Implementation of Multiple Searching Algorithms to Search for a Block That May or May Not Exist Within a Certain Area in Minecraft

I Gusti Ngurah Alit Dharma Yudha - 13524072
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: adharmayudha@gmail.com , 13524072@std.stei.itb.ac.id

***Abstract*—This paper implements and compares multiple search algorithms within the Minecraft game environment. We evaluate Brute Force, Greedy, BFS, DFS, and Divide and Conquer algorithms for locating specific block types within a single chunk. Performance metrics, including execution time and blocks checked, are collected and analyzed to determine the most efficient approach for this constrained search problem.**

Keywords—Search Algorithms, Brute Force, Greedy, Divide and Conquer, BFS, DFS, Performance Comparison, Minecraft

I. Introduction

Minecraft is a widely popular sandbox video game that allows players to explore, build, and interact within a procedurally generated 3D world composed entirely of blocks. The game world is divided into chunks, which are 16×16 block areas extending from the world's minimum height (Y = -64) to its maximum height (Y = 320), resulting in a total of 16 × 16 × 384 = 98,304 blocks per chunk. Players can mine various types of blocks, such as ores, stone, grass blocks, and so on which are distributed according to specific world-generation rules [4]. The java version of the game (which is what we're going to focus on) has around 800 unique blocks.

Imagine a case when a player has built a large structure, then they want to remove or replace a certain block in that build. Normally the player would have to look at each block one at a time and eventually remove all the blocks desired. Maybe another case, where a player just wants to cheat and find certain ores in their vicinity to get resources quickly. Despite the game's extensive features, Minecraft does not natively provide a built-in function to locate a specific block type within a chunk. Players must manually explore and mine areas to find desired blocks, which can be time-consuming and inefficient, especially when searching for rare ores like diamonds or emeralds.

Searching algorithms are the cornerstone of computer science, providing the foundation for solving complex optimization and retrieval problems. While theoretical analysis is essential, practical implementation and testing in varied environments offer deeper insights into algorithm performance [1]. This paper explores the application of several classic search algorithms within the unique constraints of a 3D, procedurally generated game world that is Minecraft.

The primary objective is to implement and empirically compare the performance of Brute Force, Greedy, Breadth-First Search (BFS), Depth-First Search (DFS), and Divide and Conquer algorithms. These algorithms are tasked with a specific problem: locating all instances of a target block type within a single Minecraft chunk. This size provides a consistent and non-trivial search space for the algorithms to handle.

This research is motivated by the need for efficient block-finding tools by modding a game and getting a good comparison between searching algorithms.

II. Literature Review

A. Brute Force

Brute Force, also known as exhaustive search, is the most straightforward approach to solving a problem. It involves systematically enumerating all possible candidates and checking each one against the solution criteria [1][2]. In the context of searching for a block within a Minecraft chunk, this means checking every single block in the chunk one by one until all blocks have been examined. For a standard chunk with 16 × 16 × 384 = 98,304 blocks, the brute force algorithm will perform exactly this many checks. Its complexity is O(n), where n is the number of blocks in the search space. While simple to implement and guaranteed to find all occurrences of the target block, brute force is not optimized for speed, especially when the search space is large or when the target block is rare. It serves as a useful baseline for comparing the efficiency of more sophisticated algorithms [2].

B. Greedy

The Greedy algorithmic paradigm is a heuristic approach commonly used for optimization problems. At each step, it makes the locally optimal choice with the hope that this will lead to a globally optimal solution [3]. Greedy algorithms do not always produce optimal solutions, but they often provide fast, acceptable approximations (pseudo-optimal solutions) [3].

For this block-searching problem, we implement a height-priority greedy strategy. This strategy leverages knowledge about the distribution of ores in the Minecraft world. Different ores have specific Y-level ranges where they are most commonly found [4]. For example:

- Coal is most abundant around Y = 96
- Iron has peaks at Y = 16 and Y = 232
- Diamond is most common at Y = -59, near the bottom of the world

The greedy algorithm prioritizes searching these known "best" Y-levels first, checking all blocks at those heights before moving to less promising levels. By doing so, it aims to find target blocks more quickly than brute force, reducing the number of blocks that need to be checked. However, because it may stop searching once a block is found (or after exhausting the priority ranges), it does not guarantee finding all instances of the target block, making it suitable for scenarios where finding any occurrence is sufficient.

