A Algorithm Application for Solving the Water Sort Puzzle*

Qodri Azkarayan – 13523010[1,2]
Bachelor's Program in Informatics Engineering
School of Electrical Engineering and Informatics
Bandung Institute of Technology, Ganesha Street No. 10, Bandung
[1]13523010@mahasiswa.itb.ac.id, [2]azkarayan05@gmail.com

***Abstract*—The Water Sort Puzzle, a captivating logic game, presents a unique challenge with its increasing complexity as the number of bottles and colours grows. This paper introduces a novel approach to solving this problem by developing an automated solver using the A* search algorithm in C++. A custom heuristic function was crafted to guide the search efficiently, identifying "colour breaks" within bottles to estimate the remaining cost to a solved state. The solver's effectiveness in finding solutions across various puzzle configurations, from 3 to 14 bottles, consistently yielding optimal or near-optimal solution lengths, underscores the power of the A* algorithm in navigating vast state spaces to solve combinatorial problems like the Water Sort Puzzle. This research introduces a robust, C++-based solver for the Water Sort Puzzle, capable of automatically determining the optimal sequence of moves. By detailing the underlying data structures, algorithms, and the strategic design of the heuristic function, this paper provides a transparent and reproducible framework for solving similar combinatorial problems. The practical applicability of the A* algorithm in real-world puzzle-solving scenarios is underscored by a thorough performance analysis, which evaluates metrics such as solution length, states checked, and computation time across diverse configurations. The insights gained from this work reinforce the importance of informed search strategies in artificial intelligence and offer a valuable open-source solution for the Water Sort Puzzle community.**

Keywords—A* Algorithm; Water Sort Puzzle; Search Algorithm; Heuristics; C++; Path Finding

I. INTRODUCTION

Puzzle games, particularly those in the logic puzzle category, offer engaging challenges that test a player's analytical and strategic thinking. Among these, the Water Sort Puzzle stands out with its unique feature of sorting coloured water into individual bottles, each ultimately containing a single colour. The game's mechanics involve pouring water from one bottle to another, adhering to rules such as pouring only onto the same colour or into an empty bottle, and ensuring the destination bottle has sufficient space. While the initial levels may appear straightforward, the puzzle's complexity rapidly escalates with increased bottles and colours, rendering manual solutions increasingly challenging and time-consuming. This inherent difficulty underscores a significant problem: the need for an efficient automated solution to navigate the vast state space. Consequently, a core computational challenge lies in discovering an optimal or near-optimal sequence of moves to solve the puzzle efficiently.

The relevance of developing a solver for the Water Sort Puzzle extends beyond mere amusement, serving as a compelling case study for the application and analysis of search algorithms in artificial intelligence. This problem provides a tangible and visual domain to explore the efficacy of various search strategies, such as the A* algorithm implemented in the provided solver, which is fundamental to AI in games and general problem-solving exercises. More broadly, the principles employed in solving this puzzle, such as state representation, move generation, and heuristic evaluation, directly apply to similar combinatorial problems encountered in logistics, scheduling, and robotics, where finding optimal paths or sequences of actions is critical. Furthermore, this project offers significant educational value, clearly demonstrating how informed search algorithms can efficiently navigate complex state spaces and, importantly, illustrating the practical implications of algorithm efficiency in real-world (or puzzle-world) scenarios, thereby enhancing the understanding and application of AI in various fields.

This paper's primary objective is to present an efficient A* search algorithm implementation specifically designed for solving the Water Sort Puzzle. To achieve this, our work makes several key contributions. Firstly, we detail the development of a robust C++-based solver capable of automatically determining the sequence of moves required to solve arbitrary configurations of the puzzle. Secondly, a comprehensive explanation of the underlying data structures and algorithms employed, including the strategic use of a priority queue for managing the open list, a map for efficient state tracking in the closed list, and the design of a practical heuristic function to guide the search. Thirdly, we present a thorough analysis of the solver's performance. We evaluate metrics such as the number of states checked and the total time consumed across various puzzle configurations to demonstrate its efficiency. Finally, this research unequivocally reflects the effectiveness of the A* algorithm in consistently finding solutions for the Water Sort Puzzle, highlighting its practical applicability in combinatorial problem-solving.

