Enemy AI Movement Model in Lethal Company as a Hamiltonian Path Heuristic with Precision Radius Analysis

A Comparative Study of Classical and Modified Nearest Neighbor Heuristics

Richard Samuel Simanullang - 13524112
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: samsr472@gmail,com@gmail.com , 13524112@std.stei.itb.ac.id

***Abstract*— Enemy AI is a basic feature in modern games where the enemy behaves as if they have their own consciousness. Enemy AI varies . Lethal Company is a co-op horror game about scavenging at abandoned moons to sell scrap to the Company. Lethal Company is well-known for the enemy's AI, which uses good logical operations and varies among different species of monsters. This paper analyzes the pathfinding mechanism of the enemy AI in the video game Lethal Company. The AI's patrolling behavior, which requires it to visit a series of strategic points, is modeled as an attempt to solve the Traveling Salesperson Problem (TSP) through a Hamiltonian path heuristic approach. The goal of this study is to identify the algorithm used, visualize its behavior, and analyze its effectiveness compared to standard approaches.**

Keywords—component; Graph, Hamiltonian Circuits, Hamiltonian Paths, Traveling Salesperson Problem, Nearest Neighbor Heuristics, Lethal Company, Enemy AI

I. INTRODUCTION

The recent development of the gaming industry has brought about challenges in the design of unique and interesting games. This development cannot be separated from the mechanics and workings of the game behind the scenes. Lethal company is one of the games that has developed and is famous for the way it works behind the scenes.

Lethal Company is a co-op horror game about scavenging at abandoned moons to sell scrap to the Company. Lethal Company is well-known for the enemy's AI, which uses good logical operations and varies among different species of monsters. Even though Lethal Company is just a game, the enemy AI's logic regarding its search mechanism could help humans solve many everyday problems.

This paper examines how principles from discrete mathematics, particularly graphs, are used in the formation of enemy AI in Lethal Company. This paper specifically focuses on how the enemy AI performs search and exploration in Lethal Company.

As Fig 1.1 shows, nodes are placed around the map to form the basis of the enemy AI mechanism. Fig 1.2 shows the enemy executing the AI mechanism constructed from the developer's source code. This is what makes the lethal company game interesting to explore more deeply.

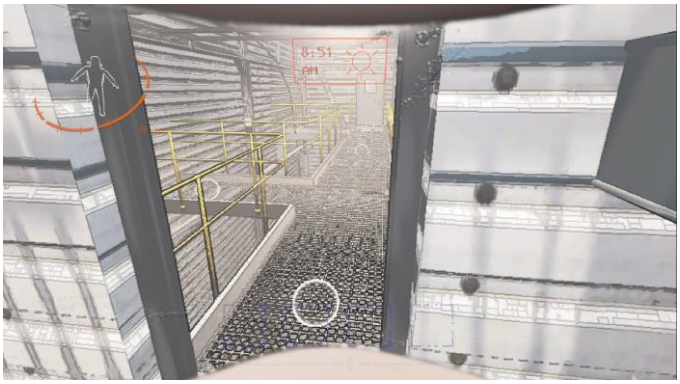


Fig 1.1 Nodes in Lethal Company
Source:
Lethal Company

Fig 1.2 Enemy's AI working in realtime
Source:
Lethal Company

II. THEORETICAL BASIS

A. *Graph*

Graph defined as a collection of nodes connected by lines or edges. In this case,

$$G = (V, E)$$

Where G represents the graph, V represents the Vertices, and E as its Edges.

Graph is commonly used to represent discrete objects and relationships between them. In Fig 2.1, A, B, C, and D considered as the vertices/nodes and the lines connecting them considered as edges.

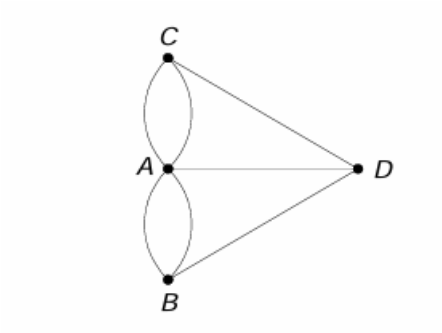


Fig 2.1 Graph with its nodes and edges
Source :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

Based on the presence of loops or multi-edges in the graph, graphs are classified into two types. First one is simple graph. Simple graph is a graph that connect its nodes with single edges.