C. Divide and Conquer

Divide and Conquer is a powerful algorithmic paradigm that solves a problem by recursively breaking it down into smaller, more manageable subproblems of the same type, solving each subproblem, and then combining their solutions [5]. The general approach consists of three steps:

- Divide: Split the problem into smaller subproblems.
- Conquer: Solve each subproblem recursively. If a subproblem is small enough, solve it directly (the base case).
- Combine: Merge the solutions of the subproblems to form the solution to the original problem.

For our block-searching implementation, we adapt this approach by recursively subdividing the 3D chunk into smaller regions called octants. The chunk is divided into eight equal sub-regions, and this process continues until a region is small enough (e.g., 4×4×4 = 64 blocks). At this base case, a linear search is performed to check all blocks in that region. The results from all sub-regions are then combined to produce the complete set of found block positions. This approach is similar to the quadtree subdivision used in spatial indexing and has a complexity of O(n log n) for balanced partitions [5].

D. DFS and BFS

Breadth-First Search (BFS) and Depth-First Search (DFS) are fundamental graph traversal algorithms widely used for searching and pathfinding [6]. While they are traditionally applied to graphs, they can be adapted to traverse the 3D grid of blocks in a Minecraft chunk by treating each block as a node and the adjacency relationships between neighboring blocks as edges.

- Breadth-First Search (BFS)

  BFS explores a graph by visiting all neighboring nodes at the current depth level before moving on to nodes at the next depth level [6]. In our implementation, BFS searches the chunk by exploring blocks in a 3D grid, expanding outward from a starting position (e.g., the center of the chunk at Y = 64). It uses a queue (FIFO) data structure to manage the order of exploration. BFS guarantees that all blocks at a given distance are checked before moving further out, ensuring that the search is thorough and systematic. The complexity of BFS is O(V + E), where V is the number of vertices (blocks) and E is the number of edges (adjacencies) [6].

- Depth-First Search (DFS)

  DFS, in contrast, explores as far as possible along a single branch of the graph before backtracking [6]. It uses a stack (LIFO) data structure, which can be implemented either recursively or iteratively. Our implementation uses an iterative DFS with an explicit stack to avoid stack overflow issues that can occur with recursive DFS on large graphs. Starting from a central position, DFS follows a path of adjacent blocks until it cannot go further, then backtracks to explore another path. This approach is memory-efficient compared to BFS but does not guarantee the shortest path or the most systematic coverage of the search space. The complexity of DFS is also O(V + E) [6].

Both BFS and DFS are well-suited for exploring the block grid, but their different traversal strategies may lead to different performance characteristics when searching for target blocks in a chunk. BFS tends to explore more broadly, while DFS goes deeper into the structure, which may be advantageous depending on the distribution of the target blocks.

III. Implementation

To make this mod the author used fabric as the Minecraft mod loader. The project structure used is available publicly by the folks over at Fabric [7]. This project is of course being implemented in java, as we are going to use Minecraft java edition version 26.1.2. The main project structure is as follows,

```
./
├─ /src
│  ├─ /main
│  │  └─ /java
│  │     └─ /net
│  │        └─ /sukabotol
│  │           └─ /finyoblock
│  │              ├─ /mixin
│  │              ├─ FindyoblockMod.java
│  │              ├─ SearchResult.java
│  │              └─ TheCommands.java
│  └─ /resources
├─ build.gradle
```

└ gradle.properties

The main build tool that has been provided by the project template is `gradle`, which can be seen by the `build.gradle` and the `gradle.properties` files. The resources directory in the src directory just contains configurations needed for the mod like the version, the name, etc.