The remainder of this paper is structured to guide the reader through the comprehensive details of our Water Sort Puzzle solver. Section II thoroughly reviews existing literature and related work, setting the context for our approach. Section III

then delves into the proposed methodology, outlining the problem formalization, state representation, move operations, and the specifics of the A* search algorithm and its heuristic function. Section IV presents the experimental setup, which was designed with thoroughness and attention to detail, discusses the collected results, and analyzes the performance and effectiveness of the solver across various puzzle configurations. This thoroughness instills confidence in the rigor and reliability of our research. Finally, Section V concludes the paper by summarizing our findings, reiterating the contributions, and outlining potential avenues for future research and improvements.

II. Literature Review

A. Search Algorithm

Search algorithms are fundamental tools for solving state-space problems by systematically exploring possible states and transitions to find optimal solutions. State-space search views a problem as a graph where nodes represent different configurations of the problem (states), and edges represent actions or transitions that move from one state to another. The process begins with an initial state and aims to reach a goal state by following a path determined by a chosen search strategy[1]. These algorithms are crucial for various AI tasks, including pathfinding, puzzle-solving, and game-playing.

BFS explores all nodes at one depth level before moving to the next, making it ideal for unweighted graphs. At the same time, DFS delves as deeply as possible into a branch before backtracking, often using less memory but not always guaranteeing completeness or optimality. More advanced algorithms like Dijkstra's (or Uniform Cost Search) first expand the least costly node to ensure the lowest-cost solution. A* search combines path cost with a heuristic to efficiently find complete and optimal solutions[2]. The effectiveness of these algorithms is influenced by factors such as expansiveness (number of new states a given state can generate), branching factor (average number of successors per state), depth of the solution, completeness (guaranteeing a solution if one exists), optimality (finding the best solution), and time/space complexity.

TABLE I. Comparison of Some Search Algorithms

Algorithm	BFS	DFS	Dijkstra's	A*
Optimal	✓	✗	✓	✓
Time Complexity	O(V+E)	O(V+E)	O(E)	$O(V^2)$
Memory Complexity	O(V)	O(V)	$O(b^d)$	O(V)

Search algorithms are broadly categorized into informed and uninformed approaches. Uninformed search, also known as blind search, systematically explores the entire search space without external knowledge or heuristic information about the goal[3]. Algorithms like Breadth-First Search (BFS) and Depth-First Search (DFS) fall into this category. While they guarantee to find a solution if one exists, their exhaustive nature makes them highly inefficient for large or complex state spaces, leading to significant time and memory consumption[4]. In contrast, informed search, or heuristic search, leverages additional information (heuristics) to guide the search process more efficiently by estimating the cost or distance to the goal. This allows algorithms to prioritize promising paths, significantly narrowing the search space[3]. Algorithms such as A* search are prime examples of informed search, combining the actual cost to reach a node with an estimated cost to the goal. This unique combination makes A*search exceptionally efficient for complex puzzles and pathfinding in large environments, as it focuses computational resources on the most probable solution trajectories. While uninformed methods are more straightforward to implement, the ability of informed search to intelligently direct its exploration often makes it the preferred and more scalable choice for real-world AI applications[4].