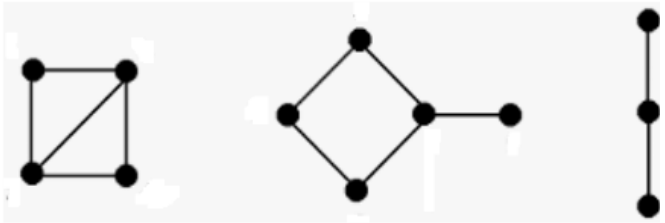


Fig2.2 Simple Graph
Source :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

The second one is unsimple graph. Unsimple graph is a graph that has multi-edges and loops.

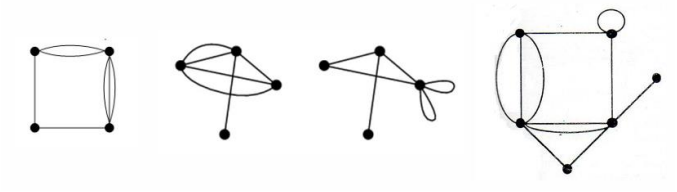


Fig 2.3 Unsimple Graph
Source :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

B. *Graph Terminology*

There are some terminologies in graph, such as :

1. Adjacent

   Two vertices are said to be adjacent if they are directly connected.

2. Incidency

   For arbitrary edge $e = (v_j, v_k)$ say e is adjacent to v_j, or e is adjacent to v_k.

3. Isolated Vertex

   Isolated Vertex is a vertex that has no edge that is adjacent to it.

4. Null Graph

   Graph whose edge set is the empty set.

5. Degree

   Degree of a vertex is the number of edges that are adjacent to that vertex. Notation : d(v).

6. Path

   A path is a sequence of vertices where each consecutive pair of vertices is connected by an edge. In a simple graph, path formed when no vertex is repeated, ensuring each vertex is visited at most once.

7. Cycle/Circuit

   Cycle/Circuit is a path that starts and ends at the same vertex.

8. Subgraph

   Suppose $G = (V, E)$ is a graph. $G_1 = (V_1, E_1)$ is a subgraph of G if $V_1 \subseteq V$ dan $E_1 \subseteq E$.

C. *Hamiltonian path and Hamiltonian circuit*

Hamiltonian path is a path that goes through each vertex in the graph exactly once, but does not end at the same vertex as the starting vertex. Hamiltonian circuit is a circuit that goes through each vertex in graph exactly once, except the origin vertex. Graphs that have Hamiltonian circuits are called

Hamiltonian graphs, while graphs that only have Hamiltonian paths are called semi-Hamiltonian graphs.

Fig 2.4 Hamiltonian graph
Source :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf

D. Travelling Salesperson Problem and Nearest Neighbor Algorithm

The Travelling Salesperson Problem (TSP) is a well-known optimization challenge in computer science. It asks for the shortest possible route that visits each city exactly once and returns to the starting point. It means that TSP is significantly linked to the Hamiltonian Circuit. TSP is an NP-hard problem, meaning there is no known efficient solution for large datasets, but various algorithms can provide exact or approximate solutions.

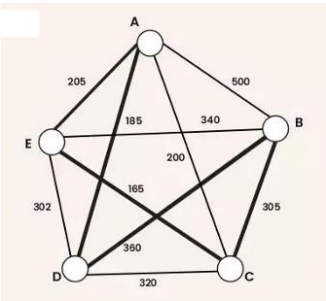


Fig 2.5 Travelling Salesman Problem
Source :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf

One of the approaches in TSP Algorithm is Nearest Neighbor Algorithm. Nearest Neighbor(NN) Algorithm starts at a random city and repeatedly visits the nearest unvisited city. The primary advantage of the NN algorithm is its speed and simplicity. For a graph with *n* vertices, the algorithm has a time complexity of $O(n^2)$. Nearest Neighbor Algorithm uses distance metrics to identify nearest neighbor, these neighbor are used for classification and regression task[4]. To identify nearest neighbor we use below distance metrics :

1. Euclidean Distance

Euclidean distance is defined as the straight line distance between two points in a plane or space.

