Jurnal Teknik Informatika (JUTIF)
Vol. 5, No. 6, December 2024, pp. 1883-1892

DOI: https://doi.org/10.52436/1.jutif.2024.5.6.4031
p-ISSN: 2723-3863
e-ISSN: 2723-3871

OPTIMIZATION OF BACKTRACKING ALGORITHM WITH HEURISTIC STRATEGY FOR LOGIC-BASED SORTING PUZZLE GAME SOLVING

Eka Qadri Nuranti B[1], Naili Suri Intizhami[*2], Primadina Hasanah[3]

[1,2]Computer Science, Institut Teknologi Bacharuddin Jusuf Habibie, Indonesia
[3]Mathematical Science, University of Liverpool, United Kingdom
Email: [1]eka.qadri@ith.ac.id, [2]naili.suri@ith.ac.id, [3]primadina.hasanah@liverpool.ac.uk

(Article received: October 17, 2024; Revision: December 23, 2024; published: January 04, 2024)

Abstract

Puzzle Game Sorting is a logic-based puzzle game where players must transfer colored balls into tubes until each tube contains only one color. Although it appears simple, the game becomes increasingly challenging at higher levels, testing players' logical thinking and patience. This study proposes using the backtracking algorithm combined with optimization strategies, such as conflict heuristics and lookahead, to address players' challenges at advanced levels. The test results indicate that the optimized backtracking algorithm can solve the game faster and with more efficient steps compared to manual methods. Specifically, heuristic optimization strategies significantly improved performance, reducing execution time by up to 91.4% and the number of steps by up to 76.9% at the most complex levels. These findings demonstrate that combining the backtracking algorithm and optimization strategies is an effective solution for solving puzzles in Sorting, particularly at levels with increasing complexity.

Keywords: *backtracking algorithm, game, heuristic, lookahead, puzzle, sorting.*

1. INTRODUCTION

Puzzle-based games maintain fans because they can train logical and strategic thinking skills in fun ways [1], [2]. One example of this genre is Puzzle Game Sorting [3], a simple but challenging concept. In this game, players sort several different types of color balls into several tubes until each tube is filled with only one color. The key characteristic of these puzzles is that each tube operates like a stack, requiring players to adhere to the "last-in, first-out" (LIFO) principle [3]. This feature introduces an additional challenge, as players must strategically plan their moves to ensure they do not obstruct access to items that will be needed in subsequent steps. Although it appears easy, this game requires high precision and patience, especially at higher levels, where the number of types of letters and tubes becomes more complex.

Increasing levels of difficulty often make players feel frustrated, especially when they find out that the strategy they are using has reached a dead end. Each level in this game also has different solving steps. This makes the game both interesting and challenging. Therefore, players must concentrate and be patient to determine the proper steps in completing the game for each level.

One of the methods that can be used to solve this problem is the backtracking algorithm [1], [4], [5], [6]. Research by D. Chen et al. [4] applied optimization to the backtracking algorithm through the addition of knowledge learning. M. Esteve et al. [5] combined heuristics and backtracking in Efficiency Analysis Trees to improve accuracy and efficiency. Furthermore, P. Garg et al. [6] evaluated three backtracking approaches—Breadth-first search, Depth-first search, and a parallel hybrid of both—to solve Sudoku puzzles, revealing significant performance improvements of 28% to 56%. Vidal [7] study enhances heuristic search by leveraging relaxed plans and a lookahead strategy, improving performance and scalability in solving larger and more complex planning problems. These studies demonstrate the versatility of backtracking algorithms and their potential for optimization in solving complex problems. The backtracking algorithm can also be combined with other algorithms or methods, such as power outage management [8], signal power [9], and robotics [10].

The backtracking algorithm is widely used to solve problems involving the search for solutions from many possible options because of its speed in finding the correct solution [1], [5]. It can become less efficient when dealing with puzzles of high complexity, owing to the many possibilities to explore. This reduced the effectiveness of the algorithm's results.

The many possibilities explored can also impact the results, which could be better. The results could be better in this game when the time used to complete the game and the number of steps used are excessive. Therefore, optimization of the backtracking algorithm is required so that the solution search process becomes faster and more efficient, one of which is by implementing the lookahead strategy [7], [11], [12], [13] and conflict heuristics [5], [14], [15].

This study applies an optimized backtracking algorithm to Puzzle Game Sorting, a logic-based puzzle where heuristic strategies have not yet been explored, unlike their established use in Sudoku [1], [6], [15]. The integration of lookahead strategies, rarely combined with heuristics for solution search, adds further novelty. By employing optimization techniques such as lookahead and conflict heuristics, the study reduces steps and execution time, enhancing efficiency at higher levels. These findings highlight the potential of combining backtracking with advanced strategies to address increasingly complex puzzles, improving algorithm performance and the gaming experience.

2. METHOD

This research consists of several steps, which will be explained in the following flowchart (Figure 1). The process begins with the Start node, where the research framework and objectives are initialized. The first step, Configuring Game Levels, involves designing the Puzzle Game Sorting configurations with varying levels of complexity to ensure a gradual increase in difficulty. These configurations serve as the basis for subsequent simulations.

