Application of Minimum Spanning Trees and Graph Theory in Generating Perfect Mazes

Jason Edward Salim - 13524034
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: jasonedwardsalim@gmail.com , 13524034@std.stei.itb.ac.id

***Abstract*—This paper explores algorithmic approaches using the graph theory of minimum spanning tree and basic graph traversal algorithms for generating and solving perfect mazes. We will analyze various algorithms including methods like Prim's algorithm and breadth-first search and evaluate their properties in creating solutions. A perfect maze is a specific type of complex passage network characterized by the inexistence of circuits and a unique path between any two points with no isolated paths. To generate such structure, randomized prims algorithm was applied by treating each maze cell as graph vertex and potential walls as removable edges. The maze is iteratively generated by connecting unvisited cells through arbitrarily selected walls. For solving the generated maze, the breadth first search approach was used due to its ability to guarantee the shortest path in unweighted graphs. The entire implementation was conducted in Python, using NumPy for grid operations and Matplotlib for visualizing the results. Obtained results prove that the proposed method successfully generates and solves randomized perfect mazes.**

***Keywords*—graph theory; minimum spanning tree; maze generation; perfect maze**

I. Introduction

A maze is an intricate and confusing set of connecting routes in which it is hard to find one's exit. A maze is related to fields such as robotics, animal studies, or spiritual and recreational activities. A perfect maze is defined as a complex network with inexistence of circuits and a unique path between any two points with only one possible solution. The layout of the maze itself is often used as a basic model for various problems related to pathfinding and navigation.

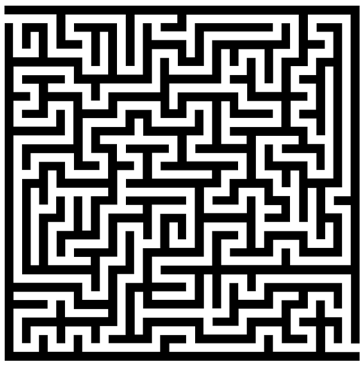

Figure 1. Maze Example. Adapted from [2].

This paper explores algorithmic approaches using the graph theory of minimum spanning tree and basic graph traversal algorithms for generating and solving perfect mazes. We will analyze various algorithms including methods like Prim's algorithm and breadth-first search and evaluate their properties in creating solutions.

II. Theoretical Basis

A. Graph

1) Graph Definition

A graph G = (V, E) is a mathematical structure that is consisted of a non-empty set of vertices V and a set of edges E. In this context, each vertex represents a cell or intersection and each edge represents a possible route or connection between two adjacent cells.

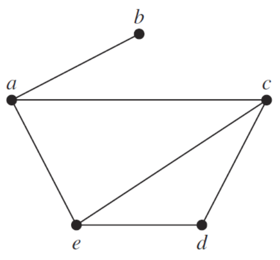

Figure 2. Simple Graph Example. Adapted from [1].

2) Graph Terminology

Graphs can be classified into multiple types based on the properties of their edges. A simple graph is undirected and is not allowed to have multiple edges and loops. A multigraph is undirected and is allowed to have multiple edges but no loops. A pseudograph is undirected and is allowed to have multiple edges and loops. There exist simple directed graphs, directed multigraph, and mixed graphs for directed graphs. We focus mainly on undirected graphs, as the layout of a maze is undirected, each passage between cells can be traversed in both directions.

3) Graph Representation

There are several methods to represent a graph:

1. Adjacency Matrices

The adjacency matrix **A** of undirected graph ***G*** is a zero-one matrix with 1 as its (i, j)th entry when v_i and v_j are adjacent, and 0 if otherwise. If **A** is = $[a_{ij}]$, then

$$a_{ij} = \begin{cases} 1, & if\ \{v_i, v_j\}\ is\ an\ edge\ of\ G, \\ 0, & otherwise. \end{cases} \quad (1)$$

For example, the graph in figure 2 adjacency matrix is:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

with ordering of vertices a, b, c, d, e respectively.