$$d(x, X_i) = \sqrt{\sum_{j=1}^{d} \left(x_j - X_{i_j}\right)^2}$$

2. Manhattan Distance

This is the total distance the salesman would travel if the salesman could only move along horizontal and vertical lines like a grid or city streets

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

3. Minkowski Distance

Minkowski distance is like a family of distances, which includes both Euclidean and Manhattan distances as special cases.

$$d(x, y) = \left(\sum_{i=1}^{n} (x_i - y_i)^p\right)^{\frac{1}{p}}$$

The nearest neighbor algorithm (NN) starts its tour from a fixed vertex i_1, goes to the nearest vertex i_2 (i.e., $c(i_1, i_2) = min\{c(i_1, j) : j \neq i_1\}$), then to the nearest vertex i_3 (from i_2) distinct from i_1 and i_2, etc.

III. ANALYSIS

Lethal company enemy's AI uses a different algorithm than any TSP problem solving method. They use a heuristic Nearest Neigbor algorithm in the enemy roaming/searching mechanism. The following algorithm that will be explained later does not represent solving the TSP problem directly, but rather gives 'character' to the enemy.

The author has decompiled the Lethal Company game to get the original source code. The following is a section-by-section explanation of the enemyAI C# code that constructs the enemy roam/search logic :

A. Search Initialization

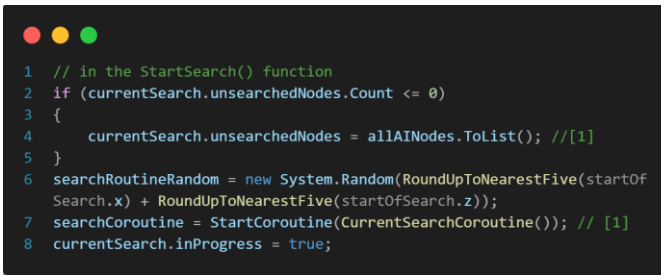
As soon as enemy spawned in, enemy's AI starts its roaming/searching mechanism by checking all nodes that already placed across the entire map.

```
if (enemyType.isOutsideEnemy)
{
    allAINodes = GameObject.FindGameObjectsWithTag("OutsideAINode");
    if (enemyType.nestSpawnPrefab != null)
    {
        for (int i = 0; i < RoundManager.Instance.enemyNestSpawnObjects.Count; i++)
        {
            if (RoundManager.Instance.enemyNestSpawnObjects[i] == null)
            {
                RoundManager.Instance.enemyNestSpawnObjects.RemoveAt(i);
            }
            else if (RoundManager.Instance.enemyNestSpawnObjects[i].enemyType == enemyType)
            {
                UseNestSpawnObject(RoundManager.Instance.enemyNestSpawnObjects[i]);
                break;
            }
        }
    }
    if (GameNetworkManager.Instance.localPlayerController != null)
    {
        EnableEnemyMesh(!StartOfRound.Instance.hangarDoorsClosed || !GameNetworkManager.Instance.localPlayer
Controller.isInHangarShipRoom);
    }
}
else
{
    allAINodes = GameObject.FindGameObjectsWithTag("AINode");
}
```

Fig 3.1 AINode search
Source :
Lethal Company Developers

All detected nodes are put into a list that will be used later. The AI starts every search coroutine by finding every unsearched node that is available in the AINode list. This step effectively defines the problem “visit all nodes in all AINodes”.

This code logic is an application of the Hamiltonian path behaviour. Because the game program code is made according to the developer's wishes, we can manipulate the program code so that when the enemy is on the last unvisited node, the next node is the first node accessed. That way we can assume that we can use Traveling Salesman Problem (TSP) to model the weighting problem in this pathfinding AI.



```
// in the StartSearch() function
if (currentSearch.unsearchedNodes.Count <= 0)
{
    currentSearch.unsearchedNodes = allAINodes.ToList(); //[1]
}
searchRoutineRandom = new System.Random(RoundUpToNearestFive(startOf
Search.x) + RoundUpToNearestFive(startOfSearch.z));
searchCoroutine = StartCoroutine(CurrentSearchCoroutine()); // [1]
currentSearch.inProgress = true;
```

Fig 3.2 add AINode to list
Source :
Lethal Company Developers

B. Next node selection (Nearest Neighbor Heuristics)

In CurrentSearchCoroutine(), AI calls ChooseNextNodeIn SearchRoutine() to decide where to go next. This coroutine uses the Nearest Neighbor heuristic algorithm to select the next node from the list of unvisited nodes. The algorithm process can be broken down into this following steps:

1. Settle a reference point

   Unlike the usual Nearest Neighbor algorithm which uses the object’s current location as a reference point, this algorithm uses the previous location as its reference point and stores it in the currentSearchStartPosition() variable.