The FindyoBlockMod.java file is the entrypoint of the whole mod and its whole job is to initialize the mod. As seen in this snippet below, all it does is initialize a logger and calls a method in the TheCommands.java that is "register".

```
public class FindyoblockMod implements
ModInitializer {
    public static final String MOD_ID =
"findyoblock";
    public static final Logger LOGGER =
LoggerFactory.getLogger(MOD_ID);

    @Override
    public void onInitialize() {
        LOGGER.info("Test");


CommandRegistrationCallback.EVENT.register((d
ispatcher, registryAccess, environment) -> {
            TheCommands.register(dispatcher);
        });
    }
}
```

The "TheCommands.java" is where everything happens. First of we have the "register" method mentioned above, this method is the one initializing/creating the command that we can use later in Minecraft. The beginning part looks like this roughly,

```
dispatcher.register(Commands.literal("findblo
cks")
.then(Commands.argument("mode",
StringArgumentType.word())
.suggests(MODE_SUGGESTIONS)
.then(Commands.argument("block",
StringArgumentType.greedyString())
.suggests(BLOCK_SUGGESTIONS)
.executes(context -> {
```

This block of code is basically creating a command that you can use in minecraft. So first it will create a command "/findblocks", then it accepts an argument of "mode", then and argument of "block". It also shows suggestions for "mode" and "block" for easier use. These suggestions are declared as "mode" having all the algorithms available and "block" having every block available in Minecraft.

The actual Implementation of the command is as follows,

```
Player player =
context.getSource().getPlayer();
if(player == null){
context.getSource().sendFailure(Component.lit
eral("Only players can use this command!"));
return 0;
}
String mode =
StringArgumentType.getString(context,
"mode").toLowerCase();
String target =
StringArgumentType.getString(context,
"block");
boolean valid = false;
for (Object block : BuiltInRegistries.BLOCK)
{
String currentBlockName =
BuiltInRegistries.BLOCK.getKey((net.minecraft
.world.level.block.Block) block).toString();
if (currentBlockName.equals(target)) {
valid = true;
break;
}
}
if (!valid) {
context.getSource().sendFailure(Component.lit
eral("§cInvalid block! Use TAB to see
available blocks."));
return 0;
}
```

What this snippet does is getting the player object, getting the Strings that the user entered in the command for the mode and the block. Then it validates the player, check if the entity running the command is an actual player. Because if it is not a player and for example the command is ran on a server console, the program would not work because it does not have a player position. Then it also validates the block in the user-entered String for the block, check if that is an actual block that exists in Minecraft.

The rest of it is just getting the actual player position, getting the world/dimension, calculating the chunk coordinates, running the searching algorithm, and lastly saving it into a text file. Some interesting parts of that, chunk coordinates are calculated by integer dividing the player's x and z coordinates by 16. Why? its because a chunk in

Minecraft is 16 blocks long and 16 blocks wide with an infinite height value. Getting the world/dimension is also important since each world can have the same coordinates values but different terrain/blocks.

Now we come to the algorithms, before that, Each searching algorithm has the same input and output. The input being the world/dimension, the chunk coordinates, and the target block. For the output, it is saved as the SearchResult object, the implementation is within the SearchResult.java and the attributes it holds are as follows,

```
public class SearchResult{
        private final List<BlockPos>
found_positions;
        private final int
total_blocks_checked;
        private final long
execution_time_nano;
        private final int found_count;
        private String algorithm;
```

Now let's start with the actual algorithms. The author will not provide much of the code snippets. The full source code will be available in the Github repository in references section. The snippet of the brute force implementation is as follows,

```
for (int x = 0; x < 16; x++) {
            for (int z = 0; z < 16; z++) {
                for (int y = world.getMinY();
y < world.getMaxY(); y++) {
                    blocksChecked++;
                    BlockPos pos = new
BlockPos(startx + x, y, startz + z);
                    String blockId =
BuiltInRegistries.BLOCK.getKey(world.getBlock
State(pos).getBlock()).toString();
                    if
(blockId.equals(target)) {
                        found.add(pos);
                    }
                }
            }
        }
```

What this algorithm does is systematically checking every single block within the chunk one by one. Before this snippet the method calculates the starting x and z axes, then uses three nested loops to iterate through all possible X positions (0 to 15), all possible Z positions (0 to 15), and all possible Y positions from the world's minimum height to maximum height. At each coordinate, it retrieves the block at that position, compares it against the target block, and if they match, it adds that position to the list of found blocks. Its behavior is predictable and its implementation straightforward, though it performs the maximum possible number of checks regardless of whether the target block is found early or is extremely rare.