2. Adjacency List

The adjacency lists of undirected graph ***G*** is a lists of vertices that are adjacent to each vertex of the graph. For example, the adjacency lists for the graph in figure 2 are:

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

3. Incidence Matrices

The incidence matrix of undirected graph ***G*** is a zero-one matrix with 1 as its (i, j)th entry when v_i and e_j are incident, and 0 if otherwise. If **M** is = $[m_{ij}]$, then

$$m_{ij} = \begin{cases} 1, & when\ edge\ e_j\ is\ incident\ with\ v_i, \\ 0, & otherwise. \end{cases} \quad (2)$$

B. Tree

1) Tree Definition

A tree is an undirected graph that is connected and does not contain multiple edges or loops. Any tree must be a simple graph. An undirected graph is a tree if and only if there is a unique path between any two of its vertices. These properties align perfectly with the definition of a maze, in which there exists a unique path between any two given points. Therefore, a maze can be modeled as a spanning tree over the grid-shaped graph that represents the grid of cells. This connection forms the basis for using minimum spanning trees algorithms in maze generation.

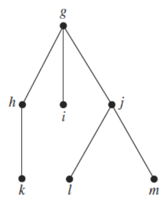

Figure 3. Rooted Tree Example. Adapted from [1].

A rooted tree is a tree where a vertex are acting as the root and other edges are directed away from the root. A rooted tree is called an *m-ary tree* if every internal vertex has no more than *m* direct descendants. A *full m-ary tree* means every internal vertex of that tree has exactly *m* children.

2) Application of Trees

Binary search trees is an efficient searching algorithm for ordered elements. Each child of a vertex is assigned as a right or left child. Vertices represent elements and are given values so that the value of a vertex is both bigger than the values of all vertices in its subtree on the left and smaller than the values of all vertices in its subtree on the right.

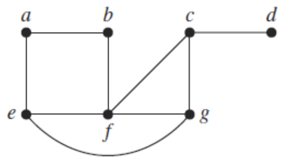

Figure 4. Simple Graph G. Adapted from [1].

3) Spanning Trees and Minimum Spanning Trees

A spanning tree of simple graph *G* is a subgraph of *G* that is a tree containing every vertex of *G*. Spanning trees of graph *G* can be found by removing edges that form simple circuits.

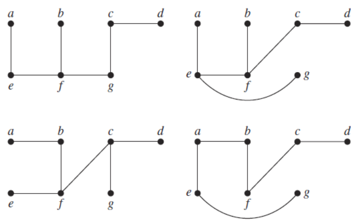

Figure 5. Spanning Trees of G. Adapted from [1].

A maze can be formed into a spanning tree of the grid-shaped graph. Creating a spanning tree guarantees connectivity and inexistence of circuits.

A minimum spanning tree in a weighted graph is a spanning tree where the total values of the edges weights in the tree is a minimum. A wide variety of problems are solved using various algorithms for finding minimum spanning trees, such as:

1. Prim's Algorithm

```
procedure Prim (input G:  Graph, output T: Tree)
{I.S. G is an undirected graph with n vertices that is
connected and has weights in each of its vertices.}
```

```
{F.S. A minimum spanning tree of graph G}

ALGORITHM
T ← edge with the least weight in graph G
i traversal [1..(n-2)]
    e ← a minimum-weight edge incident to a vertex in T
    {e must not form a simple circuit if added to T }
    T ← T added with e
```

2. Kruskal's Algorithm

```
procedure Kruskal (input G:  Graph, output T: Tree)
{I.S. G is an undirected graph with n vertices that is connected and has weights in each of its vertices.}
{F.S. A minimum spanning tree of graph G}

ALGORITHM
T ← empty
i traversal [1..(n-1)]
    e ← any minimum-weight edge in G
    {e must not form a simple circuit if added to T }
    T ← T added with e
```

C. *Graph Traversal Algorithm*

Graph traversal algorithms are methods used to visit every vertex and edge in a graph. These methods are necessary for various applications such as pathfinding, topological sorting, and in this context, maze solving. While the previous minimum spanning tree algorithms are used to generate the perfect maze structure, graph traversal algorithms provide ways to find the shortest path from the entrance to the exit point of the maze.