2. Distance calculations

   The algorithm iterates through the unvisited (available) node in the unsearched node list. For each node, it calculates the pathDistance between the node and the previous node that stored in currentSearchStart Position() using Euclidean distance. Let d_n denote the distance between node $n \in S$ and the previously visited node p_{prev}, this calculation can be expressed as:

$$\forall n \in S, d_n = dist\left(n, p_{previous}\right)$$

$$n_{\text{next}} = \arg\min_{n \in S} d_n$$

   where S is the set of unvisited nodes, and p_{prev} is the reference node from the previous process.

```
public float GetCurrentDistanceOfSearch()
{
    return Vector3.Distance(currentSearchStartPosition, currentTargetNode.transform.position);
}
```

Fig 3.3 Distance calculation with Euclidean distance
Source:
Lethal Company Developers

3. Comparison and selection for the next node

   The algorithm stores the shortest distance in closestDist [2] variable, and the node associated with that distance in chosenNode variable. The chosenNode will be the next destination for the AI, while the currentNode will be the currentSearchStartPosition as the coroutine starts again.

```
// in ChooseNextNodeInSearchRoutine()
if (!currentSearch.startedSearchAtSelf)
{
    GetPathDistance(currentSearch.unsearchedNodes[i2].transform.position, currentSearch.
currentSearchStartPosition);
}
if (pathDistance < closestDist && (!currentSearch.randomized || !gotNode || searchRoutin
eRandom.Next(0, 100) < 65))
{
    closestDist = pathDistance; //[2]
    chosenNode = currentSearch.unsearchedNodes[i2];
    gotNode = true;
    if (closestDist <= 0f && !currentSearch.randomized)
    {
        break;
    }
}
```

Fig 3.4 Choosing the next node
Source:
Lethal Company Developers

C. Precision Radius Implementation

Lethal company uses precision radius on players and nodes to do the node check mechanism. Precision radius, or searchPrecision(in Lethal Company terms), used in two ways:

1. Mark visited Node

   In CurrentSearchCoroutine, the AI is considered to have reached the next node if the distance is smaller than the searchPrecision.

2. Eliminate nearby nodes (Radius based pruning)

After the enemy sets the current next node as visited, it also sets all nodes within the searchPrecision radius as visited.

```
// inside CurrentSearchCoroutine while, after SetDestinationToPosition called
SetDestinationToPosition(currentSearch.currentTargetNode.transform.position);
for (int i = currentSearch.unsearchedNodes.Count - 1; i >= 0; i--)
{
    if (Vector3.Distance(currentSearch.currentTargetNode.transform.position, currentSearch.
unsearchedNodes[i].transform.position) < currentSearch.searchPrecision)
    {
        EliminateNodeFromSearch(i);
    }
    if (i % 10 == 0)
    {
        yield return null;
    }
}
```

Fig 3.5 Eliminating nearby nodes within searchPrecision radius
Source:
Lethal Company Developers

IV. IMPLEMENTATION AND RESULT

To simulate the enemy search and roam algorithm in the lethal company, Python programming language is used to visualize how the lethal company heuristic algorithm traces the hamiltonian path.

In addition, the number of weights on a semi-Hamiltonian graph is compared when using the original (classical) nearest neighbor algorithm with the lethal company nearest neighbor heuristic algorithm.

A. Initialization

The simulation began by importing important libraries, parameters, and node position generation. The starting node is set as node 0, to know where the graph pathing starts.

