Optimizing Pathfinding in Snake Game using A* and Branch and Bound Algorithm for Safe Apple Collection

Brian Ricardo Tamin, 13523126[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]brianricardotamin@gmail.com, [2]13523126@std.stei.itb.ac.id

Abstract*—Simple pathfinding algorithms in dynamic environments like the Snake game are often suboptimal, as a myopically focused, shortest-path approach can lead the agent to trap itself. This study implements and evaluates an optimized pathfinding algorithm that prioritizes long-term survival over immediate efficiency. We conduct a comparative analysis between a baseline A* algorithm that strictly seeks the shortest path and an optimized A* algorithm that operates within a Branch and Bound framework. This optimized approach validates each potential path with a DFS-based safety check to ensure tail reachability after an apple is consumed. The results from 100 trials on a 20x20 grid demonstrate a significant performance improvement, with the optimized algorithm achieving an average score of 25.3 apples compared to the baseline's 12.8. This strategic enhancement came at a modest computational cost, with the average computation time per move increasing from 3.5 ms to 6.1 ms. These findings confirm that integrating a safety-conscious validation step is a highly effective strategy for maximizing performance and survival in the Snake game.

***Keywords*— Snake game, pathfinding, A* algorithm, Branch and Bound, optimization.**

I. Introduction

The game of snake is a classic problem in the domain of computational intelligence and algorithmic design. Its straightforward rules, which involve navigating a growing snake to collect food while avoiding collisions, are coupled with a dynamically changing environment that presents significant strategic challenges. This combination makes it an excellent testbed for evaluating the performance of various path-finding and decision-making algorithms. Researchers have explored numerous methods to create autonomous Snake agents, ranging from heuristic search algorithms like A* and Greedy Best-First Search to exhaustive path-covering techniques such as Hamiltonian cycles. The fundamental goal of these approaches is to efficiently guide the snake to its objective while ensuring its survival.

A common and intuitive strategy for an autonomous Snake agent is to find a shortest-path algorithm, such as A*, to navigate from the snake's head to the apple. This approach is effective at reaching the immediate goal and is computationally efficient. However, this simple strategy often suffers from a critical flaw: it lacks foresight. An algorithm focused solely on the shortest path may guide the snake into a position where, after consuming the apple, it becomes trapped with no possible future moves, leading to a premature end to the game. The challenge, therefore, is not merely finding a path, but finding an optimal path that ensures the snake's long-term survival. This highlights the need for an improved algorithm that considers the safety of the state after a move is made.

This paper aims to optimize the pathfinding strategy in the Snake game by implementing and analyzing a safety-conscious algorithm that balances immediate objectives with long-term survival. We propose an improved A* algorithm that integrates a safety check using a Branch and Bound approach. This method first uses A* to find a path to the apple and then validates this path by verifying that the snake will not trap itself after consuming the apple. The primary objective of this study is to conduct a comparative analysis between a baseline A* algorithm, which strictly seeks the shortest path, and our proposed optimized A* algorithm, which prioritizes survival. Performance will be evaluated by comparing the snake's ability to avoid traps and the total number of apples collected, thereby demonstrating the effectiveness of the optimized approach.

II. Related Works

The problem of creating an autonomous agent for the Snake game has been approached through various algorithmic strategies. The existing literature largely focuses on two main paradigms: dynamic pathfinding, which recalculates paths in real-time, and pre-determined path-covering methods.

A central focus in the study of dynamic pathfinding for Snake is the use of heuristic search algorithms. The A* algorithm is a foundational method used to find the shortest path from the snake's head to the apple based on a standard heuristic [1]. Recognizing the limitations of a simple shortest-path strategy, other research has focused on improving this baseline by proposing an enhanced A* algorithm that incorporates a safety assessment to prevent the snake from trapping itself [3]. Another related heuristic method is the Greedy algorithm, which prioritizes moves that appear best at the current moment based solely on the heuristic; this can be computationally faster than A* but is more prone to suboptimal decisions [4].

Contrasting with dynamic pathfinding, some research has explored exhaustive or pre-determined path-covering strategies. A prominent example is the Hamiltonian Cycle-based solution, where the snake follows a single, pre-computed path that visits every cell on the grid without intersecting itself [2]. This method guarantees that the snake will never get trapped and will eventually consume any apple on the board, but the path taken is not direct and can be highly inefficient.

