Finding an Optimal Path to A Desired Ending with Maximum CGs in Otome Game Using Backtracking

Bertha Soliany Frandi - 13523026
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: bertha.soliany@gmail.com , 13523026@std.stei.itb.ac.id

***Abstract*—This paper presents a backtracking algorithm to solve the challenge of finding optimal paths in otome games, aiming to maximize collectible Computer Graphics (CGs) and reach desired endings. The algorithm utilizes a depth-first search strategy with effective pruning techniques. The developed program successfully models game narratives, optimizes for multiple objectives, and provides a clear step-by-step guide for users. The program also allows users to do simulation for the obtained optimal path. While demonstrating strong performance and guaranteeing optimal solutions, the approach faces limitations with exponential growth in extremely large graphs and its dependence on fully observable game information, making it less suitable for real-world games. Future work will focus on integrating advanced heuristic search for scalability, incorporating reinforcement learning for handling hidden states, and developing enhanced graphical visualizations for improved user experience.**

Keywords—backtracking; otome game; computer graphics

I. Introduction

The popularity of video games has significantly increased over the years. This occurrence led to the emergence of numerous genres. One of the most popular genres nowadays is dating simulation games or commonly known as *otome* games. Otome games are story-driven interactive experiences typically targeted toward a female audience. This type of game focuses on building romantic relationships with various characters.

While otome games traditionally emphasize narrative and character development, many modern titles incorporate diverse gameplay mechanics such as rhythm-based sequences, turn-based combat, or strategy elements. A defining feature of otome games is their branching storyline where the player's decisions determine the route, romance outcomes, and ultimate endings.

A key collectible in these games is Computer Graphics (CGs) that represent significant story moments. However, not all CGs are accessible in a single playthrough. In many free-to-play otome games, such as Mystics Messenger, players must make specific choices to unlock CGs whereas premium games like Café Enchanté often include access to all CGs once purchased. Furthermore, within a single route, certain choices may not unlock a CG whilst the other choice present will trigger the CG for the story. This mechanism makes a specific type of challenge for players to obtain or experience the CG. The free-to-play game gives the challenge to unlock as many CGs as possible while the premium otome games player will want to know the full story of the CG and not only look at the CG. It's like giving an extra enjoyment to the game if the plater can get the CG in premium games.

Community-driven wikis sometimes provide detailed guides to obtaining CGs and endings. But it is rather tasking to search for it one by one and construct it to a single path that will ensure the plater to get the desired ending and maximal CGs. This desire to experience the story and CGs while getting the desired ending presents an opportunity for algorithmic assistance.

This paper proposes the use of a backtracking algorithm to explore branching narrative paths and identify an optimal sequence of choices that leads to a desired ending while unlocking as many CGs as possible. This approach assumes the availability of structured data about the game's choices and outcomes such as those often compiled in fan-made wikis.

II. Theoretical Foundation

A. Backtracking

Backtracking is an algorithm for solving problems that gradually constructs potential solutions and the stops a path when it finds that it cannot result in a viable or ideal solution. It employs a depth-first search strategy, thoroughly analyzing each potential decision branch before going on to the next. This approach works particularly well for combinatorial search space problems like pathfinding, riddles, and constraint statisfication issues.

Every narrative choice made in an otome game can be visualized as a branching node in a tree. Finding a route from the game's start to a particular conclusion while optimizing collection rewards (CGs) is the goal. Backtracking enables this by assessing every potential course of action and going back

(backtracking) when the path results in an undesirable outcome or does not enhance the CG collection.

Pruning is a key idea in backtracking, which is the practice of ignoring entire subtrees after it is determined that they cannot produce a better result than what has already been discovered. When compared to a blind, brute-force approach, this significantly boosts performance.

The general features of backtracking include a recursive approach, incremental solution construction, abandonment of partial solutions upon constraint violations, and suitability for problems involving many decision points and outcomes.

This is one of the many examples of pseudocode for backtracking algorithms.

```
function backtrack(path, current_state):
    if goal_condition_met(current_state):
        update_best_solution(path)
        return
    for option in get available_choices(current_state):
        if is_valid(option, current_state):
            apply(option, current_state)
            backtrack(path + [option], current_state)
            undo(option, current_state)
```

In this pseudocode, path tracks the sequence of decisions made so far and current_state reflects the current progress (e,g., affection points, CGs collected). The algorithm explores valid choices, recursively extends the current path and backtracks if necessary.

