priority value of a vertex > 0) changes the shortest path and its total cost. However, while the cost of the shortest Hamiltonian path on the prioritized graph is somewhat larger than the nonprioritized one, the path produced is more consistent with a priority-constrained observation path where higher priority objects should appear earlier.

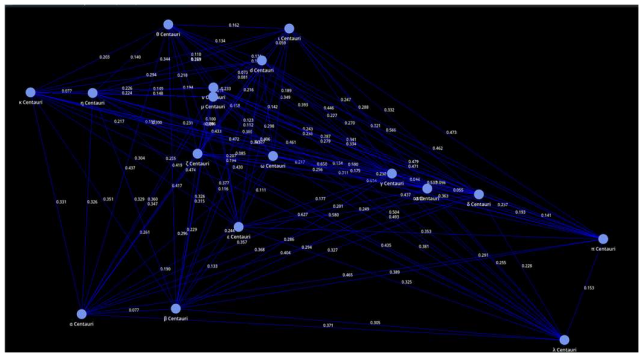


Fig. 9. The K_{17} complete graph of Centaurus constellation, generated by the Godot program. (Source: Author).

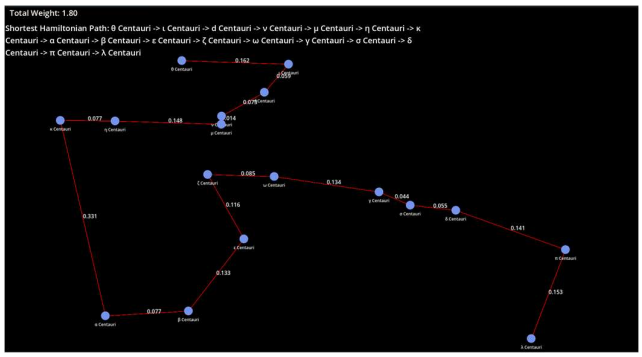


Fig. 10. The shortest Hamiltonian path for a nonprioritized input data. (Source: Author)

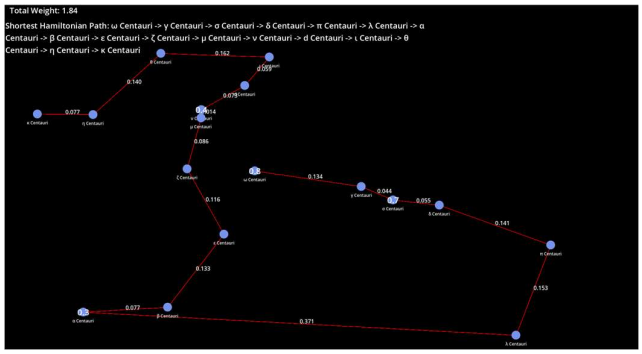


Fig. 11. The shortest Hamiltonian path using the prioritized input data. (Source: Author).

C. *Efficacy of Finding the Shortest Hamiltonian Path*

As discussed in the methodology section, we will first calculate the average cost of randomly selected 1000 Hamiltonian paths.

Using the nonprioritized DSOs input data, we get the average path cost of 4.58 radians, a best cost of 3.11 radians, and a worst cost of 5.98 radians. By using formula (9), we get an improvement score of around $I = 60.69\%$.

Next, using the prioritized DSOs input data, the average path cost is 4.83 radians, best cost is 3.53 radians, and the worst cost is 5.99 radians. Using those values, we get an improvement score of approximately $I = 61.9\%$.

From those results, we get a few interesting observations. One is that the average cost of prioritized input data seems to be slightly higher than the nonprioritized input data, be it from the shortest path found using the Held-Karp algorithm, or the randomly chosen path one. This can be explained by the fact that both the random and the shortest path method prioritize vertices with higher priority value, which will sacrifice some cost and make it a longer path, but at the same time more naturally reflects the fact that prioritized objects should appear or be visited earlier.

Next, the shortest path cost is significantly lower than the best cost calculated from 1000 random paths. This can mean that, for astronomers wanting to efficiently observe DSOs, it is always better to find the shortest path as the improvement or reduction on distance cost alone is well above 60%.

Finally, we can also conclude that, with such a high improvement compared to a random observation path, both a prioritized and nonprioritized observation route will benefit greatly by finding the shortest Hamiltonian path. This can significantly save time and energy on space telescopes and potentially reduce maintenance costs as well by reducing unnecessary movement. The same also applies to ground-based telescopes, as the best time window for observation is generally limited.

V. Conclusion

This paper demonstrates that optimizing the observation route of various DSOs is possible by modeling it as a complete graph, where each vertices represent a DSO and edges the angular distance to other vertices. Afterwards, finding the shortest Hamiltonian path to visit every vertex once by using a modified Held-Karp Algorithm.

By using the stars and DSOs of the Centaurus constellation, it is found that, compared to the average cost of 1000 random Hamiltonian paths, the shortest Hamiltonian path gives an improvement score of 60.69% for a nonprioritized observation route, and 61.9% for a prioritized one.

These findings suggest that graph-based optimization of observation routes can significantly improve the efficiency of DSOs observation routes. By reducing the total angular distance to traverse along the sky, this approach potentially saves both time and energy for both space telescopes and ground-based telescopes.

Video and Program Github Link

The Godot program the author created for visualization and pathfinding can be found here: https://github.com/KenzYo1/DSO-Observation-Route-Optimization/tree/main

The video can be found here:

https://www.youtube.com/watch?v=aDpd1TD8iX0

Acknowledgment

The author gives their many thanks and gratitude to God, as He has supported the author both physically and mentally through His numerous blessings and gifts. Next, the author also thanks his families, as they had been there every day, and especially during rough times. The author also thanks Dr. Ir. Rinaldi, M.T., the lecturer of the Discrete Math course, for teaching this course and allowing the author to understand it. Finally, the author also thanks his friends for the companionship they have given.

References

[1] Rinaldi. (2026). IF 1220 Matematika Diskrit Semester 2. Graf (Bagian 1). Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/20-Graf-Bagian1-2026.pdf

[2] Held, M., & Karp, R. M. (1962). A Dynamic Programming Approach to Sequencing Problems. *Journal of the Society for Industrial and Applied Mathematics*, *10*(1), 196–210. http://www.jstor.org/stable/2098806

[3] Meeus, J. "The Earth's Globe," in *Astronomical Algorithms*, 1st ed. Lichmond, Virginia, USA: Willman-Bell, 1991, ch. 10, pp. 80. [Online]. Available: https://dn760001.eu.archive.org/0/items/astronomicalalgorithmsjeanmeeus1991/Astronomical%20Algorithms-%20Jean%20Meeus%20%281991%29_text.pdf

[4] Wenger, M., "The SIMBAD astronomical database. The CDS reference database for astronomical objects", *Astronomy and Astrophysics Supplement Series*, vol. 143, EDP, pp. 9–22, 2000. doi:10.1051/aas:2000332.

[5] Rosetta Code. "Held-Karp algorithm." Rosetta Code. Accessed: June. 19, 2026. [Online.] Available: https://rosettacode.org/wiki/Held%E2%80%93Karp_algorithm#Python

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, 19 Juni 2026

Kenzo Yo
13525016