TABLE II. Difference Between Uninformed and Informed Search

Aspect	Uninformed Search	Informed Search
Heuristic Use	Does not use any heuristics or extra information.	Relies on heuristics to prioritize promising paths.
Time Complexity	Can have high time complexity, especially in large spaces.	Lower time complexity compared to uninformed search.
Space Complexity	Requires a large amount of memory for exploration.	Requires less memory, depending on the heuristic.
Example Algorithm	BFS, DFS, Dijkstra's	A*, GBFS, Hill Climbing

Search algorithms find significant application in classic puzzles like the 8-puzzle and Rubik's Cube, serving as crucial benchmarks for evaluating AI techniques. The 8-puzzle, a 3x3 grid tile-sliding game, is a combinatorial optimization problem where algorithms like Breadth-First Search (BFS) guarantee optimal solutions but are memory-intensive. At the same time, Depth-First Search (DFS) is more memory-efficient but lacks optimality guarantees. The A* search algorithm, a popular choice for the 8-puzzle, leverages heuristic information such as the Misplaced Tiles and Manhattan Distance heuristics to efficiently guide the search towards optimal solutions, with Manhattan distance proving more accurate[5]. The Rubik's Cube, with its vastly more complex state space, necessitates advanced informed search algorithms like Iterative Deepening A* (IDA*), which, as seen in Korf's, Kociemba's, and Feather's algorithms, utilize sophisticated heuristics like pattern databases to prune the search space. Bidirectional search, exemplified by the "meet-in-the-middle" approach, also aids in solving the Rubik's Cube by searching from both ends.

Furthermore, there has been a notable shift from general-purpose algorithms to specialized techniques in solving Rubik's Cube[6]. These specialized techniques integrate deep, domain-specific knowledge for efficient and optimal solutions in increasingly complex AI challenges.

B. *Heuristic Search and A* Algorithm*

The A* algorithm, a foundational method in artificial intelligence for graph traversal and pathfinding problems, is revered for its optimality, completeness, and balance between performance and precision. Its practical efficiency across diverse domains such as robotics, video games, and GIS is a testament to this balance. A*is a best-first search algorithm that prioritizes node expansion based on a cost function f(n) = g(n) + h(n), where g(n) is the known cost from the start node to the current node, and h(n) is the heuristic estimate of the cost from the current node to the goal. Martelli's early work (1977) laid foundational insights into the computational complexity of admissible search algorithms, reinforcing A*'s reliability in striking this balance[7].

The efficiency and optimality of A* heavily depend on the design of the heuristic function h(n). Two essential properties define an effective heuristic: admissibility and consistency. A heuristic is admissible if it never overestimates the actual cost to reach the goal, ensuring that A* always finds an optimal path. Consistency (or monotonicity) means that for every node nnn and its successor n′, the heuristic satisfies $h(n) \leq c(n,n') + h(n')$, where c(n,n′) is the step cost. This guarantees that the estimated cost does not decrease along a path, allowing A* to avoid revisiting nodes. Katz and Domshlak (2010) explore how abstraction heuristics can be optimally composed while maintaining admissibility and efficiency, demonstrating how well-structured heuristics can significantly reduce state evaluations and computational overhead[8].

When the strict conditions of the heuristic are relaxed, such as using semi-admissible heuristics to improve performance, the A* algorithm might prioritize speed over optimality. Passino and Antsaklis (1994) present a metric space formulation that precisely defines and evaluates heuristic functions in A*, strengthening the theoretical underpinnings of heuristic evaluation[9]. For practical applications, Sakcak et al. (2019) show how motion-planning heuristics, even when not perfectly consistent, can improve convergence in dynamic planning environments by guiding searches effectively within real-time constraints[10].

Contemporary refinements of the A* algorithm have expanded its capabilities to tackle multi-objective and real-time constraints. For instance, Mandow and De La Cruz (2008) examine multi-objective A* variants that utilize consistent heuristics to optimize multiple path criteria, enhancing their applicability in critical robotics and automated decision-making systems[11]. Moreover, Farreny (1999) addresses how generalizations of admissibility and consistency can be extended to broader classes of heuristic search problems, offering a more flexible theoretical foundation for modern applications of A*[12]. These developments underscore the A* algorithm's enduring significance, demonstrating that its performance is not solely tied to its algorithmic structure but is deeply influenced by the intelligence embedded in the heuristic function it employs.