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
import math

# --- Parameters ---
NUM_NODES = 30
SEARCH_RADIUS = 0.2
START_NODE = 0

# Generate same node positions
np.random.seed(42)
pos = {i: (np.random.rand(), np.random.rand()) for i in range(NUM_NODES)}
```

Fig 4.1 Visualization and Comparison Initialization
Source:
Author's source code

B. Simulation

The simulation function has two parameters, first the type of search algorithm used, and second is the title of the image. The function begins by initializing several variables to track the simulation state, such as:

- physically_visited = []: An ordered list to store the sequence of nodes the object physically travels to
- radius_visited = set () : A set to store nodes that considered as visited because they fall within the search radius of the physically_visited node.
- edge_trail = []: A list of tuples, where each tuple represents a directional edge (start_node, end_node).
- total_weight = 0 : A float data type to store the total distance of the path traveled by the object.
- curr_node = START_NODE: an integer that represents the current node where object is, initialized to the starting node.

The main simulation loop in Fig 4.2 continues as long as the total number of visited nodes (including radius visited nodes) is less than the total number of nodes available. This loop ensures the process imitates the Hamiltonian path.

Inside the loop, the first step is to check the current node. If the current node is valid, then add the node into the physically_visited list. Before going into the next node, the program traverse into every nodes that's not already physically visited and radius visited. Any node that is within the searchPrecision range of the current node, added into the radius_visited list.

```
# --- Simulation ---
def simulation(tipe_heuristik, jdl):
    # Run a pathfinding simulation based on the given heuristic type
    # and generate a visualization plot.

    # Variables to track status
    physically_visited = []
    radius_visited = set()
    edge_trail = []
    total_weight = 0

    curr_node = START_NODE

    # Main algorithm loop
    while len(physically_visited) + len(radius_visited) < NUM_NODES:

        if curr_node not in physically_visited and curr_node not in radius_visited:
            physically_visited.append(curr_node)

            for i in range(NUM_NODES):
                if i != curr_node and i not in physically_visited and i not in radius_visited:
                    if distance(pos[curr_node], pos[i]) < SEARCH_RADIUS:
                        radius_visited.add(i)
```

Fig 4.2 Initializing simulation code
Source :
Author's source code

Next is to determine which node will be the next node. This is the part where the 2 algorithms are applied.

1. Identify unvisited nodes
   As seen in Fig 4.3, a list of unvisited nodes is created by finding all nodes that aren't already in visited lists (physically and radius based).
2. Compare the algorithm
   The type of algorithm parameter determines the logic for selecting the next node.
   - Classic Nearest Neighbor Algorithm
     The reference point (ref_node) is the current node. The object simply chooses the closest unvisited node to the current node.
   - Lethal Company NN heuristic

The reference point (ref_node) for the distance calculation is set to the *second-to-last* physically visited node (physically_visited[-2]). This means the object decides its next move based on where it was *before* its current location. This can create less direct, more unpredictable paths.

```
        all_visited_node = set(physically_visited).union(radius_visited)
        unvisited_node = [n for n in range(NUM_NODES) if n not in all_visited_node]

        if not unvisited_node:
            break

        if tipe_heuristik == 'lethal_company_style':
# Lethal Company Heuristic NN : Find the closest node from the physically visited prev node
            ref_node = physically_visited[-2] if len(physically_visited) > 1 else START_NODE
        else:
# Classic NN : Find the closest node to the curr node
            ref_node = curr_node

        shortest_dist = float('inf')
        next_node = -1

        for candidate_node in unvisited_node:
            dist_to_ref = distance(pos[ref_node], pos[candidate_node])
            if dist_to_ref < shortest_dist:
                shortest_dist = dist_to_ref
                next_node = candidate_node

        if next_node == -1:
            break

        edge_weight = distance(pos[curr_node], pos[next_node])
        total_weight += edge_weight
        edge_trail.append((curr_node, next_node))
        curr_node = next_node
```

Fig 4.3 Next node selection based on the algorithm
Source :
Author's source code

C. *Visualization*

The program visualizes the paths on the graph one by one based on its algorithm. Nodes that belong to the physically visited section are marked in blue. The radius visited ones are marked in green. The unvisited ones is marked in gray for debugging purposes (as we already know that Hamiltonian path doesn't allow any unvisited nodes).

