Implementation of Dynamic Weighted Graphs Based on User Preferences in an Interactive Web Application for Video Game Recommendations

Hugo Daniel Johansen Napitupulu - 13525049
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: hugodjna70ok@gmail.com, 13525049@std.stei.itb.ac.id

***Abstract*— This paper presents a game recommendation system built using a Complete Directed Graph consisting of 50 distinct video games and 2,450 persistent edges. The system utilizes discrete mathematics concepts to calculate a precise similarity factor between games. To optimize content discovery, we implemented a dynamic weight mutation system based on user preferences. When a user likes a game, the backend automatically promotes similar games and suppresses unrelated ones. Conversely, when a user dislikes a game, the system executes the inverse operation of suppressing similar games and uniformly boosting alternative options.**

Keywords*— *Recommendation System, Directed Graph, Weighted Graph, Adjacency Matrix, Video Games, Complete Graph

I. Introduction

As the rapid growth of the video game industry invites more creative ideas into the market, video games have become very diverse in terms of genres. This phenomenon of genres and attributes create an environment in which customers are overwhelmed with the choices presented among them. To mitigate this, developers have implemented recommendation systems which have become a crucial component integrated in modern digital platforms.

In real world implementations, recommender systems utilize the analysis of user historical data and behaviour. These systems filter millions of options using collaborative filtering, content-based filtering, and hybrid models [1]. These systems have been designed to intake millions of data in order to output correct recommendations in a decent time-complexity.

In a purely discrete mathematical sense, it is entirely possible to create a recommendation system utilizing concepts of graph theory. Game entities can be modeled as a set of vertices, while the similarity of certain attributes can be mapped as a set of edges. This paper will delve into an implementation of said recommendation system utilizing a dynamically weighted graph.

II. Theoretical Basis

A. Graph Definition

A graph can be defined as a representation of discrete objects and their relationships with other discrete objects. A graph is defined by the vertices that makes it up and a set of edges connecting a pair of vertices. Mathematically, graph G is defined as:

$$G = (V, E)$$

In this case, V is a non-empty set of vertices, while E is a set of edges connecting a pair of vertices that is allowed to be empty.

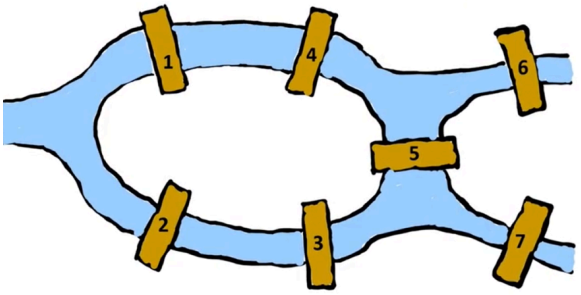

Fig 1. The Seven Bridges of Königsberg [2]

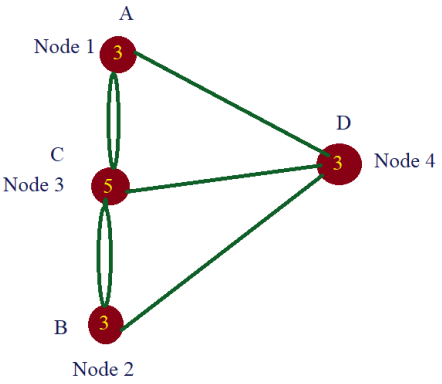

Fig 2. Representasi Königsberg Bridges Sebagai Graf [2]

The Konigsberg bridges problem is a prime example of translating discrete objects into graphs.

B. Complete Graph

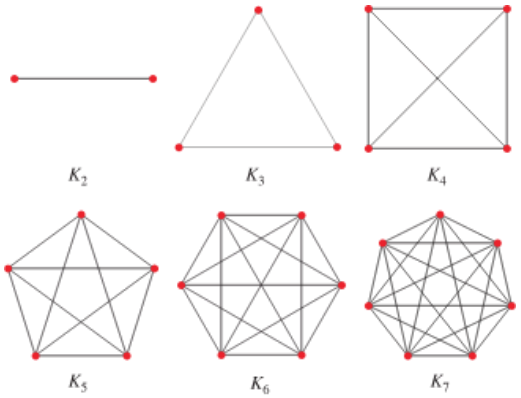

Fig 3. Complete Graphs [3]