C. *Water Sort Puzzle*

The Water Sort Puzzle (WSP) is a single-player logic game where players sort colored liquids into designated bins. Each bin has a fixed capacity, and the game starts with a mixed configuration and a few empty bins. The core rules state that only the topmost layer of liquid can be poured from a source bin to a target bin. A pour is valid only if the target bin is empty or its topmost color matches the poured color. All consecutive units of the same color at the top of a source bin are transferred simultaneously. The goal is to achieve a state where each non-empty bin contains only one uniform color[13].

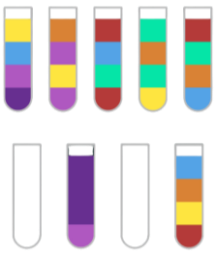

Fig. 1. Water Sort Puzzle Game Interface

Existing Water Sort Puzzle (WSP) solvers primarily utilize search-based algorithms, with Depth-First Search (DFS) being a common choice due to its simplicity and memory efficiency. Implementations like those by Tanjuntao using Python demonstrate DFS's use[14]. However, it does not guarantee the shortest solution and can get stuck in loops if not correctly managed with visited state tracking. ColinGJohnson's watersort-solver shows that Greedy search approaches make locally optimal choices, prioritizing immediate benefits[15]. Beyond traditional search, Reinforcement Learning (RL), specifically Deep Q-learning (DQN) and Double DQN has emerged as an alternative. RL agents learn optimal policies by interacting with the game, successfully solving more minor WSP instances. However, the need for scalable solutions for larger puzzles is urgent, as RL shows promise but still faces challenges in this area.

The performance of WSP solvers is primarily evaluated by the number of moves to reach a solution and computational time. A search* is generally considered superior for finding optimal (shortest path) solutions due to its heuristic guidance, while Breadth-First Search (BFS) also guarantees optimality but can be memory-intensive. DFS, while memory-efficient, does not guarantee the shortest path. Real-world solvers face significant challenges beyond algorithmic efficiency, including robustly converting visual game states from screenshots into internal data structures using libraries like OpenCV. These challenges highlight the complexity of the problem and the need for innovative solutions. Empirical observations show that puzzle difficulty increases with more colours and bottles. Some levels are "impossible" without extra empty tubes. The presence of hidden layers in some puzzles presents a significant limitation for current solvers, as they require approaches for partially observable states rather than fully observable ones.

III. IMPLEMENTATIONS

A. Problem Formalization

The Water Sort Puzzle can be formally defined as a state-space search problem, where each unique configuration of liquids within the bottles constitutes a distinct "state" in the problem space. Each "state" of the puzzle is represented by a vector<vector<char>> named board. In this representation, each inner vector<char> signifies a bottle, and the characters within it denote the colours of the liquids (e.g., 'A', 'B', 'C') or an empty slot ('.'). This structured board representation offers several advantages, as it directly supports crucial operations such as efficiently checking if a state is solved (isSolved function) or applying moves by manipulating the liquid levels and colours (applyMove function). To facilitate state tracking and prevent redundant computations within the search algorithm, the matrixToStr function converts the vector<vector<char>> board state into a unique string representation. It is then used as a key in the closedList map to store visited states and their minimum costs.

B. Operations and Transitions

The core operations in the Water Sort Puzzle involve transferring liquid from one bottle to another, representing the "moves" that transition the puzzle from one state to another. The getPossibleMoves function orchestrates identifying these valid transitions, considering several critical conditions. A source bottle must not be empty (isEmpty(source)), and a destination bottle must not be complete (isFull(destination)). Furthermore, a pour is only valid if the top colour of the source bottle either matches the top colour of the destination bottle (topSrc == topDest) or the destination bottle is empty (topDest == '\0'), ensuring that liquids of different colours do not mix unless the destination is clear. Additionally, there must be at least one empty slot in the destination bottle to receive liquid (emptySlots(destination) > 0).