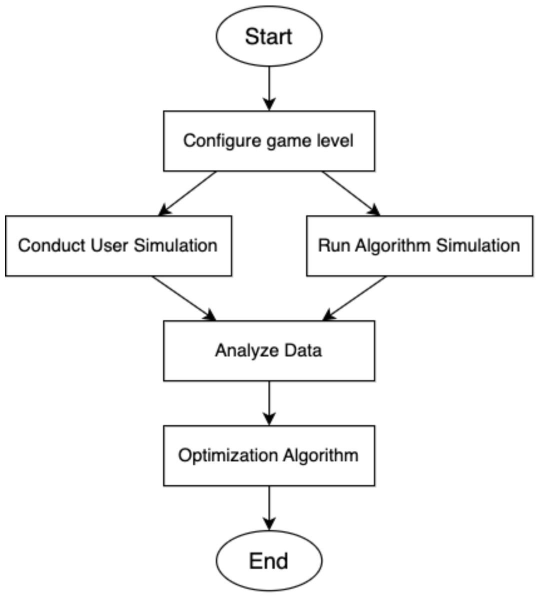


Figure 1. Research Method

From this point, the process splits into two parallel activities: Conducting User Simulation and Running Algorithm Simulation. In the user simulation phase, participants interact with the puzzle, and their actions are monitored using a tracking tool that captures metrics such as completion time, the number of moves, and intermediate states. Concurrently, the algorithm simulation tests the backtracking approach on the same puzzle configurations, recording metrics like execution time, the total nodes explored, and the number of backtracking operations.

The results from both simulations are then consolidated in the Analyze Data phase. This stage focuses on evaluating and comparing the efficiency of human problem-solving strategies and algorithm performance. The insights derived from this analysis are used to enhance the backtracking algorithm in the Optimize Algorithm phase. Finally, the process concludes at the End node, marking the completion of the research workflow. This step-by-step methodology ensures a structured and systematic approach to optimizing the algorithm.

2.1. Puzzle Game Sorting Simulations

This study includes a Puzzle Game Sorting game simulation as part of the research on problem-solving algorithms using the backtracking approach . The simulation aims to evaluate the algorithm's effectiveness in solving the color-sorting problem in tubes. Four levels of the game, each with increasing difficulty, were designed to test the algorithm's performance. These levels included progressively complex configurations, from two tubes with three colors to five tubes with six colors, ensuring a gradual increase in problem complexity.

The study collected data by engaging 6 (six) participants who attempted to solve the puzzle configurations. A custom tool tracked their actions, capturing the time taken, the number of moves made, and the intermediate states of the tubes after each move. Simultaneously, the backtracking algorithm was applied to the same configurations, with its performance evaluated through metrics such as execution time, the total number of nodes explored in the search space, and the number of backtracking operations performed.

Additionally, an observational phase analyzed participants' strategies and common mistakes, yielding valuable insights into human problem-solving behaviors. These findings were directly compared to the algorithm's performance, highlighting areas where optimization could improve efficiency. By integrating data from human participants and the algorithm, the study provides a comprehensive understanding of the problem-solving process. It supports the development of enhanced strategies for optimizing the backtracking algorithm in solving increasingly complex puzzle configurations.

2.1.1. Level 1

In the first level, the game configuration consists of three tubes:

- Tube 1: ['A', 'B', 'A', 'B']
- Tube 2: ['B', 'A', 'B', 'A']
- Tube 3: [] (empty)

In this level, there are only two types of letters, namely 'A' and 'B,' with two full tubes and one empty tube. This is considered an introductory level with a low difficulty level, allowing players to learn the basics of the game and the strategy of moving letters between tubes.

2.1.2. Level 2

At the second level, the number of tubes was increased to five:

- Tube 1: ['A', 'B', 'C', 'A']
- Tube 2: ['B', 'B', 'C', 'A']
- Tube 3: ['C', 'A', 'B', 'C']
- Tube 4: [] (empty)
- Tube 5: [] (empty)

In this level, there are three types of balls ('A,' 'B,' and 'C') spread across three tubes, with two empty tubes as solution spaces. The difficulty of this level increases because of the addition of additional letters and more letter combinations that need to be rearranged correctly.

2.1.3. Level 3

In the third level, the game becomes more challenging with the addition of new types of letters and more complex tube configurations:

- Tube 1: ['C', 'B', 'C', 'D']
- Tube 2: ['A', 'C', 'B', 'A']
- Tube 3: ['A', 'B', 'D', 'A']
- Tube 4: ['C', 'B', 'D', 'D']
- Tube 5: [] (empty)
- Tube 6: [] (empty)

In this level, four types of letters are introduced in this level ('A', 'B', 'C', and 'D'), and there are a total of six tubes, two of which are empty. The difficulty increases significantly as more balls are moved and rearranged in the correct order with a more optimal number of moves.

2.1.4. Level 4

At the fourth level, the game reaches its highest level of difficulty, with five types of letters and a larger number of tubes:

- Tube 1: ['B', 'D', 'E', 'B']
- Tube 2: ['A', 'D', 'D', 'A']
- Tube 3: ['E', 'C', 'B', 'C']
- Tube 4: ['B', 'C', 'A', 'E']
- Tube 5: ['C', 'E', 'D', 'A']
- Tube 6: [] (empty)
- Tube 7: [] (empty)

Five types of letters ('A', 'B', 'C', 'D', and 'E') are spread across five tubes, while two empty tubes are used as spaces for solving. The main challenge at this level is the increasing number of combinations and moving strategies, which require more in-depth calculations to find a solution.

2.2. User Simulation

This research focuses on user simulation as a key method for understanding problem-solving strategies in the Puzzle Game Sorting game. The study involves direct observation of six users as they attempt to complete several game levels, ranging from easy to complex. Participants' interactions with the game are carefully monitored to evaluate how they face challenges, make decisions, and overcome obstacles.