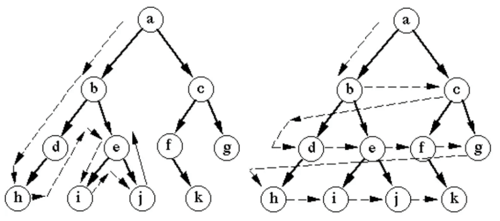

Figure 6. DFS and BFS Illustration. Adapted from [3].

1. Breadth-First Search

Breadth-first search explores all the neighboring vertices at current depth before it continues to the vertices at the next level.

2. Depth-First Search

Depth-first search explores start at a chosen starting vertex and explores as far as it can along each branch before going back to go to another branch. This approach makes it particularly intuitive for navigating maze structures.

III. Problem Definition and Solution

A. *Problem Definition*

A perfect maze is a specific type of complex passage network characterized by the inexistence of circuits and a unique path between any two points with no isolated paths. When formed as a graph, the structure of the maze simplifies certain computational problems. The main problem in this paper centers on the generation of perfect mazes and the shortest path solution available within the mazes. The algorithmic problems involve:

1. Generating mazes that stick to the definition of perfect maze.
2. Determining the shortest path available between two given points within the generated maze.

This paper focuses on analyzing and presenting algorithmic solutions for this problem. We will study how minimum spanning trees and graph traversal algorithms help in generating and solving a perfect maze.

B. *Problem Solution*

This section presents the algorithmic solutions that address the problems defined before. We use the principles of graph theory, trees, and specific minimum spanning tree algorithm and graph traversal algorithm to systematically generate these mazes and efficiently determine optimal solutions within them.

1) *Perfect Maze Generation*

The problem of generating a perfect maze is characterized by connectivity and the absence of circuits. This problem is effectively solved by generating a spanning tree from the initial grid graph. Each cell in the grid represents a vertex and the potential passages between adjacent cells represent edges. By selecting a subset of these edges that form a spanning tree, we guarantee that every cell is reachable from every other cell and that no redundant paths exist. We use Prim's algorithm for this maze generation.

The Randomized Prim's Algorithm solves the maze generation problem by iteratively growing the maze from an initial starting point basically adapts Prim's minimum spanning tree algorithm by changing minimum weight edge selection with random edge selection. The algorithm keeps a set of cells that is already part of the maze and a list of walls that connect a cell inside the maze to an unvisited cell outside it. Initially, an arbitrary starting cell is chosen and marked as part of the maze. All the walls that are adjacent to the starting cell are then added to the list of walls. In each step, the algorithm repeats by selecting an arbitrary wall from the list. The chosen wall connects a cell that is already part of the growing perfect maze to a cell that has yet to be visited. If the selected wall connects a visited to an unvisited cell, it will be removed and create a passage. The cell that is yet to be visited is then added to the set of maze cells. All adjacent walls from this new cell to its currently unvisited neighbors are then inserted to the list. This execution is iterated until the list is empty, indicating that all cells have been added to the maze, forming a single connected component without any circuits. The result is guaranteed to be a perfect maze.

2) Maze Solving

The complexity of solving a perfect maze is significantly reduced compared to general maze solving, we are guaranteed a unique path between any two points. We use breadth-first search for this problem.

Breadth-first search primary advantage in this problem is that it always finds the shortest path from the entrance to the exit. This happens because breadth-first search checks all possible paths at a certain distance before moving deeper. In a perfect maze that has a unique path between any two points and no circuits, breadth-first search reaches the exit point with the minimum number of steps.

IV. Implementation

The implementation of the problem solution is written in Python, chosen for its simplicity and extensive support for scientific computing and graphical visualization using its third-party packages NumPy and Matplotlib. I will present the algorithmic notation of each function created for the implementation. However, readers can access the full Python code implementation at Application of Minimum Spanning Trees and Graph Theory in Generating Perfect Mazes - IF1220.