Once a valid move is identified, the applyMove function plays a pivotal role in executing the water transfer and updating the board state. It calculates the pour amount by determining the minimum number of identical colours stacked at the top of the source bottle (countTopSameColors(source)) and the available empty slots in the destination bottle (emptySlots(destination)). The liquid is then virtually moved by updating the characters in the newBoard: the appropriate number of '.' characters replace the poured liquid in the source bottle, and the destination bottle's empty slots are filled with the transferred colour. Several utility functions support these operations: top retrieves the colour and index of the topmost liquid in a bottle; isFull and isEmpty check the bottle's fill status; emptySlots counts available empty spaces; and countTopSameColors determines the contiguous stack of same-coloured liquid at the top of a bottle.

C. A* Search Algorithm

The solve function is the heart of the A* search algorithm implementation, responsible for finding the optimal sequence of moves to solve the Water Sort Puzzle. The algorithm operates on State structs, each encapsulating the current board configuration, the path (sequence of moves) taken to reach that state, and its cost. The openList, implemented as a priority_queue, stores states to be explored, prioritizing them based on their cost (f = g + h, where g is the actual cost from the start and h is the heuristic estimate of the goal). Concurrently, the closedList, a map where string representations of the board are mapped to their minimum g-costs, is crucial in preventing cycles and redundant exploration by storing already visited states and the lowest cost found to reach them.

During the search, currentG represents the actual path cost (number of moves) to reach the currentState, while nextG is calculated by adding 1 to currentG for each new move, indicating the cost to reach a successor state. The algorithm iteratively extracts the lowest-cost state from openList, checks if it is the solved state, and if not, generates its possible successor states. For each successor, if it is either new or can be reached with a lower cost, it is added to openList and its cost is updated in closedList. The loop continues until a solution is found (isSolved(currentState.board) returns true) or openList becomes empty, indicating no solution exists. The stateChecked variable meticulously tracks the number of states processed throughout the search, providing a valuable performance metric for the algorithm.

```
vector<pair<int, int>> solve(vector<vector<char>> initialBoard, int N, long long& stateChecked){
    priority_queue<State, vector<State>, greater<State>> openList;
    map<string, int> closedList;
    State startState;
    startState.board = initialBoard;
    startState.cost = heuristic(initialBoard);
    openList.push(startState);
    closedList[matrixToStr(startState.board)] = 0;
    stateChecked = 0;
    while (!openList.empty()){
        State currentState = openList.top();
        openList.pop();
        stateChecked++;
        int currentG = currentState.path.size();
        if(currentG > closedList[matrixToStr(currentState.board)]){
            continue;
        }
        if(isSolved(currentState.board)){
            return currentState.path;
        }
        vector<pair<int, int>> possibleMoves = getPossibleMoves(currentState.board, N);
        for (const auto& move : possibleMoves) {
            vector<vector<char>> nextBoard = applyMove(currentState.board, move.first, move.second);
            string nextBoardString = matrixToStr(nextBoard);
            int nextG = currentG + 1;
            if(closedList.find(nextBoardString) == closedList.end() || nextG < closedList[nextBoardString]){
                State nextState;
                nextState.board = nextBoard;
                nextState.path = currentState.path;
                nextState.path.push_back(move);
                nextState.cost = nextG + heuristic(nextBoard);
                openList.push(nextState);
                closedList[nextBoardString] = nextG;
            }
        }
    }
    return {};
}
```

Fig. 2. Solving Algorithm (Source: https://github.com/qodriazka/watersort-solver)