During the simulation, the completion time for each level is recorded to assess the participants' efficiency and ability to solve the puzzle manually. These observations provide valuable insights into human problem-solving behaviors, which are later compared with the performance of the backtracking algorithm. Data collected from the users, including their strategies and time taken, serve as comparison parameters for evaluating the efficiency and effectiveness of the algorithm. The findings from this user simulation are critical for benchmarking the algorithm's performance and identifying areas for optimization, as summarized in Table 2.

Table 2. Completion Time

User ID	*Level Completion Time (seconds)*				*Average Completion Time*
	1	**2**	**3**	**4**	
user1	108.46	183.57	174.41	216.00	170.61 seconds
user2	59.37	117.74	133.28	138.82	112.30 seconds
user3	185.42	98.39	137.77	174.26	148.96 seconds
user4	198.87	104.60	139.14	236.65	169.32 seconds
user5	217.52	116.51	162.09	222.76	179.72 seconds
user6	109.68	127.37	117.96	161.16	129.04 seconds

Table 3. Completion Steps by User

User ID	*Number of Level Steps*			
	1	**2**	**3**	**4**
user1	7	10	14	17
user2	7	10	15	17
user3	7	10	14	18
user4	11	10	14	20
user5	8	10	14	19
user6	7	11	14	17

Table 2 and Table 3 show variations in completion time and number of moves among the six users at four game levels. User2 showed the best performance with the fastest average time (112.30 seconds) and a consistent number of moves, indicating an efficient and systematic strategy. In contrast, user1, who had the slowest average time (170.61 seconds), also demonstrated a higher variability in performance, suggesting potential inefficiencies in decision-making or a trial-and-error approach.

At the beginning level, most users required only seven to eleven moves, with minimal variation, reflecting the relatively low complexity of the task. However, by level 4, the game's complexity increased significantly, as seen in the performance of user4, who needed up to 20 moves and took the longest time to complete the level. This indicates that higher levels present more challenging configurations that demand greater precision and problem-solving effort.

Furthermore, the analysis highlights that fewer moves often correspond to faster completion times, but this is not always true. For example, user6, who required only seven moves in level 1, still had a slower average time than user2. This suggests that efficiency in solving the puzzle involves not only minimizing the number of moves but also optimizing the time spent on each decision.

The results of this observation can help researchers identify patterns of difficulty and levels that are obstacles for players. Data from this manual experiment are then compared with the results of the application of the backtracking algorithm to evaluate the efficiency of time and steps. This comparison provides a clearer understanding of the effectiveness of the backtracking algorithm in completing the game, especially at highly difficult levels.

2.3. Backtracking Algorithm

The backtracking algorithm is an effective solution search technique for combinatorial and optimization problems [4], [6]. This algorithm works by exploring all possible solutions through a Depth-First Search (DFS) approach [6]. One of the main characteristics of backtracking is its ability to cancel decisions (backtrack) when the path taken is proven to not lead to a solution and continue the search with other alternatives [2]. Figure 2 shows the backtracking algorithm.

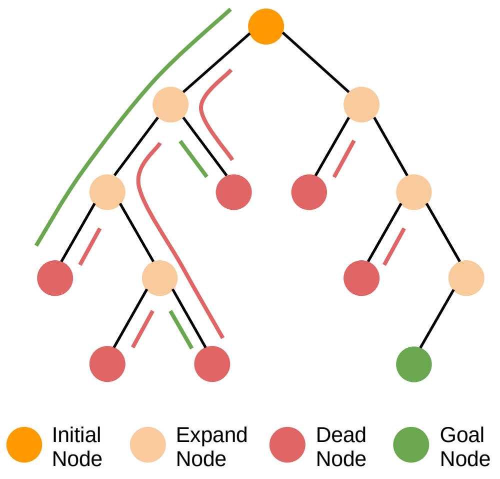


Figure 2. Backtracking Algorithm Concept

The process starts at the initial node, and each action produces a new child node. If the chosen path reaches a dead end, the algorithm backtracks to the previous node and then attempts another step.

During exploration, an expanded node is expanded to produce child nodes. If the move is valid and the new state has not been explored before, then the state becomes a live node. However, if the state is invalid or has been explored previously, the node becomes dead and is pruned to avoid repeated exploration. The algorithm stores visited states using bounding functions in structures such as visited.

This process continues until the goal node is reached (the problem is solved) or all possible paths are explored. If a solution is found, the steps are displayed. Otherwise, the algorithm reports that no solution was possible.

2.4. Backtracking Algorithm In Games

The challenge of the Puzzle Game Sorting game lies in moving letters between tubes using precise and efficient steps. To solve this game automatically, a backtracking algorithm that explores every possible letter movement using a Depth-First Search (DFS) [6] strategy is used. If a step does not lead to a solution, the algorithm returns (backtracks) and attempts an alternative step.

By applying a bounding function and pruning, the algorithm can avoid repeated exploration and reduce computation time. The implementation of this algorithm not only ensures that all paths are tried but also maximizes efficiency by marking and skipping previously visited states.