Next is the implementation of the greedy algorithm. For this implementation there needs to be some definition for the heuristics, in this implementation we are only providing heuristics for ores as we are doing the heuristic by spawn height. Because in Minecraft, ores have a certain minimum and maximum height they can spawn at. So in the source code there are definitions of the maximum and minimum height of different ores, from that the greedy method iterates similarly to the brute force one but it only iterates through the maximum and minimum height of an ore, not of the world max and min height.

Next is the implementation of the divide and conquer algorithm. This algorithm recursively splits the chunk into smaller, more manageable three-dimensional regions until each region is small enough to search directly. This is similar to how you would iterate through an octree. It starts with the entire chunk defined by its minimum and maximum corners, then checks the size of the region. If the region is small, specifically 64 blocks or fewer, it performs a linear search by checking every block in that region directly. If the region is larger, it divides it into eight smaller regions called octants by finding the midpoints in the x, y, and z axes. It then recursively applies the same algorithm to each of these eight octants, collecting the results from all of them. Finally, it combines the results from all sub-regions into a single list of found positions.

Next is the implementation of the depth first search (DFS) algorithm. This algorithm treats the chunk as a three-dimensional graph but exploring it by going as deep as possible along one path before backtracking to explore another. It begins by placing the starting position, the center of the chunk at Y equals 64, onto a stack data structure and marking it as visited. While the stack is not empty, it pops the top position from the stack, checks if that block matches the target, and then pushes all six neighboring positions onto the stack, marking each as visited. This creates a "go deep first" behavior, where the algorithm follows a single path of adjacent blocks until it cannot go further, then backtracks to the previous branching point and explores another path. This memory-efficient approach uses only the stack to store the current path. However, it can get stuck exploring a long path before finding the target and may miss nearby blocks if they are in a different branch. The implementation uses an explicit stack rather than recursion to avoid stack overflow errors that could occur in a world with 384 blocks of height.

Next is the implementation of the breadth first search (BFS) algorithm. This algorithm also treats the chunk as a three-dimensional graph where each block is a node connected to its six neighbors: up, down, north, south, east, and west. It begins by placing the starting position, which is the center of

the chunk at Y equals 64, into a queue data structure and marking it as visited. While the queue is not empty, it removes the next position from the front of the queue, checks if that block matches the target, and then adds all six neighboring positions to the back of the queue, marking each as visited to prevent revisiting. This approach ensures that all blocks at the current distance are checked before moving further outward.

IV. Testing

There will be a few tests conducted the first is ensuring that the mod accurately counts the correct amount of target blocks and correctly ouputs its coordinates. For this test we will use this,

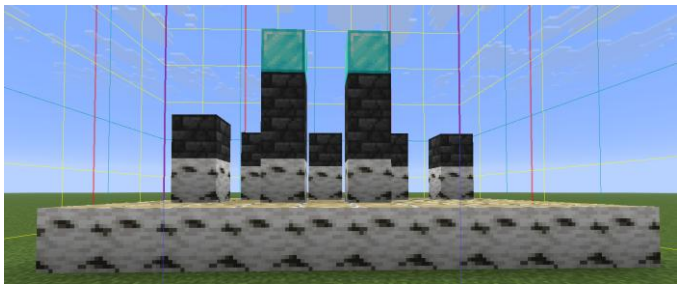

Fig. 1. Accuraccy test - testing grounds

This is set in a superflat world with 1 layer of grass blocks, 2 layers of dirt, and 1 layer of bedrock. Withing this chunk I have placed exactly 67 birch logs, 9 cracked deepslate tiles, and 2 diamond blocks.

Using the brute force method, divide and conquer, and bfs algorithms we get these results,

```
========================================
Search Results Report
========================================
Timestamp: 2026-06-20 00:41:01
Target Block: minecraft:birch_log
Mode: bruteforce
Chunk: (-2, -1)
Algorithm: Brute Force
Blocks Found: 67
Blocks Checked: 98048
Execution Time (ms): 7.508
```

Fig. 2. Count accuraccy test – brute force

```
========================================
Search Results Report
========================================
Timestamp: 2026-06-20 00:43:03
Target Block: minecraft:birch_log
Mode: divideconquer
Chunk: (-2, -1)
Algorithm: Divide & Conquer
Blocks Found: 67
Blocks Checked: 98048
Execution Time (ms): 16.503
```