Use cases of backtracking are solving puzzles, finding all permutations or combinations, navigating game tress, and solving decision-based problems like narrative branching in games.

Since otome games have a decision-tress structure and several goals (such as maximizing CGs and reaching a specific conclusion), backtracking is a suitable option. However, its success is dependent on the capacity to specify limitations and optimize path evaluation to avoid unnecessary processing.

B. Otome game and CGs

A type of visual novels known as otome games focuses on romantic narratives told from a primarily female point of view. The genre is mostly intended for female viewers, hence the term otome (乙女) in Japanese means “maiden.” Gamers take on the role of a female lead who engages with several male characters, each of whom stands for a distinct romantic path. Various tale scenarios and endings are possible since these pathways branch based on the decisions made by the player.

Although romance is a major theme in the genre, the main draw of otome games is the opportunity to immerse oneself in a deep, emotional story that is influenced by the choices made by the player. Character connections, the plot’s trajectory, and the opening of special tale events are all impacted by the decisions made throughout the narrative. While gameplay elements like riddles, stat-building, or time-limited events are frequently added to otome games, the emotional journey and branching tales continue to be the primary focus.

Computer Graphics, sometimes referred to as event CGs, are one of the main collectible components in otome games. A CG is a full-screen illustrated image that represents a significant plot event, like a dramatic scene, a romantic confession, or an emotional turning point. Players can unlock these CGs as rewards depending on their decisions and advancement.

By providing visual representation to tale segments that would otherwise be text-based, CGs increase the narrative’s emotional impact and level of immersion. For players who are completionists and work to unlock the entire CG gallery the game offers, they also serve as collector milestones. But not every CG will appear in a single playthrough. Some have to do with particular character paths, speech options, attachment levels, or unspoken prerequisites that aren’t mentioned in the game.

For example, in Mystic Messenger, certain CGs are only available if the player chooses the correct responses during chatroom interactions. In contrast, all CGs are accessible after purchase in premium games such as Café Enchanté, but players still have to figure out the right way to stimulate them inside the narrative.

CGs are a major factor in player involvement since they act as both rewards and story milestones. Many players consider unlocking CGs to be an essential component of the ultimate gaming experience since it provides them with both visual pleasure and a feeling of accomplishment in the story. This makes CG acquisition an appropriate target for optimization with computational approaches like backtracking.

III. Implementation

The implementation of the optimal pathfinding system for otome games is structured into several key components, encompassing the story data, the core backtracking algorithm, the execution environment, and the presentation of results. The code can be fully seen in author’s GitHub under the name “makalah_stima”.

A. Dummy Story Structure

The foundational element of the program is the story. For the sake of implementing backtracking algorithm, the story is present using JSON format that are later parsed into Java objects. The reason for using JSON format is for easy modification and expansion of narrative content without requiring code changes. The “Mystic Messenger” themed story serves as the concrete example, demonstrating the system’s ability to navigate complex character-based routes and collectible elements, such as CGs.

Three of Mystic Messenger’s character are being used for the dummy story. They’re Jumin Han, 707

(Seven), and Zen. There is no specific reason for choosing them but the author only chose three character and not all character exist in Mystic Messenger because three character is enough for demonstrating the program.

The story_data.json file dictates the entire narrative, including its title, description, a collection of nodes, and CG description. Each node within the JSON corresponds to a StoryNode object, which is the fundamental unit of the story. A StoryNode encapsulates ‘id’, ‘title’, ‘description’, ‘cgs’, ‘choices’, ‘isEnding’, and ‘endingType’. ‘id’ is a uniquie identifier for the node. For example, “start” and “jumin_route”. ‘title’ is a descriptive title for the scene (e.g., “Mystic Messenger Chat”). ‘description’ is the narrative text displayed to the player for that scene. ‘cgs’ is a list of CGs that are unlocked upon visiting this node and representing visual events within the game. ‘choices’ is a list of Choice objects. Each of it detailing an ‘id’ for user selection, ‘text’ for the option, and ‘destination’ for the ‘id’ of the next StoryNode if this choice is made. This list defines the branching pathways from the current node. ‘isEnding’ is a Boolean flag indicating if the node marks an end to a story path. ‘endingType’ is a string that categorizing the type of ending reached which contributes the path’s overall score.