Figure 3 shows how the backtracking algorithm was implemented in the Puzzle Game Sorting game. Starting from the initial node generation, node expansion, and bounding function application to the stopping condition, each component of the algorithm will be discussed to show how backtracking works effectively to find the optimal solution. The following is an explanation of the core of the flowchart:

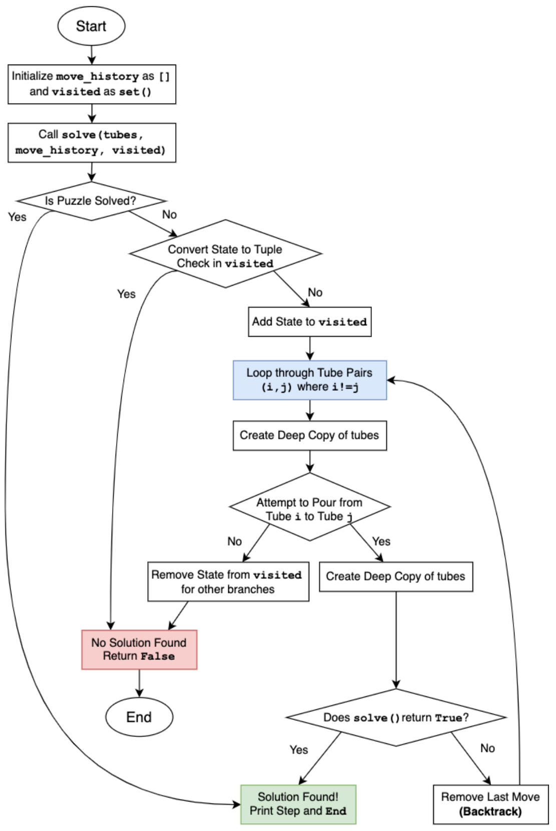

Start
Initialize move_history as [] and visited as set()
Call solve(tubes, move_history, visited)
Is Puzzle Solved?
Yes
No
Convert State to Tuple Check in visited
Yes
No
Add State to visited
Loop through Tube Pairs (i,j) where i!=j
Create Deep Copy of tubes
Attempt to Pour from Tube i to Tube j
No
Yes
Remove State from visited for other branches
Create Deep Copy of tubes
No Solution Found Return False
End
Does solve() return True?
Yes
No
Solution Found! Print Step and End
Remove Last Move (Backtrack)

Figure 3. Backtracking Algorithm Flowchart for Puzzle Sorting Game

1. Start: The algorithm begins by calling the main function. In this step, the execution time recording begins to calculate the total time required by the algorithm.
2. Initialize Variables: move_history is initialized as an empty list to store the steps for moving the balls between the tubes. In addition, visited is initialized as an empty set to store the state of the tubes that have been visited, thereby avoiding cycles and repeated exploration.
3. The solve() function is recursively called to begin searching for a solution.
4. The algorithm checks whether all tubes are filled (i.e. each tube contains balls of one color or is empty).
   a. If Yes: Solution is found. The program prints the solution steps and then ends the process.
   b. If No: The algorithm continues by exploring the remaining steps.
5. Convert State to Tuple and Check in visited (Bounding Function): The algorithm converts the current state of the tube into a tuple so that it can be stored in the visited set:
   a. If the status has been visited, the algorithm backtracks to avoid cycling and repeated exploration.
   b. If it has not been visited, its status is added to visited.
6. Expand Node: The algorithm iterates for each pair of tubes (i, j) and attempts all valid moves between tubes.
7. Generating Initial Nodes: Every time a move is tested, the algorithm creates a deep copy of the current tube state to maintain the original state. Therefore, this could not be changed
8. Dead Node and Bounding Function: The algorithm tries to move a letter (Attempt to Pour) from tube i to tube j.
   a. If successful: This move is added to the move_history
   b. If unsuccessful: The algorithm immediately backtracks and attempts another pair of tubes.
9. If the transfer is successful, the algorithm calls the solve() function recursively with the new tube state.
10. Pruning and Backtracking: After a recursive call is completed, the algorithm checks the result (Check Recursive Result).
    a. If a solution is found: The steps are printed and the process ends.
    b. If not: The algorithm deletes the last step from the move_history and attempts the next step (backtrack).
11. Remove State from visited: Once all branches of this state are explored, it is removed from visited so as not to block other branches in the search.
12. No Solution Found: If a solution is not found after all possibilities have been tried, the algorithm ends the process with a message indicating that no solution was found.
13. End: The algorithm ends by displaying the total execution time in milliseconds.

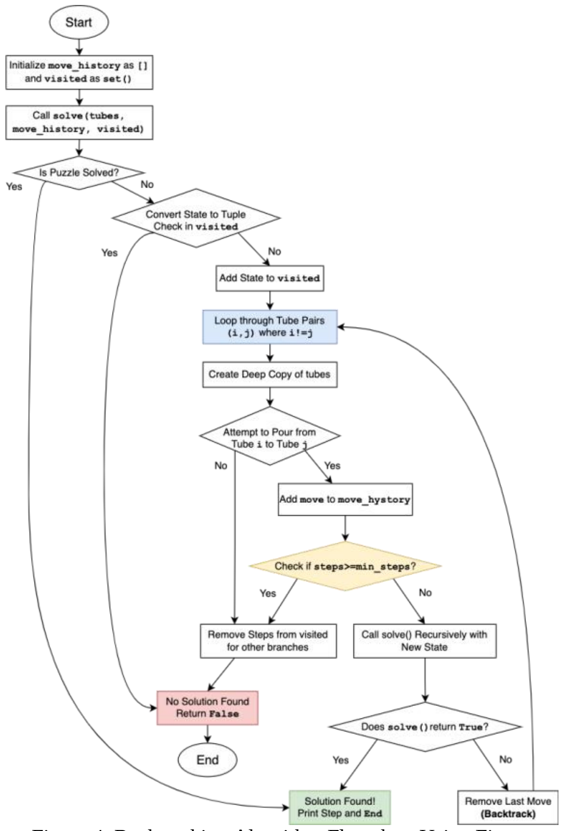
The flowchart illustrates how the backtracking algorithm works in solving a Puzzle Game Sorting problem using recursion, backtracking, and bounding to ensure efficient exploration and avoid unnecessary steps.