A. Perfect Maze Generation

1. CreateGrid(w, h)

   This function initializes a two-dimensional grid $w \times h$ where each cell's value is set to 1. The width w and height h are set to be odd numbers to keep a consistent structure of walls and passages that is necessary for perfect maze.

```
function CreateGrid(w, h) → array [0..(h-1)] of array [0..(w-1)] of integer
{Specification: Initialize each cell's value with 1 and return it as a NumPy two-dimensional array.}

ALGORITHM
→ np.ones((h, w), dtype=int)
```

2. GetWalls(x, y, $grid$)

   This function fetches the list of walls two steps away in four cardinal directions (north, south, east, west) that have the possibility to connect to cells yet to be visited. For each wall, it records the intermediate wall's coordinate, which can be taken out to connect cells.

```
function GetWalls(x, y, grid) → array [0..(CAPACITY-1)] of tuple [0..3] of integer
{Specification: Returns a list of walls adjacent to (x, y) that lead to unvisited cells.}

ALGORITHM
directions ← [(-2, 0), (2, 0), (0, -2), (0, 2)]
walls ← []

for (i, j) in directions do
    xf ← x + i
    yf ← y + j
    if (0 < xf < grid.shape[1]) and (0 < yf < grid.shape[0]) and (grid[yf][xf] = 1) then
        xt ← x + i // 2
        yt ← y + j // 2
        walls.append((xf, yf, xt, yt))
→ walls
```

3. GenerateMaze(w, h)

   The main function for the Randomized Prim's Algorithm. The steps are:

   a. Initializes the grid by calling CreateGrid() function.
   b. Chooses an arbitrary starting point using Python's random library.
   c. Extends the maze by iteratively choosing and removing walls that connect visited and unvisited cells.
   d. Guarantees the result is a perfect maze.
   e. Inserts an entrance and exit to the maze.

```
function GenerateMaze(w, h) → array [0..(h-1)] of array [0..(w-1)] of integer, tuple [0..1] of integer, tuple [0..1] of integer
{Specification: Generate a perfect maze and returns its lists, entrance coordinate, and exit coordinate}

ALGORITHM
grid ← CreateGrid(w, h)
xs ← random.randrange(1, w, 2)
ys ← random.randrange(1, h, 2)
grid[ys][xs] ← 0
walls ← GetWalls(xs, ys, grid)
entrance ← (0, 0)
exit ← (0, 0)

while (walls) do
  wall ← random.choice(walls)
  x1, y1, x2, y2 ← wall
  walls.remove(wall)
  if (grid[y1][x1] = 1) then
    grid[y1][x1] ← 0
    grid[y2][x2] ← 0
    walls.extend(GetWalls(x1, y1, grid))
```

```
x ← 1
grid[0][x] ← 0
entrance ← (x, 0)

x ← w - 2
grid[h-1][x] ← 0
exit ← (x, h-1)

→ grid, entrance, exit
```

B. Maze Solving

1. SolveMaze(*maze*, *entrance*, *exit*)

   This function is used to solve the generated maze using breadth-first search approach to find the shortest path solution.

```
function SolveMaze(maze, entrance, exit) → array [0..CAPACITY] of tuple [0..1] of integer
{Specification: Solve the generated perfect maze using breadth-first search and return an array of tuple coordinates of the path}

ALGORITHM
start ← (entrance[0], entrance[1])
end ← (exit[0], exit [1])
queue ← deque([(start, [start])])
visited ← set([start])
directions ← [(-1, 0), (1, 0), (0, -1), (0, 1)]

while (queue) do
    (y, x), path ← queue.popleft()
    if ((x, y) = (end[1], end[0])) then
        → [(p[1], p[0]) for p in path]
    for dy, dx in directions do
        ny ← y + dy
        nx ← x + dx
        if (0 <= nx < maze.shape[1] and 0 <= ny < maze.shape[0] and maze[ny][nx] == 0 and (ny, nx) not in visited) then
            visited.add((ny, nx))
            queue.append(((ny, nx), path + [(ny, nx)]))

→ None
```

C. Visualization

1. PlotMaze(*maze*)

   This function is used to plot the maze using Python's third-party library Matplotlib.

```
procedure PlotMaze(input maze: array [0..(h-1)] of array [0..(w-1)] of integer)
{Specification: Display generated perfect maze using PyPlot from Matplotlib library}

ALGORITHM
  plt.figure(figsize=(10, 10))
  plt.imshow(maze, cmap='binary')
  plt.axis('off')
  plt.title('Generated Perfect Maze')
  plt.show()
```