The StoryLoader.java utility class is responsible for loading the story from JSON. It parses the story_data.json into a Story object. This Story object then holds a map of all StoryNode instances, making them accessible by their IDs. The Story class also maintains all ‘cgDescriptions’ and the ‘startNodeId’.

The story is characterized by character routes, multiple endings, and CG acquisition. As said before, there are three characters that the dummy story has. Each of them has their own routes. The narrative is distinctly separated into routes for Jumin, Seven, and Zen. Each of it commencing from the “start” node. Each character’s route contains unique StoryNode sequences and choices that lead to different relationship progressions and outcomes.

For the ending, the dummy story contains a total of 10 ending nodes that categorized into various ending type. For Jumin’s route there are good and normal ending. For Seven’s route there are good, bad, normal, and secret ending. For Zen’s route there are good and normal ending. These diverse endings allow the StoryPathFinder to target specific narrative conclusions.

For the CG acquisition, the author has already stated that CGs are one of collectible in otome game. Much more if it’s an otome game like Mystic Messenger. The dummy story gave 24 distinct CGs available across all routes. Each StoryNode that unlocks a CG list its ‘id’ within its ‘cgs’ array. The Story object stores detailed description (e.g., "jumin_wedding": "Beautiful wedding ceremony with Jumin”). This enrich the context of each collected artwork. The program ensures that CGs are only counted once per path even if a node is visited multiple times within a cycle prevention context.

B. Algorithm Design

The core intelligence of the program resides in the StoryPathFinder class that implements a backtracking algorithm to identify optimal narrative paths based on user-defined criteria (primarily maximizing CG collection for a desired ending).

The pseudocode for backtracking is the one that already mention in the theoretical foundation. In the context of the StoryPathFinder.java implementation, ‘path’ corresponds to a StoryPath object which dynamically stores the ‘nodeSequence’ and ‘choiceSequence’ taken so far and the ‘current_state’ implicitly includes the current StoryNode (‘currentNodeId), the ‘collectedCgs within the StoryPath, and the ‘visitedInPath’ set used for cycle detection. ‘goal_condition_met’ is satisfied when ‘currentNode.isEnding()’ is true. ‘update_best_solution’ involves adding the ‘completePath’ to the ‘allPaths’ list for later sorting and selection. “get_available_choices’ is ‘currentNode.getChoices()’. ‘is_valid(option, current_state)’ is checked by ‘!visitedInPath.contains(nextNodeId)’ for ensuring no cycles are formed within a single path. ‘apply(option, current_state)’ is impliciyly handle by Java’s call stack and the creation of new ArrayList and HashSet objects when extending the path for preventing modifications to previous states.

The backtrack method in StoryPathFinder initiates a Depth-First Search (DFS) from the story’s start node (‘story.getStartNodeId()’). For each ‘currentNode’, it iterates through all its ‘choices’. For every choice, it identifies the ‘nextNodeId’.

A crucial aspect is the ‘visitedInPath’ HashSet. Before traversing to ‘nextNodeId’, the algorithm checks if ‘nextNodeId’ has already been visited in path within the ‘currentPath’. If not, a ‘newVisited’ set is created, copying the current ‘visitedInPath’ and adding ‘currentNodeId’. Then, the backtrack method is recursively called with the ‘nextNodeId’ and the updated path state. This effectively prunes paths that would otherwise enter infinite loops and ensuring termination and efficiency.

For the CGs and endings tracking, the program has a part for that too. The StoryPath class serves as a mutable record of a single traversal through the story. It sorts the ‘nodeSequence’, ‘choiceSequence’ and ‘collectedCgs’. When the backtrack method processes a ‘currentNode’, it creates an ‘extendedPath’ by calling ‘cuurentPath.extend(currentNodeId, null, currentNode.getCgs()). This method in StoryPath adds the ‘currentNodeId’ to the ‘nodeSequence’ and ‘addAll’ new CGs from the ‘currentNode’ to the ‘collectedCgs’ set. This effectively tracking all CGs acquired along the path. If a ‘currentNode’ is an ending node, the ‘extendedPath’ is marked complete by calling ‘extendedPath.complete(currenNode.getEndingType())’. This ‘complete’ method sets the ‘isComplete’ flag to true and records the ‘endingType’ for the path. All ‘compltePath’ objects are then added to the ‘allPaths’ list

within the ‘StoryPathFinder’, which holds all valid narrative conclusions discovered during the search.