2.5. Algorithm Optimization

The implementation of the backtracking algorithm requires optimization strategies to ensure that the solution search process is carried out more efficiently. This optimization aims to minimize the number of steps required and reduce unproductive searches [15]. In this study, the following two optimization strategies were used.

2.5.1. First Optimization

Steps are prioritized to improve the efficiency of the backtracking algorithm. Refer to Figure 4; the yellow rhombus shape demonstrates optimization implementation using heuristics. The ordering of steps is focused on moving letters to tubes with matching colors or empty tubes, minimizing conflicts and speeding up the solution process.



Start
Initialize move_history as [] and visited as set()
Call solve(tubes, move_history, visited)
Is Puzzle Solved?
Yes
No
Convert State to Tuple Check in visited
Yes
No
Add State to visited
Loop through Tube Pairs (i,j) where i!=j
Create Deep Copy of tubes
Attempt to Pour from Tube i to Tube j
No
Yes
Add move to move_hystory
Check if steps>=min_steps?
Yes
No
Remove Steps from visited for other branches
Call solve() Recursively with New State
No Solution Found Return False
End
Does solve() return True?
Yes
No
Solution Found! Print Step and End
Remove Last Move (Backtrack)

Figure 4. Backtracking Algorithm Flowchart Using First Optimization used Heuristic Strategy

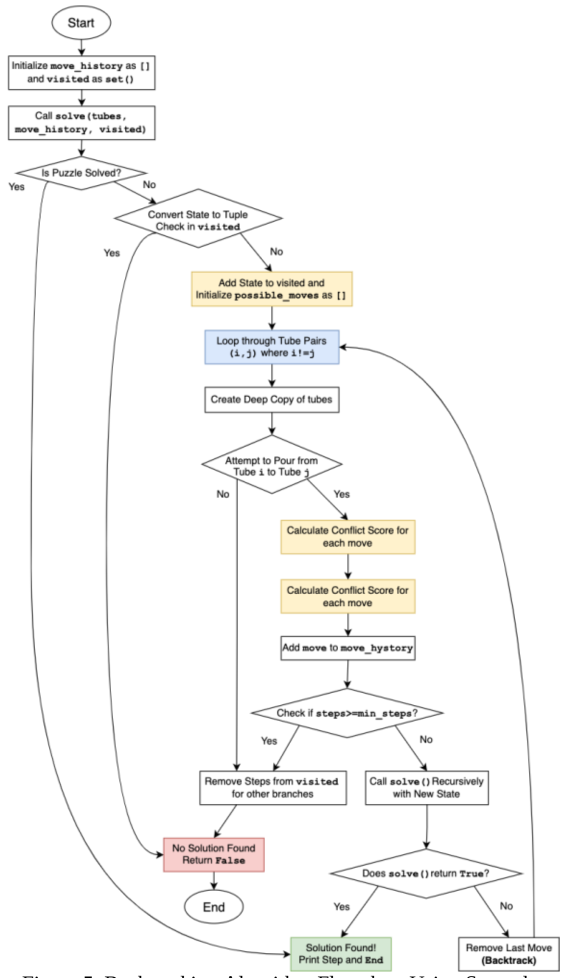
Meanwhile, pruning stops exploration if the current number of steps exceeds the minimum required steps (min_steps); by cutting off unproductive exploration, the algorithm can reduce search time and find the optimal solution faster. This strategy ensures that the algorithm focuses on more relevant and efficient steps.

The primary steps for the first optimization are as follows:

1. Balls are moved to tubes that already contain the same color, increasing the likelihood of fully filling the tube and reducing the total number of steps.
2. If the number of steps exceeds the optimal limit (min_steps), the search is terminated to prevent unproductive exploration.

2.5.2. Second Optimization

The second optimization aims to enhance the efficiency of the backtracking algorithm by applying lookahead strategy [7], [11], [12], [13] and conflict heuristics [5], [14], [15]. In this context, lookahead enables the algorithm to not only focus on the current step but also consider its impact on the next two steps. This helps the algorithm select more advantageous moves that simplify the subsequent stages of the solution process.


Start
Initialize move_history as [] and visited as set()
Call solve(tubes, move_history, visited)
Is Puzzle Solved?
Yes
No
Convert State to Tuple Check in visited
Yes
No
Add State to visited and Initialize possible_moves as []
Loop through Tube Pairs (i,j) where i!=j
Create Deep Copy of tubes
Attempt to Pour from Tube i to Tube j
No
Yes
Calculate Conflict Score for each move
Calculate Conflict Score for each move
Add move to move_hystory
Check if steps>=min_steps?
Yes
No
Remove Steps from visited for other branches
Call solve() Recursively with New State
No Solution Found Return False
End
Does solve() return True?
Yes
No
Solution Found! Print Step and End
Remove Last Move (Backtrack)
Figure 5. Backtracking Algorithm Flowchart Using Second Optimization used Heuristic+Lookahead Strategy

Meanwhile, conflict heuristics play a role in reducing moves that cause letter conflicts, such as placing a letter into a tube where a different letter is at the bottom. By minimizing conflicts, the algorithm can reach the optimal solution faster without unnecessary additional steps. The combination of these two strategies allows the algorithm to work more efficiently and effectively in solving puzzles with higher levels of difficulty.