A complete graph is a type of graph in which each pair of its vertices are connected by an edge. A complete graph with n vertices is referred as K_n

C. Weighted Graph

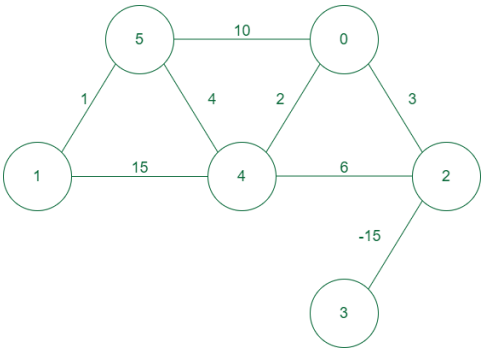

Fig 4. Weighted Graph [4]

A weighted graph is a type of graph in which edges are assigned values, called weights. Weights usually represent cost, distance, or other measuring units.

Compared to non-weighted graphs, weighted graphs provide a more flexible analysis between two vertices that provides insights into the structure of a graph and the relationships between its vertices.

D. Directed Graph

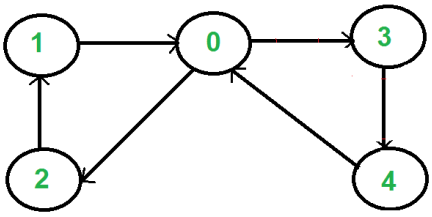

Fig 5. Directed Graph [5]

A directed graph is a type of graph in which edges have a direction associated with them. Directed graphs allow to model more complex relationships which can be useful when directionality is important.

E. Adjacency Matrix

An adjacency matrix represents a graph in a matrix form. In the context of a directed graph, edges have a direction associated with them, meaning the adjacency will not always be symmetric.

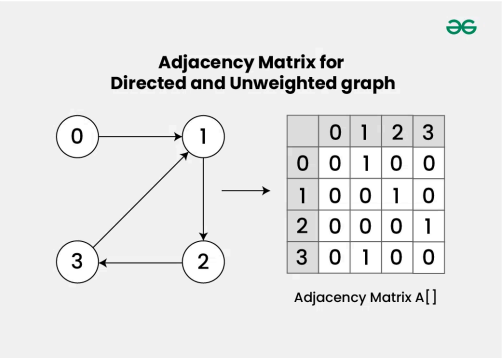

Fig 6. Adjacency Matrix For Directed Unweighted Graph [6]

Consider a graph with N vertices, the adjacency matrix A is defined as a N x N matrix where A[i][j] = 1 if a directed edge from vertex i to vertex j exists, otherwise A[i][j] = 0

In the context of a directed and weighted graph, the values of A[i][j] = W, in which W is the weight from vertex i to vertex j.

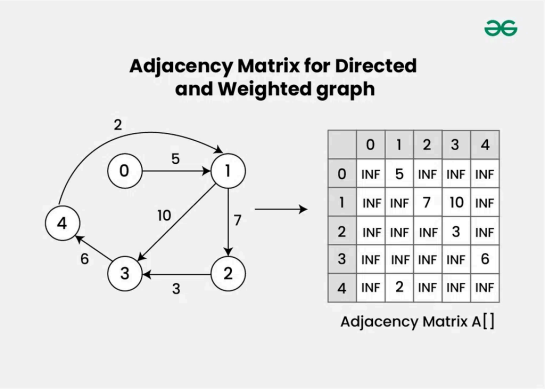

Fig 7. Adjacency Matrix For Directed Weighted Graph [6]

III. IMPLEMENTATION

A. Implemantation Architecture

The implementation is organized as a repository containing distinct frontend and backend services. The system utilizes TypeScript across both environments. The backend makes use of a Prisma schema to create a concise model for game and edge nodes, which will be stored in a SQLite Database, while

the frontend acts as a presentation for the database. The codebase is structured in the following hierarchy:

```
game-graph-recommender/
├── frontend/            # React Client
│   ├── src/
│   │   ├── App.tsx      # Main UI & Matrix Renderer
│   │   └── main.tsx
│   ├── vite.config.ts
│   └── package.json
├── backend/             # Express API Server
│   ├── src/
│   │   └── index.ts     # Graph API Controllers & Server entry
│   ├── prisma/
│   │   ├── schema.prisma # Graph Edge Database Schema
│   │   └── seed.ts      # N * (N-1) Complete Graph Seeding Engine
│   └── package.json
├── .gitignore           # Keeps your database & secrets safe
└── README.md            # You are here!
```