There are also scoring system in the program. Once all possible paths to a target ending are found, the ‘selectedBestPath’ or ‘selectedBestPathToSpecificEnding’ method ranks them based on a defined scoring system. There are base score and ending bonus. Base score is collected from collected CG which contributes 10 points while ending bonus are awarded based on the ending type.

C. *Tools and Execution*

The entire application is develop using Java and designed to run as a command-line interface (CLI) application. This happens cause the program prioritizing accessibility and ease of use without complex graphical dependencies.

The program is built using Java and require Java 11 or higher for compilation and execution. The project structure is organized into packages for modularity and maintainability as it is made using object-oriented program concept. The user application which is entirely menu-driven within console is being made possible through StoryApplication.java as the main entry point. It present users with a main menu that offer two modes. Mode 1 is for finding optimal path for a desired ending with maximal CG obtain using backtracking and mode 2 is playing a simulation. Of course, there’s option for exiting the program too.

Mode 1 is used with calling ‘findOptimalPathMode’. Users can select a character and a specific ending that they want. The program dynamically lists the available endings by filtering StoryNode Ids. Once selections are made, the StoryPathFinder is invoked to fins the optimal path for that specific ending.

Mode 2 allows users to experience the dummy story made for this paper. It offers three options, interactive optimal mode, auto mode, and free exploration mode. Free exploration mode will happen if there are no optimal path found. This can happen if the user has yet to do mode 1 to find an optimal path for a specific ending. Free exploration mode allows users to make their own choices without guidance to experience the story organically and seeing where their decisions lead. This represents a normal gameplay of an otome game. Interactive optimal mode can be used when users has already used the program to find an optimal path (invoking the mode 1). The program then will save the path find and use it for interactive optimal mode and auto mode. In interactive optimal mode, user can freely make choices with the optimal choice for the current scene is highlighted. This provides guidance while allowing player agency. There are notes for this mode. Users can choose that freely. The program will lead the users to the optimal path because the purpose of this mode is for the users to see it more clearly about the choice being made. The last mode, auto mode, allows program to automatically play through the last found optimal path with optional delays between scenes for a more narrative experience. The ‘previewPath’ method provides a quick overview of the route before full simulation.

Although console based, the StoryPathSimulator and StoryApplication utilze ANSI escape codes for basic test formatting, including colors and bolding to enhance readability and highlight important information. Scene content, CG collection notifications, and choice displays are dynamically rendered to provide an immersive textual experience.

D. *Output and Results*

The program provides detailed outputs for both optimal pathfinding and simulation, offering insight into the game’s structure and the algorithm’s performance. These detailed outputs not only guide the user but also provide valuable analytical data for game designers or researchers studying narrative structures and player engagement.

When an optimal path is found, the program presents it in a clear step-by-step format. Path summary includes the overall score, CGs collected, number of scenes/nodes (length), and ending type. Optimal route includes each step that clearly shows the step number, the node title, and the choose action with the exact text of the optimal choice to make. This serves as a direct guide for users. Collected CGs part is a comprehensive sorted list of all cg IDs and their description collected along that specific optimal route is presented at the end. Result part is a concluding message confirms that this is the best path for users chosen character and ending.

Beyond just the optimal path, the program offers in-depth statistics about the backtracking process and the overall story structure. Search performance part reposts the total path the backtracking algorithm explored and the complete path found. This provides a direct measure of the algorithm’s workload and the complexity of the narrative graph. The path statistics part is for calculating and displaying the minimum and maximum scores achieved across all paths, the average score of all complete paths, the average number of CGs collected per path, and the average number of nodes traversed in a path. Top paths by score part is for listing the top 3 or fewer if less than three paths are found by their score, along with their collected CGs and ending type. Ending type distribution part is to provide a count of how many paths leads to each ending type. This offer insights into the prevalence of different outcomes in the story. Lastly, the CG collection frequency part is for analyzing how frequently each individual CG is collected across all explored paths, sorted by frequency. This can indicate how common or rare certain CGs are to obtain through typical playthroughs.