Figure 5 is an optimization by adding the yellow steps based on the backtracking algorithm and heuristic optimization in Figure 4. The following steps are applied:

1. When making a move, a lookahead strategy is used to assess two steps ahead, determining whether the current move will facilitate subsequent moves.
2. Conflict Heuristic: Minimize moves that increase color conflicts, such as placing a ball into a tube where a different color is at the bottom layer.
3. Utilizing empty tubes more efficiently to temporarily store letters.
4. Moves are ordered based on conflict scores to accelerate the process of reaching a solution.

The optimization of the backtracking algorithm through the implementation of pruning, step prioritization, lookahead, and conflict heuristics enhances the algorithm's efficiency and effectiveness. These strategies help to minimize unproductive exploration, enabling the algorithm to find the optimal solution more quickly, particularly in letter-sorting puzzles with higher levels of complexity.

Implementing these optimizations not only improves the algorithm's performance but also provides a more practical solution for application in strategy games, such as letter sorting puzzles into tubes.

Table 4 Comparison of Backtracking Algorithm Optimization

Aspects	*First Optimization*	*Second Optimization*
Optimization Technique	Depth limit and maximum steps (min_steps)	Prioritizing moves based on conflict scores
Optimization Objective	Reducing unnecessary steps with depth limits	Minimizing color conflicts during moves
Priority Method	No priority order for moves	Moves with the lowest conflict scores are prioritized
Usage Scenario	Suitable for large cases with many steps	Effective when there are many tubes with mixed colors
Heuristic Complexity	Simple	More complex with conflict score calculations

3. RESULT

Several key insights emerged from solving the puzzle sort game using the backtracking algorithm before and after implementing the first and second optimization strategies. The performance of the backtracking algorithm before and after optimization offers valuable insights into the impact of increasing game complexity on computational efficiency. Notably, the algorithm's performance degradation in more complex levels highlights its limitations in handling large search spaces, which becomes increasingly apparent as the game levels rise. This finding underscores the importance of addressing the growing computational demands that come with higher levels of complexity to maintain efficiency.

Table 5. Backtracking Algorithm Performance in Games

Level	*Time (ms)*	*Steps*
1	3.41	7
2	8.94	18
3	5.52	32
4	41.21	78

As detailed in Table 5, the backtracking algorithm's performance demonstrates a clear trend: both execution time and the number of steps required to solve the puzzle rise consistently with an increase in game level and complexity. At Level 1, the execution time was notably low at 3.41 milliseconds, requiring just seven steps to complete the game. However, by Level 4, the execution time escalated dramatically to 41.21 milliseconds, with the number of steps soaring to 78. This substantial increase in time and steps suggests that higher levels of the game present a much broader and more intricate search space, requiring the algorithm to explore a significantly more significant number of potential game states. As complexity rises, the increase in computational load reflects a core characteristic of backtracking algorithms: as the problem space expands, the number of paths the algorithm must investigate grows exponentially, resulting in longer execution times and more steps.

Table 6 Comparison of User Steps and Backtracking Algorithm for Level 2

No	*User Steps*	*Backtracking Algorithm*
1	From Tube 1 to Tube 4	From Tube 1 to Tube 5
2	From Tube 2 to Tube 4	From Tube 1 to Tube 4
3	From Tube 1 to Tube 5	From Tube 2 to Tube 4
4	From Tube 2 to Tube 5	From Tube 2 to Tube 5
5	From Tube 3 to Tube 5	From Tube 1 to Tube 2
6	From Tube 3 to Tube 2	From Tube 1 to Tube 4
7	From Tube 1 to Tube 2	From Tube 2 to Tube 1
8	From Tube 3 to Tube 4	From Tube 3 to Tube 2
9	From Tube 1 to Tube 4	From Tube 2 to Tube 5
10	From Tube 3 to Tube 5	From Tube 1 to Tube 2
11		From Tube 3 to Tube 1
12		From Tube 1 to Tube 2
13		From Tube 2 to Tube 1
14		From Tube 3 to Tube 2
15		From Tube 1 to Tube 5
16		From Tube 2 to Tube 4
17		From Tube 1 to Tube 2
18		From Tube 3 to Tube 1

It is necessary to pay attention to Table 6 to carry out further optimization on the backtracking algorithm. The backtracking algorithm's execution highlights significant inefficiencies, particularly in its tendency to revisit states and perform redundant actions. For example:

- Step 5–7: The algorithm performs a sequence of moves between Tube 1 and Tube 2, followed by Tube 1 to Tube 4 and then back to Tube 2. These steps ultimately do not contribute directly to solving the puzzle and are reversed later.
- Step 10–13: Another set of redundant steps is observed, where balls are repeatedly moved between Tube 2 and Tube 5, Tube 1 and Tube 2, and Tube 3 and Tube 1. This brute-force exploration demonstrates the algorithm's reliance on trying all possible configurations, often leading to actions that are subsequently undone during backtracking.

In contrast, the user's approach demonstrates greater efficiency and a more deliberate strategy:

- Steps such as moving directly from Tabung 1 to Tabung 4, Tabung 2 to Tabung 4, and Tabung 1 to Tabung 5 illustrate a clear understanding of how to utilize empty tubes to group balls of the same color effectively.
- The user avoids unnecessary back-and-forth movements by planning moves that contribute directly to solving the puzzle, thereby reducing time and effort.

While the algorithm works efficiently at lower levels, its performance drops dramatically as the game complexity increases. This performance drop at higher levels highlights essential limitations of the backtracking approach, underscoring the need for optimization strategies to improve its efficiency. The algorithm can significantly reduce unnecessary moves by adopting heuristics (Figure 4) that mimic user strategies, such as prioritizing moves that immediately group elements or efficiently using empty tubes. Additionally, additional optimizations can be performed using lookahead strategy (Figure 5). With such optimizations, the algorithm can better predict unnecessary moves to solve more advanced levels of the game, where the sheer number of possible game states may exceed its capacity for timely and efficient problem-solving.