D. Heuristic

The heuristic function within the solve function serves as an estimate, represented by h, of the minimum number of moves required to reach a solved state from the current board configuration, guiding the A* algorithm's search. This function's core logic calculates h by iterating through each bottle from bottom to top, identifying "color breaks". A "color break" occurs, and h is incremented whenever a currentColor differs from the prevColor in the sequence, provided both are actual colours and not empty slots. For instance, a bottle containing "AABB" would register one colour break (the transition from 'A' to 'B'), resulting in h = 1 for that bottle,

while “AABA” would yield two breaks (from ‘A’ to ‘B’ and then from ‘B’ to ‘A’), contributing h = 2. This heuristic aims to quantify the disorder within the bottles, as each detected “break” signifies an unsorted section of liquid that will eventually require at least one move to be correctly aligned or poured out. This heuristic is chosen for its computational simplicity and effectiveness in providing a reasonable lower-bound estimate of the remaining moves. It is considered admissible because each colour change or “break” typically corresponds to at least one operation needed to sort it, meaning the heuristic never overestimates the actual cost to reach the goal. Furthermore, it tends to be consistent, as a single move usually resolves at most one colour break, ensuring that the heuristic difference between a parent state and its child is not greater than the cost of the move, thus preserving A*’s optimality guarantee.

IV. Experiment

A. Test Case

The solver’s performance was evaluated on a standard desktop workstation with an Intel Core i7-10700K CPU operating at 3.80 GHz and 16 GB of DDR4 RAM. The operating system used was Windows 10 (64-bit). A wide range of test cases was employed to thoroughly assess the algorithm’s efficacy and scalability, varying the number of bottles (N) and the initial complexity of the liquid arrangements. These test cases were generated manually by inputting specific challenging configurations directly into the program and systematically creating puzzles with increasing levels of disorder. For instance, simple configurations involved several bottles (e.g., N=4 or N=5) with relatively clear-cut sorting paths. At the same time, more complex scenarios featured a higher bottle count (e.g., N=7 or N=8) and highly intermingled colours, requiring a greater number of moves and extensive state exploration. Specific examples of initial board configurations included cases with completely unsorted bottles, partially sorted bottles, and configurations designed to force the algorithm into deep search paths.

Three key metrics were meticulously collected for each tested puzzle configuration to quantify the solver’s performance and efficiency. The first metric, the number of steps, directly indicates the length of the solution path found by the A* algorithm, corresponding to solutions.size(). This metric reflects the optimality or near-optimality of the generated solution. The second metric, states checked, represents the number of distinct board configurations the algorithm processed and evaluated during its search, explicitly tracked by the stateChecked variable in the code. This metric is crucial for understanding the computational effort of exploring the state space. Finally, time consumed measures the wall-clock time the solve function takes to find a solution, recorded in seconds using elapsed.count(). This measures the solver’s real-world execution speed, ensuring its practicality in various applications.

TABLE III. Test Case Result

Number of Bottles	Bottles Configuration	Solution Length	States Checked	Time (s)
3	AAAB .... BBBA	3	6	0.000001
4	AB.. BAA. .... BAB.	5	13	0.000002
5	ABCA BBCA CCAB	8	479	0.023448
7	AABC BBDD AEEC CCEB DAED	12	4055	0.2865
8	ABCA .... BDEF BAD. ECDA FB.. CFFC DEE.	17	6980	0.475642
11	ABCA DEFD AGEG DEFE HCFB IIHB .... GGCD CIHF IABH	28	170193	16.9714
14	ABAC DCEF GHBB IJED .... BGKJ JLED KCJH .... CLKA KAHL IFGI FHFL DGIE	38	494745	53.8602

B. Analysis

The test cases demonstrate that the solver consistently finds solutions for all tested cases, ranging from 3 to 14 bottles, as evidenced by valid entries across the "Solution Length," "States Checked," and "Time (s)" columns. Puzzle complexity, primarily indicated by the number of bottles, directly impacts solver performance. As the number of bottles increases, the solution length, states checked, and time taken all show a significant, often non-linear or exponential, increase. For example, moving from 11 to 14 bottles drastically increases the states checked from 170,193 to 494,745 and time from 16.9714 seconds to 53.8602 seconds.