```
# --- Visualization ---
G = nx.Graph()
G.add_nodes_from(range(NUM_NODES))

# Set node colors
node_color = []
for i in range(NUM_NODES):
    if i in physically_visited:
        node_color.append('skyblue')
    elif i in radius_visited:
        node_color.append('lightgreen')
    else:
        node_color.append('lightgray')

# Create the plot
plt.figure(figsize=(13, 13))
plt.title(f"{jdl}\nWeight total (distance): {total_weight:.4f}", fontsize=16)

# Draw edges and nodes
nx.draw_networkx_edges(G, pos, alpha=0.1, edge_color='gray')
nx.draw_networkx_nodes(G, pos, node_color=node_color, node_size=500)

# Draw the path taken by the enemy
D = nx.DiGraph()
D.add_edges_from(edge_trail)
nx.draw_networkx_edges(
    D, pos,
    edge_color='red',
    width=2.0,
    arrows=True,
    arrowstyle='->',
    arrowsize=20
)

# Draw node labels
node_labels = {n: str(n) for n in G.nodes()}
nx.draw_networkx_labels(G, pos, labels = node_labels, font_color='black',
                        font_weight='bold', font_size=10)

# Legenda
element = [
    plt.Line2D([0], [0], color='skyblue', marker='o',
               linestyle='', markersize=10, label='Node Dikunjungi Fisik (Physically Visited)'),
    plt.Line2D([0], [0], color='lightgreen', marker='o',
               linestyle='', markersize=10, label='Node Dikunjungi via Radius (Radius-Visited)'),
    plt.Line2D([0], [0], color='lightgray', marker='o',
               linestyle='', markersize=10, label='Node Belum Dikunjungi (Unvisited)'),
    plt.Line2D([0], [0], color='red', lw=2, label='Lintasan Enemy (Path)')
]
plt.legend(handles=element, loc='upper right')

plt.xlabel("Koordinat X")
plt.ylabel("Koordinat Y")
plt.grid(True, linestyle='--', alpha=0.5)
plt.axis('on')
plt.show()

return total_weight
```

Fig 4.4 Visualization
Source:
Author's source code

D. *Final code and Result*

The final part of the code runs the program with a total of 2 runs, namely the program with the classic NN algorithm, and with the Lethal Company Nearest Neighbor Heuristic algorithm.

```
# --- Run both simulation ---
print("MRunning simulation for Classic Nearest Neighbor (NN)...")
classic_weighted_nn = simulation('classic_nn', "Classic Nearest Neighbor (NN)")

print("\nRunning simulation for Lethal Company Heuristic Nearest Neighbor...")
heuristic_weighted_nn = simulation('lethal_company_style', "Lethal Company Heuristic Nearest Neighbor")


# --- Final comparison ---
print("\n--- TOTAL WEIGHT COMPARISON ---")
print(f"Classic Nearest Neighbor: {classic_weighted_nn:.4f}")
print(f"Lethal Company Heuristic Nearest Neighbor: {heuristic_weighted_nn:.4f}")