Table 7. Backtracking Algorithm Performance Using Optimization

	First Optimization		*Second Optimization*	
Level	*Time (ms)*	*Steps*	*Time (ms)*	*Steps*
1	4.58	8	4.58	7
2	2.03	12	9.32	14
3	4.02	36	7.95	17
4	8.13	19	3.56	18

Table 7 compares the performance of the backtracking algorithm with two distinct optimization strategies across different game levels. Both optimizations are designed to improve the algorithm's efficiency by reducing execution time and the number of steps. Interestingly, the optimizations yield different success levels depending on the game's complexity. For example, at Level 2, Optimization 1 is superior in execution time, achieving a remarkable reduction of 2.03 milliseconds compared to the 9.32 milliseconds required by Optimization 2. However, as the complexity increases, the relative performance of the two optimizations shifts. By Level 4, Optimization 2 significantly outperforms Optimization 1, reducing execution time to just 3.56 milliseconds, while Optimization 1 requires a much higher 8.13 milliseconds. This variation in performance highlights the context-dependent nature of each optimization strategy, suggesting that different methods may be better suited to varying levels of complexity.

In addition to execution time, the number of steps required by each optimization shows further differences in performance. Optimization 2 is more effective in minimizing the number of steps, especially at Level 3, where it only requires 17 steps compared to the 36 steps needed by Optimization 1. This substantial step reduction indicates that Optimization 2 is more successful at curtailing unnecessary exploration, enhancing its overall efficiency. Moreover, in the most complex scenario, Level 4, Optimization 2 again demonstrates its superiority, achieving not only a faster execution time but also a reduced number of steps. These results suggest that the advanced techniques employed by Optimization 2, such as lookahead mechanisms and conflict heuristics, play a pivotal role in enhancing the algorithm's ability to navigate complex search spaces more efficiently.

The performance comparison between the six users and the optimized backtracking algorithm reveals several key trends. At Level 1, most users needed between 7 and 11 steps, closely aligning with the algorithm, whereas Second Optimization required seven steps and First Optimization took eight steps. At Level 2, the users completed the puzzle in 10 to 11 steps. At the same time, the algorithm needed more steps—12 forFirst Optimization and 14 for Second Optimization—indicating that the users were more efficient at this level. In Level 3, while the users completed the game in 14 to 15 steps, First Optimization required significantly more steps (36), whereas Second Optimization managed with 17 steps, showing a marked improvement. At Level 4, the users took 17 to 20 steps, similar to the algorithm's performance, where Second Optimization required 18 steps and First Optimization took 19. Second Optimization demonstrated better efficiency at higher levels, although human players outperformed the algorithm at Level 3.

4. DISCUSSION

This study has demonstrated significant results in implementing the backtracking algorithm for solving the puzzle sort game with high efficiency and speed. The research expands the scope of backtracking applications in game-based problem-solving by introducing a new puzzle model. Compared to previous studies, the findings of this research show notable improvements.

The optimization techniques applied in this study also profoundly impacted reducing the number of steps required to solve the puzzle. For example, integrating heuristic-based pruning reduced redundant explorations by approximately 30%, as seen in Level 3, where the algorithm's steps decreased from 36 (using only heuristics) to 17 when combined with the lookahead strategy. Additionally, the time required to complete the puzzle for Level 4 improved significantly, dropping from 8.13 milliseconds (with heuristics) to 3.56 milliseconds (with heuristics and lookahead), representing a substantial reduction of 56.2% in execution time. When the backtracking algorithm is compared to a second optimization implementation that combines the lookahead strategy and the conflict heuristic, it reduces the execution time at Level 4 by 91.4% (from 41.21 ms to 3.56 ms) and the steps by 76.9% (from 78 to 18). These enhancements are consistent across levels, showcasing how the algorithm becomes more efficient with the combined strategies. At Level 1, steps decreased from 8 to 7, while maintaining the same execution time of 4.58 milliseconds. However, in Level 2, the addition of lookahead slightly increased the steps from 12 to 14 but allowed the algorithm to handle the puzzle's complexity with better exploration patterns. Level 3 demonstrated the most significant step reduction, reinforcing the effectiveness of the combined strategy for handling more intricate configurations.

This demonstrates how these enhancements allow the algorithm to focus on promising paths, avoid unnecessary backtracking, and adapt to varying complexities, significantly increasing its efficiency for complex puzzle configurations. These findings highlight the broader implications of this research. By demonstrating the flexibility of the backtracking algorithm in solving puzzle-based games with optimized performance, this study paves the way for its application in other domains that involve complex decision-making and large search spaces. Examples include scheduling problems, optimization in logistics, and even AI-based problem-solving in robotics or automated planning systems. The demonstrated reduction in computational time and steps suggests that the methods introduced in this study can be adapted to address challenges in these fields, where efficiency and strategic exploration are critical.

5. CONCLUSION

The research results show that the backtracking algorithm’s performance declines as the game’s level and complexity increase, with execution time rising significantly from 3.41 ms at early levels to 41.21 ms and 78 steps at Level 4. However, applying the second optimization, which integrates lookahead strategies and conflict heuristics, reduced execution time at Level 4 by 91.4% (from 41.21 ms to 3.56 ms) and steps by 76.9% (from 78 to 18). The improvement of the second optimization shows the substantial impact of these strategies in limiting excessive exploration and accelerating the solution process, making the second optimization a more practical approach for enhancing the backtracking algorithm's efficiency in solving logic-based puzzles.