IV. Analysis

This section will evaluate the performance and applicability of the backtracking algorithm within the context of otome game pathfinding. It will also compare with a different search algorithm to make the analysis more pronounced.

Backtracking proves to be a well0suited and generally efficient algorithm for solving the optimal pathfinding problem in otome game. This is primarily due to the structured nature of branching narratives. The problem can be naturally modeled as finding a path in a directed acyclic graph (DAG) or a graph where cycles are explicitly managed.

With their numerous decision points and multiple outcomes, otome games generate a large combinatorial search space. As a systematic way to explore all possible configurations, backtracking is inherently designed for such problems. It incrementally builds a solution and if partial solution cannot lead to a viable of optimal complete solution, it backs out or prunes that branch.

A key factor contributing to its efficiency is the implementation of pruning strategies. In this system, pruning is achieved primarily through cycle prevention and goal-directed search. The 'visitedInPath' set withing the backtrack method ensures that the algorithm does not revisit a node within the same path. This prevents inifinite loops in cyclic graphs and dramtically redus redundant explorations, ensuring that each explored path is unique and finite. For the goal-directed search, it is done with invoking 'findOptimalPathToSpecificEnding(targetEndingNodeID)'. With that method being invoke, the algorithm prioritizes paths leading to a designed target ending node id. Any path that reaches and ending node that is not the target ending node id is immediately terminated within the backtrack method. This effectively prunes entire subtrees that cannot lead to the desired outcome, significantly narrowing the search space compared to a general pathfinding algorithm that seeks any ending.

For the sample/dummy story, which has 33 nodes and 24 CGs, the backtracking algorithm quickly identifies the optimal path. The detailed algorithm analysis outputs the total path explores and complete paths found. While a specific numerical output is dynamic based on execution, these metrics demonstrate that the algorithm systematically explores the necessary branches without becoming overwhelmed, finding the best path based on the defined scoring system. The clear and logical structures of decisions in typical otome games makes backtracking a feasible and effective choice.

While efficient for well-structured problems, backtracking faces inherent challenges when the complexity of the problem space increases. In a theoretical worst-case scenario, if every node has a high branching factor or many choices and paths are very long, the number of possible paths can grow exponentially. If the story graph were extremely dense with many valid paths between any two nodes, an exhaustive search might become computationally prohibitive. This is a characteristic of many combinatorial search problems. There is also a factor of branching factor (the number of choices per StoryNode) and number of nodes in a path. If those increase, it can lead to a significant increase in the number of 'exploredPaths'.

Fortunately, real-world otome games with hundreds of nodes, rarely exhibit truly "worst-case" graph structures. They often segment narratives into character-specific routes which naturally partitions the larger graph into smaller more manageable sub-graphs. Current implementation leverages this by allowing users to select a character before searching for an ending type. This implicitly reduces the initial search scope to a character's specific route.

The cycle mechanism remains crucial for scalability as it ensures that the algorithm does not waste resources on redundant cyclic paths. This could otherwise lead to non-terminating searches in games with loops. The goal-directed pruning also ensures that only paths relevant to the target ending node are fully explored. These pruning techniques help to keep the practical performance within reasonable limits for typical game sizes but they cannot entirely overcome the exponential nature if the game's decision tree truly explodes in complexity.

The current backtracking model also assumes complete and explicit knowledge of the story's structure, choices, and outcomes, as provided in the JSON file. This is often an oversimplification for many otome games. Real-world otome games frequently incorporate hidden conditions or internal states that are not immediately visible to the player or easily extractable from a simple node-and-choice structure. These can include affection points, event flags, dynamic choice availability, and randomness which would make deterministic pathfinding impossible.

Without knowledge of these hidden conditions, the backtracking algorithm would be operating on an incomplete representation of the game's decision graph. It would potentially miss optimal paths that require specific hidden stat thresholds or event triggers to unlock certain crucial choices or CGs. To account for hidden conditions, the 'current_state' would need to be significantly enriched. Instead of just 'nodeSequence' and 'collectedCgs', it would need to track all relevant affection points, time stamps, and Boolean flags. This would dramatically increase the size and complexity of the state space, making direct traversal more computationally intensive and memory-demanding. The most significant challenge would be acquiring this hidden information. Game files often obfuscate these internal mechanisms, requiring extensive reverse-engineering or relying on community-driven wikis that are meticulously compiled through trial-and-error gameplay. The current program explicitly states its assumption of availability of structured data about the games and choices outcomes. To handle hidden conditions, a more advanced approach might be necessary. This could involve state-space search, heuristic search, and reinforcement learning.