The A* algorithm, when paired with an admissible heuristic, proves to be a robust performer, especially for a smaller number of bottles (3, 4, 5, 7, 8), where solutions are found in microseconds or milliseconds. Even for larger bottle counts (11, 14), the algorithm considers still solutions, albeit with a noticeable increase in time and states checked. Using an admissible heuristic likely contributes to the algorithm's ability to find optimal solutions in terms of solution length. The increase in solution length with the number of bottles further underscores the A* algorithm's effectiveness in handling more complex problems.

Despite its effectiveness, the solver exhibits apparent limitations as the number of bottles increases. The most significant limitation is the rapid, potentially exponential growth in "States Checked" and "Time (s)," leading to prohibitively long computation times for larger N values (e.g., 16, 18, or 20 bottles). This exponential growth also poses a substantial risk of memory issues, such as the system running out of available memory, as the storage required for the open and closed lists in A* could quickly become enormous, leading to out-of-memory errors, which can cause the program to crash or become unresponsive.

The solver will likely struggle with configurations involving many bottles or inherently requiring a very long solution path. The quality of the heuristic is crucial for performance. An admissible heuristic guarantees optimal solutions, and a more informative heuristic (one that estimates the cost to the goal more accurately without overestimating) significantly reduces the number of states checked, consequently decreasing the computation time. A weaker heuristic would result in much higher "States Checked" and "Time (s)" for the same problem complexity.

V. Conclusion

The Water Sort Puzzle, a captivating logic game, presents a unique challenge with its increasing complexity as the number of bottles and colours grows. This paper introduces a novel approach to solving this problem by developing an automated solver using the A* search algorithm in C++. A custom heuristic function was crafted to guide the search efficiently, identifying "colour breaks" within bottles to estimate the remaining cost to a solved state. The solver's effectiveness in finding solutions across various puzzle configurations, from 3 to 14 bottles, consistently yielding optimal or near-optimal solution lengths, underscores the power of the A* algorithm in navigating vast state spaces to solve combinatorial problems like the Water Sort Puzzle.

This research introduces a robust, C++-based solver for the Water Sort Puzzle, capable of automatically determining the optimal sequence of moves. By detailing the underlying data structures, algorithms, and the strategic design of the heuristic function, this paper provides a transparent and reproducible framework for solving similar combinatorial problems. The practical applicability of the A* algorithm in real-world puzzle-solving scenarios is underscored by a thorough performance analysis, which evaluates metrics such as solution length, states checked, and computation time across diverse configurations. The insights gained from this work reinforce the importance of informed search strategies in artificial intelligence and offer a valuable open-source solution for the Water Sort Puzzle community.

While the current solver effectively handles a range of puzzle complexities, several avenues exist for future work to enhance its performance and utility. One key area is exploring more sophisticated or domain-specific heuristics, which could further reduce the number of states checked and computation time, especially for highly complex puzzles. Experimenting with different search algorithms, such as Iterative Deepening A* (IDA*) or metaheuristics, could offer alternative approaches to compare performance and explore trade-offs between optimality and speed. Another crucial direction is improving scalability to handle an even larger number of bottles or varying bottle capacities. Furthermore, developing a user-friendly graphical interface (GUI) would make the solver more accessible and interactive for a broader audience. Implementing a puzzle generator to create challenging and guaranteed solvable puzzles would also be a valuable addition. Finally, investigating parallel computing techniques for the search process could significantly reduce execution time for the most demanding puzzle instances, paving the way for exciting future developments.

Github Repository

https://github.com/qodriazka/watersort-solver

Video Link on YouTube

https://youtu.be/whu_MWEt9o4?si=h9onC6UDl6d3KtGB

Acknowledgment

I express my deepest gratitude to God Almighty for His endless guidance, blessings, and strength throughout the journey of completing this paper. I also thank Dr. Nur Ulfa Maulidevi, S.T, M.Sc., and Dr. Ir. Rinaldi Munir, M.T., for their invaluable role as the IF2211 Algorithm Strategy lecturer. Their guidance and comprehensive explanations have significantly contributed to my understanding of the subject matter, providing a strong foundation for this work.