Fig 8. Folder Structure Of Codebase (Author Documentation)

B. Graph Components and System Constraints

To translate the abstract discrete structures into the system, the recommendation system maps graph components directly to relational database schemas. The system environment is formally bounded by the following data entities and design constraints:

1. Vertex Modeling (Game Entities)

Each video game is modeled as a discrete vertex v. v is structurally defined by a tuple containing intrinsic metadata attributes:

$$v = (id, title, genres, developer)$$

where id is a unique relational primary key, $title$ is a unique string descriptor, $genres$ represents a string of categorical attributes, and $developer$ acts as a single string tracking authorship.

```
model Game {
  id        Int    @id @default(autoincrement())
  title     String
  genre     String // genre may be filled with many
     genres seperated by a comma for e.g. "Action,
     RPG, Rougelike"
  developer String


  outgoingEdges Edge[] @relation("SourceGame")
  incomingEdges Edge[] @relation("TargetGame")
}
```

2. Edge Modeling

Edges are structural components mapping the relative recommendation strength between pairs of vertices. An edge e is defined as an ordered triplet:

$$e = (v_{source}, v_{target}, w)$$

where v_{source} represents the source game ID, v_{target} represents the destination game ID, and $w \in [0.0, 3.0]$ represents the edge weight.

The system separates edges into incoming and outgoing to reflect the asymmetry of user behaviour. For example, if a user interacts positively with Game A, the engine increases the weight of the outgoing edge directed from Game A toward Game B, making Game B highly recommended whenever a user views Game A. However, if the user dislikes Game B, the system debuffs the outgoing edge from Game B toward Game A.

```
model Edge {
  id       Int   @id @default(autoincrement())
  sourceId Int
  targetId Int
  weight   Float


  sourceGame Game @relation("SourceGame", fields:
     [sourceId], references: [id], onDelete: Cascade)
  targetGame Game @relation("TargetGame", fields:
     [targetId], references: [id], onDelete: Cascade)


  @@unique(name: "graphLink", fields: [sourceId,
     targetId])
}
```

3. The database of the recommendation system is seeded with a static catalog of 50 distinct video games. This results in a complete directed graph K_{50} with 2450 edges.

C. Graph Initialization

Each game is connected to one another with a fixed default baseline weight of 0.20. This weight will mutate based on user inputs of liking or disliking a game.

```
    let baseWeight = 0.20;


    for (const source of games) {
        for (const target of games) {
            if (source.id !== target.id) {
                await prisma.edge.create({
                data: {
                    sourceId: source.id,
                    targetId: target.id,
```

```
                    weight: baseWeight
                }
                });
            }
        }
    }
```

D. Edge Mutation Implementation

The backend exposes a REST endpoint (POST /api/interact). This implementation makes use of attributes in the game nodes to calculate how similar one game is to another. The author uses a custom function to calculate a similarity score that takes into account the game's genres and if they share the same developer.

```
   function calculateGenreSimilarity(genreA: string, genreB: string, devA:string, devB: string): number {
       const tagsA = genreA.split(",").map(t => t.trim().toLowerCase());
       const tagsB = genreB.split(",").map(t => t.trim().toLowerCase());
       const genreIntersection = tagsA.filter(t => tagsB.includes(t)).length;
       const sameDev = devA === devB;
       let score = 0.0;
       if (genreIntersection === 0){
           score = 0.1;
       } else {
           score = 0.5 + (genreIntersection * 0.25)
       }
       if (sameDev){
           score += 0.5;
       }
       return score;
   }
```

When a user executes an interaction on a specific game, the mutation logic executes. The engine will retrieve the target game node data along with its outgoing edges from the database. The genres are tokenized and trimmed to form valid sets of text strings which will be filtered to compute the cardinality of the genre tag intersection. If an intersection of genres is empty, a default baseline score of 0.1 is given. Otherwise, a base score of 0.5 is given and incremented linearly by 0.25 for each intersecting tag. Lastly, the developer strings are evaluated via a strict equality relation. An additional 0.5 bonus is given if the equality is true.