2. PlotSolvedMaze(*maze*, *path*)

   This function is used to plot the maze with its solution using Python's third-party library Matplotlib.

```
procedure PlotSolvedMaze(input maze: array [0..(h-1)] of array [0..(w-1)] of integer, input path: array [0..CAPACITY] of tuple [0..1] of integer)
{Specification: Display generated perfect maze with its solved path using PyPlot from Matplotlib library}

ALGORITHM
  plt.figure(figsize=(10, 10))
  plt.imshow(maze, cmap='binary')

  if (path) then
     h, w = maze.shape
     yc, xc ← zip(*[(y, x) for (x, y) in path])
     plt.plot(xc, yc, color='red', linewidth=4)

  plt.axis('off')
  plt.title('Generated Perfect Maze with Solution')
  plt.show()
```

3. ShowMaze()

   This function is used to run the overall program by calling necessary functions and procedures.

```
procedure ShowMaze(input l: integer, input p: integer)
{Specification: Call the necessary functions and procedures to run the program.}

ALGORITHM
output("Masukkan lebar maze:")
```

```
input(l)
output(“Masukkan Panjang maze:”)
input(p)
if (l mod 2 = 0) then
  l ← l + 1
if (p mod 2 = 0) then
  p ← p + 1
maze, entrance, exit ← GenerateMaze(p, l)
PlotMaze(maze)
path ← SolveMaze(maze, entrance, exit)
PlotSolvedMaze(maze, path)
```

V. Results

Given the problem defined before, by running the Python program we made, we get:

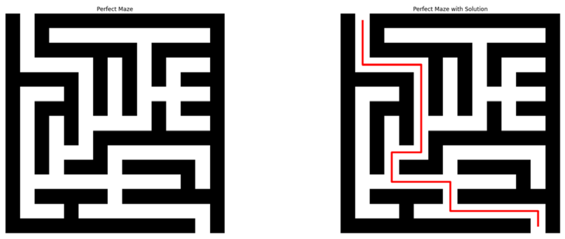

Figure 7. Generated 15x15 Perfect Maze with Its Solution. Adapted from author's implementation.

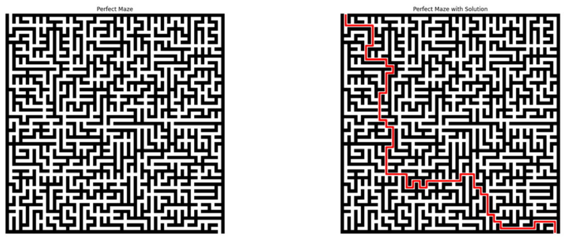

Figure 8. Generated 65x65 Perfect Maze with Its Solution. Adapted from author's implementation.

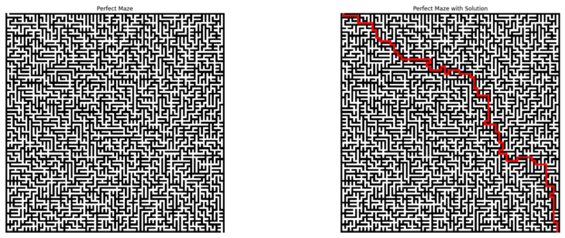

Figure 9. Generated 121x121 Perfect Maze with Its Solution. Adapted from author's implementation.