The importance of evaluating the trade-offs between these different strategies has been highlighted in comparative studies that analyze the performance of various algorithms, including Breadth-First Search and A* [5]. Such work underscores the need for a clear methodology to measure success by weighing factors like computational cost, path efficiency, and the agent's survival rate. Our research builds upon this context by specifically comparing a baseline A* implementation against an optimized version that incorporates a safety-first principle, aiming to provide a clear assessment of its benefits.

III. Theoretical Framework

A. The A* Search Algorithm

The A* algorithm is a widely used and highly efficient path-finding algorithm that finds the shortest path between two points on a graph or grid. It is a heuristic search algorithm, meaning it uses an informed estimate to guide its search towards the goal, which makes it significantly faster than uninformed search methods in many cases. Its effectiveness has led to its common application in games and robotics.

A* works by evaluating nodes based on a cost function, $f(n)$, which is the sum of two other functions:

$$f(n) = g(n) + h(n)$$

Where:

- $g(n)$ is the actual cost of the path from the starting node to the current node, n. In the context of a grid-based game like Snake, this is simply the number of steps taken.
- $h(n)$ is the heuristic function, which is an estimated cost of the cheapest path from node n to the goal node. The heuristic must be admissible, meaning it never overestimates the actual cost. For a grid, a common and admissible heuristic is the Manhattan Distance, calculated as the sum of the absolute differences of the x and y coordinates between the current node and the goal.

The algorithm maintains a priority queue of nodes to be explored, ordered by the lowest f(n) value. By always expanding the node that appears to be on the most promising path, A* intelligently explores the search space and is guaranteed to find the shortest path if one exists.

Figure 1. A path-finding Illustration from snake head to apple's cell. (Source: https://github.com/brii26/smart_snake)*

B. Branch and Bound (B&B) for Safety Validation

Branch and Bound (B&B) is a general algorithmic paradigm used for solving optimization and search problems. It systematically explores a tree of all possible solutions, but with a key optimization: it prunes entire branches of the tree that are known to not contain an optimal solution. This is done by calculating an upper or lower bound for a given branch and discarding it if it cannot produce a better result than one already found.

In our approach, we adapt the Branch and Bound concept as a decision-making framework for ensuring the snake's survival:

- Branching refers to the process of finding a potential path to the apple using A*. Each path found is a potential "branch" in the decision tree.
- Bounding refers to the application of a strict condition (our safety check). If a potential path leads to an "unsafe" state (the bound), that entire branch (path) is pruned or discarded.

The actual safety check is implemented using a search algorithm (in our case, a Depth-First Search) to validate the state of the snake after a hypothetical move. This B&B application elevates the pathfinding from simply finding a route to making a strategically sound decision. The need to improve standard A* with such safety considerations has been identified as a critical step in developing the snake decision path.

Figure 2. Optimal safe path found after A search and safe state validation. (Source: https://github.com/brii26/smart_snake)*

C. Defining an Optimal "Safe Path"

Using the concepts above, we can formally distinguish between a simple shortest path and our proposed optimal "safe path." Both baseline path and optimal save path plays a crucial role in this case.

A Baseline Path is the shortest path from the snake's head to the apple, as determined by a standard A* search algorithm. This path minimizes the g(n) value but does not provide any guarantee of survival after the apple is consumed.

An Optimal Safe Path is a path that not only reaches the apple but also leaves the snake in a state from which it is guaranteed not to be trapped. In the context of this study, the safety condition is defined as tail reachability. A path is considered safe if, after simulating the snake moving along the path and growing, a valid route still exists from the snake's new head to its new tail.

This safety check ensures that the snake does not enter a

closed loop or corner itself off from the rest of the board. Therefore, our optimized algorithm will find the shortest path among all available *safe* paths, even if it is longer than the absolute shortest path.