To understand what games to recommend and what not to, the calculated similarity factor or its inverse unsimilarity factor will be multiplied with a factor based on whether the user likes or dislikes the game alongside a threshold condition. The implementation uses a likeIncrement of 0.30 and a dislikeDecrement of 0.40. A heavier factor for disliking is implemented to ensure the recommendation system aggressively suppresses games similar to the one disliked by the user, while a balancing push-and-pull mechanism uses the unsimmilarityFactor to unpromote non-similar games on a like and surface alternative options on a dislike.

```
            const simmilarityFactor = calculateGenreSimilarity(sourceGame.genre, edge.targetGame.genre, sourceGame.developer, edge.targetGame.developer);
            const unsimmilarityFactor = 1.0 - simmilarityFactor;

            let newWeight = edge.weight;

            if (type === "LIKE"){
                if (simmilarityFactor >= simmilarityThreshold){
                    const buff = likeIncrement * simmilarityFactor;
                    newWeight = Math.min(3.0, Number((edge.weight + buff).toFixed(2)));
                } else {
                    const debuff = dislikeDecrement * unsimmilarityFactor;
                    newWeight = Math.max(0.0, Number((edge.weight - debuff).toFixed(2)));
                }
            }
            else{
                if (simmilarityFactor >= simmilarityThreshold){
                    const debuff = dislikeDecrement * simmilarityFactor;
```

```
                    newWeight = Math.max(0.0,
Number((edge.weight - debuff).toFixed(2)));

                } else {

                    const buff = likeIncrement *
unsimmilarityFactor;

                    newWeight = Math.min(3.0,
Number((edge.weight + buff).toFixed(2)));

                }

            }
```

A promotion will push adjacent edge weights upward toward a maximum ceiling 3.0, while a suppression will suppress weights to a minimum floor of 0.0.

E. Recommendation Implementation

```
app.get("/api/recommend/:gameId", async (req, res) =>
{
    const gameId = Number(req.params.gameId);

    try {
        const recs = await prisma.edge.findMany({
            where: {
                sourceId: gameId,
                weight: { gte: 0.40 }
            },
            include: {
                targetGame: true,
            },
            orderBy: {
                weight: "desc",
            },
        });

        return res.json(recs);
    } catch (error) {
        console.log("Failed to retrieve
recommendations");
        return res.status(500).json({ error: "Failed
to retrieve recommendations" });
    }
});
```

When the user is interacting with a game, the game's id is treated as the source vertex and evaluates its outgoing edges to deliver a ranked list of items. This implementation uses a fixed filter weight of 0.40 to differentiate which games are targets of recommendation.

F. Web App

The presentation layer of the recommendation system is realized using a dedicated web application built with React. This frontend application functions as a dashboard for the graph structure.

1. Dedicated Game Section

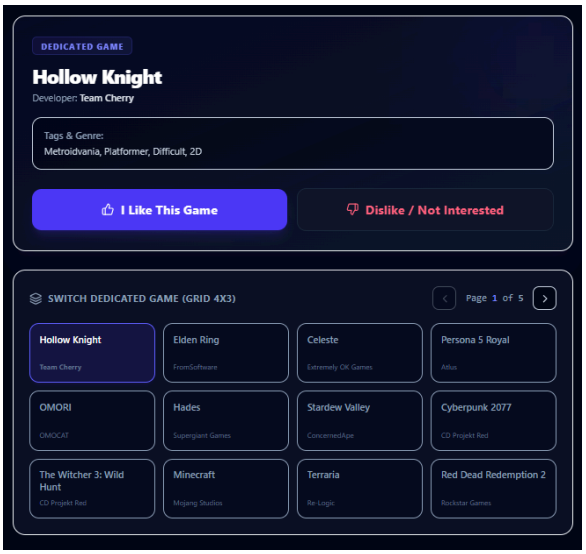

Fig 9. Dedicated Game Section (Author Documentation)

To simulate user interactions, the user may like or dislike a game in the dedicated game section. Users may also select a different game to focus on in the pagination section.