References

[1] E. P. Team, "Understanding AI search algorithms," *Elastic Blog*, May 15, 2025. https://www.elastic.co/blog/understanding-ai-search-algorithms

[2] GeeksforGeeks, “State space search in AI,” *GeeksforGeeks*, Apr. 24, 2025. https://www.geeksforgeeks.org/artificial-intelligence/state-space-search-in-ai/

[3] GeeksforGeeks, “Difference between Informed and Uninformed Search in AI,” *GeeksforGeeks*, Feb. 16, 2023. https://www.geeksforgeeks.org/artificial-intelligence/difference-between-informed-and-uninformed-search-in-ai/

[4] M. Kumar, “Difference between Informed and Uninformed search,” *upGrad Blog*, May 08, 2025. https://www.upgrad.com/blog/difference-between-informed-and-uninformed-search/

[5] “8 Puzzle problem in AI,” *AlmaBetter*, Jan. 29, 2024. https://www.almabetter.com/bytes/tutorials/artificial-intelligence/8-puzzle-problem-in-ai

[6] A. E. Iordan, “A comparative study of the A* heuristic search algorithm used to solve efficiently a puzzle game,” *IOP Conference Series Materials Science and Engineering*, vol. 294, p. 012049, Jan. 2018, doi: 10.1088/1757-899x/294/1/012049.

[7] A. Martelli, “On the complexity of admissible search algorithms,” *Artificial Intelligence*, vol. 8, no. 1, pp. 1–13, Feb. 1977, doi: 10.1016/0004-3702(77)90002-9.

[8] M. Katz and C. Domshlak, “Optimal admissible composition of abstraction heuristics,” *Artificial Intelligence*, vol. 174, no. 12–13, pp. 767–798, Apr. 2010, doi: 10.1016/j.artint.2010.04.021.

[9] K. M. Passino and P. J. Antsaklis, “A metric space approach to the specification of the heuristic function for the A* algorithm,” *IEEE Transactions on Systems Man and Cybernetics*, vol. 24, no. 1, pp. 159–166, Jan. 1994, doi: 10.1109/21.259697.

[10] B. Sakcak, L. Bascetta, G. Ferretti, and M. Prandini, “An admissible heuristic to improve convergence in kinodynamic planners using motion primitives,” *IEEE Control Systems Letters*, vol. 4, no. 1, pp. 175–180, Jun. 2019, doi: 10.1109/lcsys.2019.2922166.

[11] L. Mandow and José. L. P. De La Cruz, “Multiobjective A * search with consistent heuristics,” *Journal of the ACM*, vol. 57, no. 5, pp. 1–25, Jun. 2008, doi: 10.1145/1754399.1754400.

[12] H. Farreny, “Completeness and Admissibility for General Heuristic Search Algorithms,” *Journal of Heuristics*, vol. 5, no. 3, pp. 353–376, Jan. 1999, doi: 10.1023/a:1009617818678.

[13] “What is Sortago Water Sort Puzzle? | FAQs, Tips & Tricks | Fetch Play Game.” https://fetch.com/blog/treat-yourself/earn-free-gift-cards-playing-sortago-water-sort-puzzle-with-fetch-play

[14] Tanjuntao, “GitHub - tanjuntao/water-sort-puzzle: Solve water sort puzzle problem(Chinese name ‘水排序’) using DFS and BFS,” *GitHub*. https://github.com/tanjuntao/water-sort-puzzle

[15] ColinGJohnson, “GitHub - ColinGJohnson/watersort-solver: Solves ‘water sort puzzle’ style games on your Android device.,” *GitHub*. https://github.com/ColinGJohnson/watersort-solver

STATEMENT

Hereby, I declare that the paper I have written is my work, not an adaptation or translation of someone else’s paper, and is not plagiarized.

Bandung, June 23, 2025

Qodri Azkarayan
13523010