if classic_weighted_nn < heuristic_weighted_nn:
    print("\nKesimpulan: Heuristik Klasik NN menghasilkan lintasan yang lebih pendek (lebih optimal).")
else:
    print("\nKesimpulan: Heuristik Asli (Gaya Lethal Company) menghasilkan lintasan yang lebih pendek (lebih optimal).")
```

Fig 4.5 Final code
Source:
Author's source code

The results presented in Figure 4.6 visualize the Hamiltonian path of the graph using the classic Nearest

Neighbor algorithm. The distance (total weight) traveled using this algorithm is 3.5873 units.

Fig 4.6 Classic NN Hamiltonian Path
Source:
Author's archive

The results presented in Figure 4.7 visualize the Hamiltonian path of the graph using the Lethal Company Nearest Neighbor Heuristic algorithm. The distance (total weight) traveled using this algorithm is 5.8456 units.

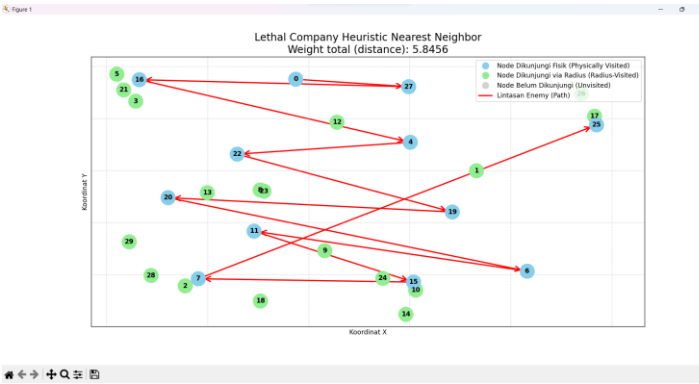


Fig 4.7 Lethal Company Nearest Neighbor Heuristic Hamiltonian path.
Source:
Author's archive

From the previous two results, it is found that the classic Nearest Neighbor algorithm is more effective than the Lethal Company Heuristic algorithm in terms of total load (distance).

V. CONCLUSION

This study analyze and demonstrates the implementations of Hamiltonian paths in Lethal Company Enemy's AI searching mechanism. By analyzing the original source code, we know that Lethal Company Enemy's AI searching mechanism uses Nearest Neighbor heuristic. However, its implementation has unique and non-standard heuristic properties. Instead of applying the classic greedy approach of choosing the closest node to the current position, the game's algorithm makes decisions based on the distance from previously visited nodes.

The "one-step-back" nature of this heuristic results in fundamentally different behavior. This causes the AI to often take unexpected and locally non-optimal paths, as its decisions are not always relevant to its current surroundings. This behavior also emphasizes the "patrolling" principle present in horror games in general.

When the effectiveness of this algorithm is evaluated using the Traveling Salesperson Problem (TSP) as a model problem- where the goal is to find the shortest route to visit all points - the search algorithm used by Lethal Company proves to be ineffective. Previous simulations consistently show that the total weight (distance) of the paths generated by this heuristic is significantly higher than those generated by the classic Nearest Neighbor algorithm. From a pure optimization perspective, this algorithm is a poor TSP solver.

Nonetheless, this "poor efectiveness" seems to be a purposeful design choice. In game design, the main goal is not the efficiency, but rather to create a unique and engaging gaming experience. With unpredictable patrol paths, players of this game cannot easily guess the enemy's AI movements. Thus, Lethal Company sacrifices path optimization in favor of creating enemies that feel more organic, dynamic, and ultimately, more threatening.

VI. APPENDIX

The following is the source for the code that have been analyzed and for implementing a comparison between the classic Nearest Neighbor Algorithm and Lethal Company Heuristics Nearest Neighbor Algorithm :

https://github.com/Lloyd565/Discrete-Mathematics-Paper---Enemy-AI-Movement-in-Lethal-Company-as-a-Hamiltonian-Path-Heuristic-

VII. ACKNOWLEDGMENT

I would like to express my gratitude to to God Almighty for His guidance which have enabled me to complete this paper for IF1220 Matematika Diskrit. I personally would also like to thank Arrival Dwi Sentosa, S.Kom., M.T., the lecturer for the IF1220 Discrete Mathematics course, for teaching the course material clearly and thoroughly, making it easier for me to complete this paper.

REFERENCES

[1] [1] R. Munir, "Graf (bagian 1)," materi kuliah, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2024. [Daring]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf.

[2] [2] R. Munir, "Graf (bagian 3)," materi kuliah, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2024. [Daring]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf.

[3] [3] WSCube Tech, "Travelling salesman problem in DSA," WSCube Tech Resources. [Daring]. Tersedia: https://www.wscubetech.com/resources/dsa/travelling-salesman-problem. [Diakses: 20 Juni 2025].

[4] [4] GeeksforGeeks, “K-nearest neighbours,” GeeksforGeeks, 23 Mei 2024. [Daring]. Tersedia: https://www.geeksforgeeks.org/machine-learning/k-nearest-neighbours/.

[5] [5] G. Gutin, A. Rafiey, S. Szeider, and A. Yeo, “The traveling salesman problem,” dalam *Handbook of Graph Theory, Combinatorial Optimization, and Algorithms*, K. Thulasiraman, S. Arumugam, A. Brandstädt, dan T. Nishizeki, Eds. Boca Raton, FL: Chapman & Hall/CRC, 2015, ch. 48.

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, 20 Juni 2025

Richard Samuel Simanullang 13524112