2. Recommendation Section

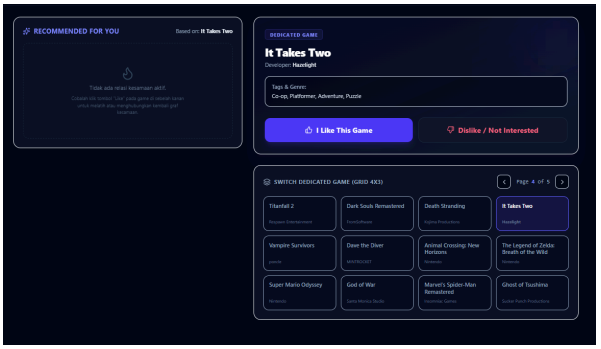

Fig 10. Empty Recommendation Section (Author Documentation)

The recommendation section will showcase every game connected with the current game in the dedicated game section that has a weight greater than or equal to 0.40.

Every interaction (LIKE / DISLIKE) will mutate the weights and update the recommendation section to reflect the new graph representation.

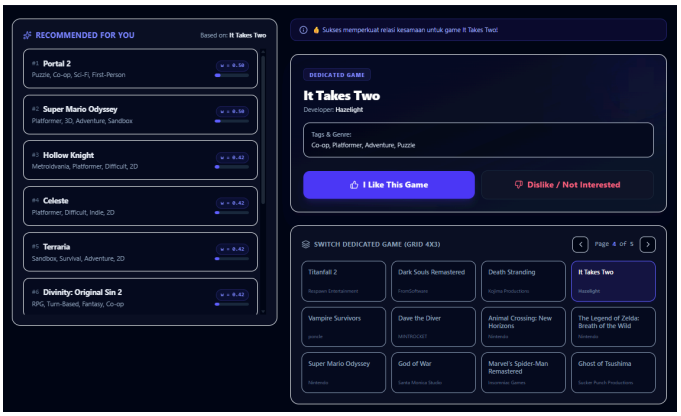

Fig 11. Recommendation Section After a Like Mutation (Author Documentation)

3. Adjacency Matrix Visualization

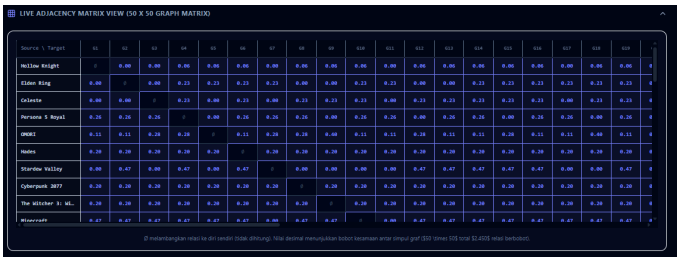

Fig 12. Adjacency Matrix of The Graph (Author Documentation)

To visualize the graph, the web app provides a 50 x 50 adjacency matrix. This matrix will update with every interaction.

IV. Testing and Observation

A. *Test Enviornment*

To test the program, the database was first seeded with a set of 50 distinct video games for each test case This ensures that each edge has a baseline weight of 0.20.

The target node chosen for this test is the game node Elden Ring. This game is developed by the developer FromSoftware and includes genres of Souls-like, Open World, Dark Fantasy, Difficult.

B. *Observations*

1. Test Case 1: LIKE

Upon liking Elden Ring, we are presented with these recommendations:

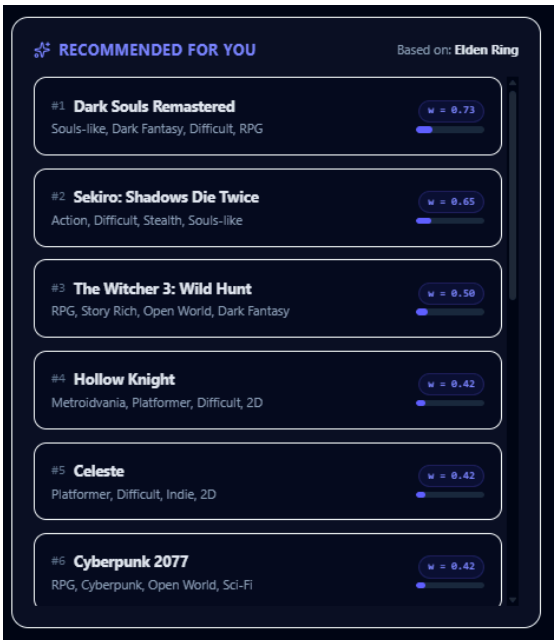

Fig 13. Recommendations of Elden Ring in Test Case 1 (Author Documentation)

The top recommendation is Dark Souls Remastered with a weight of 0.73, followed by Sekiro: Shadows Die Twice with a weight of 0.65, and The Witcher 3: Wild Hunt with a weight of 0.50. The remaining 11 recommendations all share a weight of 0.42.