Comparing backtracking algorithm with A* algorithm is a worthy exploration. A* is a best-first search (BFS) algorithm renowned for its efficiency in finding the shortest or lowest-cost path in a graph. It achieves this by using a heuristic function to estimate the cost from the current node to the goal node, combining it with the actual cost from the start node to the current node. Applying A* to this problem would require defining an admissible and consistent heuristic function. For example, to maximize CGs, a heuristic could estimate the maximum possible additional CGs that can be collected from the current node to the target ending. For scoring, this would involve estimating the maximum potential score achievable from the current node's sub-graph. If a strong, accurate, and admissible heuristic can be formulated, A* could potentially

find the optimal path much faster than exhaustive backtracking, as it intelligently prioritizes the most promising branches, avoiding deep exploration of clearly sub-optimal paths.

While A* offers theoretical performance advantages for shortest path problems, its direct application to this specific multi-objective optimal path problem (maximizing CGs and score) would require careful heuristic engineering. For a problem where finding the absolute optimal path is paramount and the graph size is manageable, backtracking offers a robust and reliable solution, often being simpler to implement than designing and validating a complex A* heuristic. The current backtracking approach is effective for the defined problem due to the clear objective function (scoring) and the ability to efficiently prune irrelevant branches.

V. Conclusion

The implemented backtracking algorithm effectively demonstrates its capability of identify optimal paths in branching narrative games like otome games, successfully maximizing collectible CGs and achieving desired endings within a character-based story structure. Its strengths lie in guaranteeing an optimal solution given explicit game data, its adaptability to various scoring objectives and the practical efficiency gained from crucial pruning techniques like cycle detection and goal-directed search. However, this approach faces limitations, notably its potential for exponential growth in extremely large or densely interconnected graphs, and a significant reliance on completely explicit game data, rendering it less effective for real-world games with hidden conditions, dynamic states, or unseen parameters. Future work should therefore focus in augmenting the program's capabilities through the integration of advanced heuristic search algorithm like A* for improved scalability, incorporating reinforcement learning to navigate environments with hidden states and partial observability and developing enhanced graphical visualizations for a more intuitive user experience and deeper narrative analysis.

Video Link at Youtube

https://youtu.be/NWJjuZJYW4M

Code Link at GitHub

https://github.com/BerthaSoliany/makalah_stima.git

Acknowledgment

The author would like to express gratitude to God for providing strength and clarity, to the game development community for their shared knowledge. The author is also grateful to otome game developers, especially, Cheritz, the developers Mystic Messenger, for creating a lovely game that inspired this study. Lastly, the author extends gratitude to the family for the support and encouragement throughout this study.

References

[1] R. Munir, "Algoritma Runut-balik (Backtracking) (Bagian 1)," [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/15-Algoritma-backtracking-(2025)-Bagian1.pdf. [Accessed: June. 23, 2025].

[2] "Backtracking Algorithm," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/dsa/backtracking-algorithms/. [Accessed: June, 23, 2025].

[3] Z. Zhang, "The Rise of Otome Game in China: Exploring the Social and Psychological Traits of Otome Game Players," [Online]. Available: https://www.ewadirect.com/proceedings/lnep/article/view/14667. [Accessed: June. 23, 2025].

[4] G. R. Diniz, R. D. D. Regis, J. C. V. Ferreira, "Otome Games: globalization and glocalization processes, conceptualization and data analysis of Brazilian players," [Online]. Available: https://www.researchgate.net/publication/374287484_Otome_Games_globalization_and_glocalization_processes_conceptualization_and_data_analysis_of_Brazilian_players. [Accessed: June 24, 2025].

Statement

I hereby declare that the paper I wrote is my own writing, not an adaptation or translation of someone else's paper, and is not plagiarized.

Bandung, 24 Juni 2025

Bertha Soliany Frandi
13523026