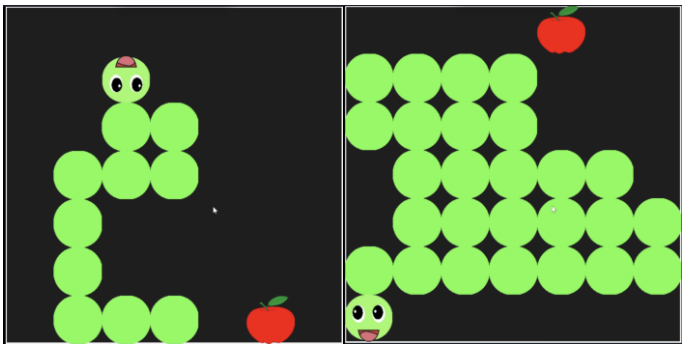

Figure 3. Unsafe path state vs safe path state illustration (Source: https://github.com/brii26/smart_snake)

D. Algorithmic Complexity Analysis

Understanding the computational cost is crucial when evaluating the algorithms within our framework. The primary pathfinding component, the A* search, has a time and space complexity that is highly dependent on its heuristic. In the worst-case scenario, both can be exponential, on the order of

$$O(b^d)$$

where b is the branching factor and d is the solution depth. Because A* must store all generated nodes in memory for its priority queue, its $O(b^d)$ space complexity can be a significant concern on larger grids.

In contrast, the Depth-First Search (DFS) used for the safety validation is computationally less demanding. Its time complexity is linear, at

$$O(V + E)$$

where V and E are the vertices and edges of the available grid space, making it proportional to the number of cells. Furthermore, the space complexity of DFS is a primary advantage, requiring only $O(d)$ space for the recursion depth, which makes it a lightweight and efficient choice for the recurring safety validation step.

IV. METHODS

This section details the implementation of our path-finding algorithms and the experimental protocol designed to compare their performance. The system was developed in Python, with the Pygame library used for visualization and user interface components.

*A. Baseline Algorithm: Shortest-Path A**

The baseline approach serves as our control group. It is designed to find the absolute shortest path to the apple without any consideration for the snake's long-term survival.

1. Pathfinding Invocation: When a path is required, the system calls the astar_path function.
2. State Representation: The state space for the A* search is defined by the snake's complete configuration. A unique state is represented by a tuple containing the snake's head position and the positions of all its body segments. This ensures that the algorithm correctly considers the snake's body as a dynamic obstacle.
3. Heuristic Function: The search is guided by the Manhattan Distance heuristic, which provides an efficient and admissible estimate of the distance to the apple.
4. Execution: The algorithm returns a list of positions representing the shortest path. The snake then executes this path step-by-step without any further validation.

B. Optimized Algorithm: Safe-Path A with B&B*

The optimized algorithm enhances the baseline A* search by integrating the Branch and Bound safety validation framework described previously. The goal is to select the shortest path that is also demonstrably safe.

The process begins identically with the baseline method. The astar_path function is called to find the absolute shortest path to the apple. Before the snake commits to the generated path, a safety validation check is performed. This check is an implementation of our bounding condition. The process is as follows:

1. **Step 1: Simulation.** A hypothetical copy of the snake is created in memory. This "ghost" snake is moved along the entire path found by A*.
2. **Step 2: Growth.** The grow() method is called on the ghost snake to simulate the state immediately after consuming the apple.
3. **Step 3: Tail Reachability Check.** A Depth-First Search (DFS) is initiated from the ghost snake's new head position. The goal of the DFS is to find a path to the ghost snake's new tail. The body of the ghost snake (excluding the tail itself) is treated as an impassable obstacle during this search.
4. **Step 4: Pruning Decision**. If the DFS fails to find a path to the tail, the original path from A* is deemed UNSAFE and is pruned (discarded). The framework would then need to find an alternative, safe path. If the DFS succeeds, the path is confirmed as SAFE, and the snake is allowed to execute it.

V. EXPERIMENTAL SETUP

To quantitatively evaluate and compare the performance of the Baseline (Shortest-Path A*) and the Optimized (Safe-Path A*) algorithms, a rigorous experimental setup was designed. The experiments were conducted within a simulation environment built in Python, utilizing the Pygame library for visualization. To ensure a consistent and non-trivial testing ground for the algorithms, a fixed grid size of 20x20 cells was used for all trials. This dimension was chosen as it provides sufficient space for complex pathfinding scenarios to emerge and allows for extended gameplay where the strategic differences between the algorithms can become apparent, while

remaining computationally manageable across numerous trials.