Dark Souls Remastered yields the highest weight because it shares three genre intersections and the same developer as Elden Ring. Sekiro: Shadows Die Twice also shares the same developer but only features two genre intersections. Meanwhile, The Witcher 3: Wild Hunt matches two genre intersections but lacks a shared developer. The remaining 11 games only yield a single genre intersection.

2. Test Case 1: DISLIKE

Upon triggering a "Dislike" action on Elden Ring, the system presents 35 games, all sharing an identical weight of 0.47. These 35 games represent the inverse of the recommendation logic, as none of them appeared in the previous "Like" test case. Due to how the algorithm calculates dissimilarity, every non-matching game is assigned the exact same weight.

C. *Analysis*

The test results demonstrate that the program is fully capable of generating logical, attribute-driven recommendations by manipulating edge weights based on explicit data intersections, specifically genres and developers. In Test Case 1, the system correctly prioritized Dark Souls Remastered with a resulting weight of 0.73 and Sekiro: Shadows Die Twice with a weight of 0.65 because they share both a common developer and multiple genre intersections with Elden Ring.

However, while the system succeeds in basic content-based matching, the test data reveals limitations in how the program handles negative feedback. In Test Case 2, the system assigned an identical weight of 0.47 to 35 different games. In a real-world scenario, a user who dislikes Elden Ring might still like a game like The Witcher 3: Wild Hunt due to its narrative focus, even if they dislike the "Souls-like" aspects of the game. The program does not take into account this underlying nuance, which creates a static output that treats all non-matching games as a uniform block.

V. Conclusion

In this paper, the author demonstrates a game recommendation engine by using discrete mathematics concepts. The engine is modeled as a complete directed graph (K_{50}) containing 2450 persistent edges. The system is able to generate attribute-driven recommendations by dynamically manipulating the weight of the graph's edges based on intersections.

GitHub Repository Link

https://github.com/ladubiosacreatura/game-graph-recommender

Acknowledgment

The author would like to give thanks to God for His guidance in the process of writing this paper, Prof. Dr. Ir. Rinaldi, M.T. for sharing his knowledge in the world of discrete mathematics, and the author's family and friends for their continuous support during the writing of this paper.

References

[1] [1] IBM, "What is a recommendation engine?," *IBM*, n.d. [Online]. Available: https://www.ibm.com/think/topics/recommendation-engine. [Accessed: June 19, 2026].

[2] H. Jayaraman, "Königsberg Bridge Problem and the Evolution of Mathematics," *Lancaster University STOR-i Student Sites*, n.d. [Online]. Available: https://www.lancaster.ac.uk/stor-i-student-sites/harini-jayaraman/konigsberg-bridge-problem-and-the-evolution-of-mathematics/. [Accessed: June 17, 2026].

[3] E. W. Weisstein, "Complete Graph," *Wolfram MathWorld*, n.d. [Online]. Available: https://mathworld.wolfram.com/CompleteGraph.html. [Accessed: June 17, 2026].

[4] "Weighted Graph," *ScienceDirect Topics*, n.d. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/weighted-graph. [Accessed: June 17, 2026].

[5] GeeksforGeeks, "What is Directed Graph? Directed Graph Meaning," *GeeksforGeeks*, n.d. [Online]. Available: https://www.geeksforgeeks.org/dsa/what-is-directed-graph-directed-graph-meaning/. [Accessed: June 17, 2026].

[6] GeeksforGeeks, "Adjacency Matrix of Directed Graph," *GeeksforGeeks*, n.d. [Online]. Available: https://www.geeksforgeeks.org/dsa/adjacency-matrix-of-directed-graph/. [Accessed: June 17, 2026].

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

Hugo Daniel Johansen Napitupulu 13525049