Fig. 3. Count accuraccy test – divide and conquer

```
========================================
Search Results Report
========================================
Timestamp: 2026-06-20 00:43:10
Target Block: minecraft:birch_log
Mode: bfs
Chunk: (-2, -1)
Algorithm: BFS
Blocks Found: 67
Blocks Checked: 98048
Execution Time (ms): 74.041
```

Fig. 4. Count accuraccy test – bfs

Based on these results we can conclude that the algorithms can result the correct amount of blocks, next we check the block position,

```
========================================
Search Results Report
========================================
Timestamp: 2026-06-20 00:47:57
Target Block: minecraft:diamond_block
Mode: greedy
Chunk: (-2, -1)
Algorithm: Greedy
Blocks Found: 0
Blocks Checked: 4352
Execution Time (ms): 0.332

Found Positions: None
========================================
```

Fig. 5. Position accuraccy test – greedy

```
========================================
Search Results Report
========================================
Timestamp: 2026-06-20 00:48:07
Target Block: minecraft:diamond_block
Mode: bruteforce
Chunk: (-2, -1)
Algorithm: Brute Force
Blocks Found: 2
Blocks Checked: 98048
Execution Time (ms): 4.793

Found Positions:
  1. (-24, -56, -10)
  2. (-24, -56, -8)
========================================
```

Fig. 6. Position accuraccy test – brute force

```
========================================
Search Results Report
========================================
Timestamp: 2026-06-20 00:48:16
Target Block: minecraft:diamond_block
Mode: dfs
Chunk: (-2, -1)
Algorithm: DFS
Blocks Found: 2
Blocks Checked: 98048
Execution Time (ms): 53.167

Found Positions:
  1. (-24, -56, -10)
  2. (-24, -56, -8)
========================================
```

Fig. 7. Position accuraccy test – dfs

Based on these results we can compare the positions with the actual ones below,

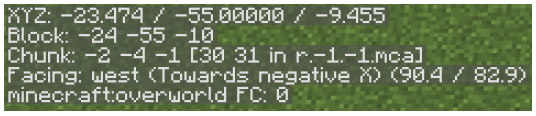

Fig. 8. Actual diamond position 1

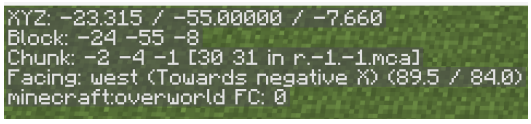

Fig. 9. Actual diamond position 2

The thing we need to look at are the "Block: " coordinates, so when we compare them with the results of the tests, we can conclude that they are accurate. As for the greedy method, this fails because in the implementation the method first compares the name of the block to get the max height and min height for serching. Because we had a diamond block, the method gets the substring diamond and sets the height range to where diamond ores spawn. This is already mentioned in the implementation part that the greedy method would only accurately work for ores.

Next is the speed comparison test, for this we will go to a normally generated world, go atop a mountain and search for coal ore. Here are the results,