This study contributes significantly to developing backtracking algorithms by introducing a practical optimization framework that improves their applicability to logic-based puzzles. The findings demonstrate that the optimized algorithm can efficiently handle increasingly complex levels, paving the way for its implementation in other areas such as automated problem-solving, logic circuit design, and intelligent systems.

Future research could focus on developing more adaptive algorithms by integrating artificial intelligence techniques, such as reinforcement learning, to dynamically adjust strategies based on the puzzle's complexity. Additionally, exploring hybrid methods that combine backtracking with other optimization approaches, such as genetic algorithms or simulated annealing, could further enhance performance and broaden the scope of applications for logic-based games and beyond.

ACKNOWLEDGEMENTS

The author would like to express sincere gratitude to everyone who contributed to the completion of this journal. Special thanks are extended to the research team and volunteers, comprising both faculty members and students, for their significant support throughout this study. The author also wishes to acknowledge the Computer Science Department, LPPM, and ITH for their invaluable facility support, which played a crucial role in the success of this research.

REFERENCES

[1] N. A. Hasanah, L. Atikah, D. Herumurti, and A. A. Yunanto, "A Comparative Study: Ant Colony Optimization Algorithm and Backtracking Algorithm for Sudoku Game," in *2020 International Seminar on Application for Technology of Information and Communication (iSemantic)*, Semarang, Indonesia: IEEE, Sep. 2020, pp. 548–553. doi: 10.1109/iSemantic50169.2020.9234267.

[2] N. Nurdin, R. Suhartono, and T. Wibowo, "The Implementation of Backtracking Algorithm on Crossword Puzzle Games Based on Android," *J. Phys.: Conf. Ser.*, vol. 1363, no. 1, p. 012075, Nov. 2019, doi: 10.1088/1742-6596/1363/1/012075.

[3] T. Ito, H. Nakamura, and K. Tanaka, "Sorting balls and water: Equivalence and computational complexity," *Theoretical Computer Science*, vol. 978, p. 114158, Nov. 2023, doi: 10.1016/j.tcs.2023.114158.

[4] D. Chen, F. Zou, R. Lu, and S. Li, "Backtracking search optimization algorithm based on knowledge learning," *Information Sciences*, vol. 473, pp. 202–226, Jan. 2019, doi: 10.1016/j.ins.2018.09.039.

[5] M. Esteve, J. J. Rodriguez-Sala, J. J. Lopez-Espin, and J. Aparicio, "Heuristic and Backtracking Algorithms for Improving the Performance of Efficiency Analysis Trees," *IEEE Access*, vol. 9, pp. 17421–17428, 2021, doi: 10.1109/ACCESS.2021.3054006.

[6] P. Garg, A. Jha, and K. A. Shukla, "Randomised Analysis of Backtracking-based Search Algorithms in Elucidating Sudoku Puzzles Using a Dual Serial/Parallel Approach," in *Inventive Computation and Information Technologies*, vol. 336, S. Smys, V. E. Balas, and R. Palanisamy, Eds., Singapore: Springer Nature Singapore, 2022, pp. 281–295. doi: 10.1007/978-981-16-6723-7_21.

[7] V. Vidal, "A Lookahead Strategy for Heuristic Search Planning," 2004.

[8] M. N. I. Siddique, M. J. Rana, M. Shafiullah, S. Mekhilef, and H. Pota, "Automating distribution networks: Backtracking search algorithm for efficient and cost-effective fault management," *Expert Systems with Applications*, vol. 247, p. 123275, Aug. 2024, doi: 10.1016/j.eswa.2024.123275.

[9] A. Mehmood, P. Shi, M. A. Z. Raja, A. Zameer, and N. I. Chaudhary, "Design of backtracking search heuristics for parameter estimation of power signals," *Neural Comput & Applic.*, vol. 33, no. 5, pp. 1479–1496, Mar. 2021, doi: 10.1007/s00521-020-05029-9.

[10] K. Okumura, M. Machida, X. Défago, and Y. Tamura, "Priority inheritance with backtracking for iterative multi-agent path finding," *Artificial Intelligence*, vol. 310, p. 103752, Sep. 2022, doi: 10.1016/j.artint.2022.103752.

[11] S. Nofal, K. Atkinson, and P. E. Dunne, "Looking-ahead in backtracking algorithms for abstract argumentation," *International Journal of Approximate Reasoning*, vol. 78, pp. 265–282, Nov. 2016, doi: 10.1016/j.ijar.2016.07.013.

[12] F. Gebali, M. Taher, A. M. Zaki, M. W. El-Kharashi, and A. Tawfik, "Parallel Multidimensional Lookahead Sorting Algorithm," *IEEE Access*, vol. 7, pp. 75446–75463, 2019, doi: 10.1109/ACCESS.2019.2920917.

[13] M. Zhang and J. Lucas, "Lookahead Optimizer: k steps forward, 1 step back,"

33rd Conference on Neural Information Processing Systems, 2019.

[14] J. A. Abdulsaheb and D. J. Kadhim, “Classical and Heuristic Approaches for Mobile Robot Path Planning: A Survey,” *Robotics*, vol. 12, no. 4, p. 93, Jun. 2023, doi: 10.3390/robotics12040093.

[15] M. D. Pratama, R. Abdillah, D. Herumurti, and S. C. Hidayati, “Algorithmic Advancements in Heuristic Search for Enhanced Sudoku Puzzle Solving Across Difficulty Levels,” vol. 5, no. 4, 2024.