The plotting of the maze is obtained from the randomly generated maze’s cells values by using Python’s third-party library Matplotlib using its PyPlot state-based interface. For example, the figure seven on this section has the following coordinate and solution.

```
[[1 0 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 0 1 0 0 0 0 0 0 0 0 0 0 0 1]
 [1 0 1 1 1 1 1 1 1 0 1 1 1 1 1]
 [1 0 0 0 0 0 1 0 1 0 1 0 0 0 1]
 [1 1 1 1 1 0 1 0 1 0 1 0 1 1 1]
 [1 0 0 0 1 0 1 0 1 0 0 0 0 0 1]
 [1 1 1 1 1 0 1 0 1 0 1 0 1 1 1]
 [1 0 0 0 1 0 1 0 1 0 0 0 0 0 1]
 [1 0 1 1 1 0 1 0 1 0 1 0 1 1 1]
 [1 0 1 0 1 0 0 0 0 0 1 0 0 0 1]
 [1 0 1 0 1 0 1 1 1 1 1 1 1 1 1]
 [1 0 1 0 0 0 1 0 0 0 0 0 0 0 1]
 [1 0 1 0 1 1 1 0 1 1 1 1 1 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 0 1 0 1]
 [1 0 1 1 1 1 1 0 1 1 1 1 1 1 1]
 [1 0 0 0 1 0 0 0 0 0 0 0 0 0 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 0 1]]
```

```
[(1, 0), (1, 1), (1, 2), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3),
(5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9), (4, 9), (3, 9),
(3, 10), (3, 11), (4, 11), (5, 11), (6, 11), (7, 11), (7, 12),
(7, 13), (8, 13), (9, 13), (10, 13), (11, 13), (12, 13),
(13, 13), (13, 14)]
```

The results shown in this section prove that the implementation succeed in generating and solving randomized perfect mazes. Each maze is generated such that it contains connectivity and no circuits, sticking to the definition of a perfect maze. The results also prove the functionality of the Python implementation for both the perfect maze generation and solving.

VI. Conclusion

This paper presented an implementation of perfect maze generation with its solving by using graph theory, specifically minimum spanning trees and graph traversal algorithm. A perfect maze, characterized by a unique path between any two points and the inexistence of circuits, was successfully modeled as a spanning tree of a grid-shaped graph. The Prim’s algorithm was utilized to generate the maze structure, guaranteeing inexistence of circuits and connectivity. Breadth-first search basic approach was also utilized to solve the perfect maze with the shortest path solution. Due to the properties of a perfect maze, breadth-first search is proven efficient, as it guarantees the shortest path solution without the necessity for circuit detection.

The implementation in Python effectively demonstrated the generation and solution of the perfect maze with the help of its extensive third-party libraries NumPy and Matplotlib. The combination of Randomized Prim’s algorithm and breadth-first search produces correct and efficient results with clear example of how tree structures and graph theory apply in maze generation problem.

VII. Appendix

Code Implementation: Application of Minimum Spanning Trees and Graph Theory in Generating Perfect Mazes - IF1220

YouTube link: Application of Minimum Spanning Trees and Graph Theory in Generating Perfect Mazes
Duration: 6 minutes 38 seconds

Acknowledgment

The author sincerely thanks the Almighty God for providing strength and opportunity to write and finish this paper successfully. The author would also like to express his sincerest gratitude for Mr. Arrival Dwi Sentosa, S.Kom., M.T., lecturer of the Discrete Mathematics class, for his guidance, teachings, and support throughout the semester. The author gratefully acknowledges the inspiration provided by Mr. Jamis Buck through his blog for his insightful method about maze generation, which helped the author in modifying minimum spanning trees algorithm and graph traversal algorithm for the program implementation. Lastly, the author extends heartfelt appreciation to his beloved parents for their unwavering support, encouragement, and prayers, which have been a constant source of strength throughout the author's academic journey.

References

[1] K. H. Rosen and K. Krithivasan, *Discrete mathematics and its applications*, vol. 6. McGraw-Hill New York, 1999.

[2] Kjpargeter, "Abstract maze background in black and white," Freepik, [Online]. Available: https://www.freepik.com/free-vector/abstract-maze-background-black-white_155988287.htm. [Accessed: June 15, 2025].

[3] A. Mehra, "Difference between BFS and DFS: a comprehensive guide," Medium, 2024. [Online]. Available: https://medium.com/@sohel.indianreveler/difference-between-bfs-and-dfs-a-comprehensive-guide-a77b934470fe. [Accessed: June 15, 2025].

[4] J. Buck, "Maze generation: Prim's algorithm," The Buckblog, 2011. [Online]. Available: https://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm. [Accessed: June 15, 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, 18 Juni 2025

[illegible]

Jason Edward Salim - 13524034