Fig. 10. Speed test - testing grounds

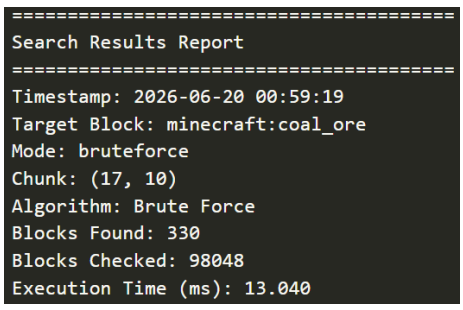

Fig. 11. Speed test – brute force

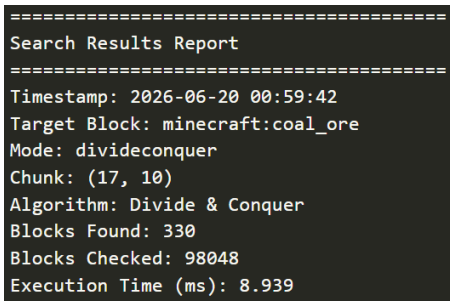

Fig. 12. Speed test – divide and cnquer

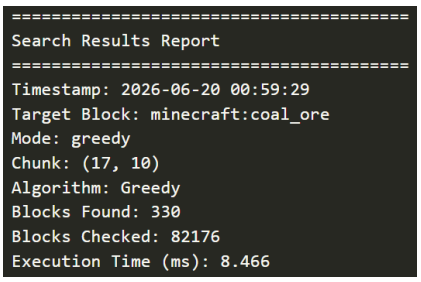

Fig. 13. Speed test – greedy

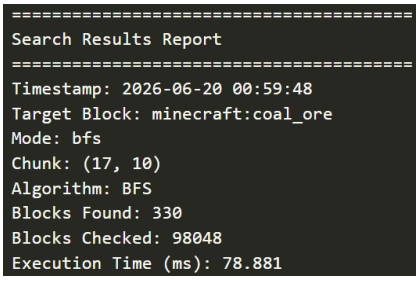

Fig. 14. Speed test – BFS

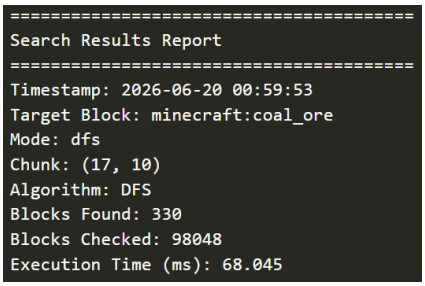

Fig. 15. Speed test – DFS

Based on these results we can conclude that for this case the fastest algorithms are the greedy and divide and conquer algorithms. With DFS being the slowest.

V. Conclusion

This paper has successfully implemented and compared five unique search algorithms (Brute Force, Greedy, Divide and Conquer, BFS, and DFS) for locating target blocks within a Minecraft chunk. The experimental results demonstrate that all algorithms correctly count and identify the exact positions of target blocks, validating the correctness of each implementation. Performance testing reveals that the Greedy algorithm, which leverages height-priority heuristics based on known ore distribution patterns, performs exceptionally fast for ore blocks but sacrifices completeness by not guaranteeing the discovery of all occurrences. The Divide and Conquer algorithm proves equally fast while maintaining completeness, making it the most balanced approach for general block searching. BFS and DFS, while complete, exhibit slower performance due to their graph traversal overhead and the use of data structures such as queues and visited sets. Notably, DFS was observed to be the slowest among the tested algorithms. The integration of these algorithms into a Minecraft mod provides a practical and engaging testbed for algorithmic comparison, offering valuable insights that extend

beyond game development to broader applications in spatial search and data retrieval.

ACKNOWLEDGMENT

The author is thankful to God Almighty for the strength required to bring this paper to fruition. Further gratitude is expressed to Ir. Rila Mandala, M.Eng., Ph.D., for his guidance during the IF2211 Algorithm Strategies course and for providing the foundational knowledge that made this application possible. This program can definitely be expanded to where it does not need to search in the same chunk as the player or even search multiple chunks.

REFERENCES

[1] R. M., "Algoritma Brute Force," in *Bahan Kuliah IF2211 Strategi Algoritma*, Sekolah Teknik Elektro dan Informatika, ITB, 2026.

[2] R. M., "Algoritma Brute Force (Bagian 2)," in *Bahan Kuliah IF2211 Strategi Algoritma*, Sekolah Teknik Elektro dan Informatika, ITB, 2026.

[3] R. M., "Algoritma Greedy," in *Bahan Kuliah IF2211 Strategi Algoritma*, Sekolah Teknik Elektro dan Informatika, ITB, 2026.

[4] "Ore Distribution 26.2 - Best Y-Levels for Every Ore," Minecraft Maps. [Online]. Available: https://www.minecraftmaps.com/tools/ore-distribution. [Accessed: June 19, 2026].

[5] R. M., "Algoritma Divide and Conquer," in *Bahan Kuliah IF2211 Strategi Algoritma*, Sekolah Teknik Elektro dan Informatika, ITB, 2026.

[6] R. M. and N. U. Maulidevi, "Breadth/Depth First Search (BFS/DFS)," in *Bahan Kuliah IF2211 Strategi Algoritma*, Sekolah Teknik Elektro dan Informatika, ITB, 2026.

[7] "Fabric Documentation – Project Structure", FabricMC. [Online]. Available: https://docs.fabricmc.net/develop/getting-started/project-structure. [Accessed: June 1, 2026].

Github repository

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

I Gusti Ngurah Alit Dharma Yudha | 13524072