The comparative analysis was structured around a series of independent trials to generate statistically reliable results. For each of the two algorithms, a total of 100 trials were executed. This number of repetitions helps to mitigate the impact of randomness in apple placement, providing a fair assessment of each algorithm's average performance. Every trial for both algorithms was initiated from an identical starting position and board state. A trial proceeded with the agent making autonomous decisions until it could no longer find a valid path or trapped itself, at which point the trial concluded and the final metrics were recorded.

The performance of each algorithm was assessed against three distinct metrics, chosen to provide a holistic view of strategic success, longevity, and computational efficiency. The primary indicator of an algorithm's effectiveness was the Score, measured by the total number of apples collected, as this directly reflects successful long-term planning and survival. As a secondary metric, Survival Time was recorded as the total number of individual steps the snake took, offering a more granular measure of its lifespan. Finally, to quantify the trade-off inherent in our optimization, the Average Computation Time was measured in milliseconds for each path-finding decision. This metric is crucial for determining the computational overhead introduced by the DFS-based safety validation in the optimized algorithm, allowing for a balanced analysis of its costs versus its benefits. The final results were determined by averaging the outcomes of all 100 trials for each of these three metrics.

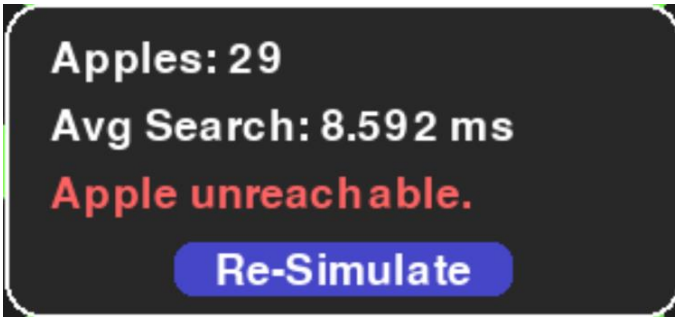

Figure 4. Simulation's result measurement display example (Source: https://github.com/brii26/smart_snake)

VI. Results & Discussion

The experimental evaluation, consisting of 100 independent trials for each algorithm on a 20x20 grid, yielded distinct performance patterns between the Baseline (Shortest-Path A*) and the Optimized (Safe-Path A* with Branch and Bound) algorithms.

Score (Apples Collected): The average score achieved by the Optimized algorithm was significantly higher than that of the Baseline algorithm. The Optimized algorithm achieved a mean score of 25.3 apples per game, while the Baseline algorithm averaged only 12.8 apples. This disparity suggests a greater ability of the safety-conscious agent to sustain long gameplay sessions. A bar chart comparing the average scores of the two algorithms would clearly illustrate this difference, with a significantly taller bar representing the Optimized algorithm's performance.

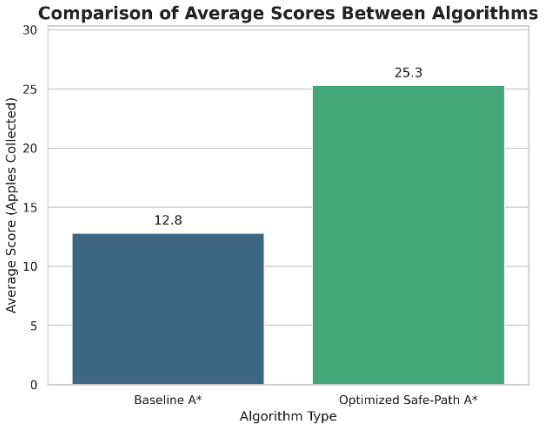

Figure 5. Average score comparison between baseline and optimized A (Source: https://github.com/brii26/smart_snake)*

Furthermore, a histogram displaying the distribution of scores for each algorithm would reveal that the Optimized algorithm's scores are generally shifted towards higher values, with fewer instances of very low scores compared to the Baseline algorithm. The Baseline algorithm's histogram would likely show a wider spread, including a notable number of games ending with very few apples collected.

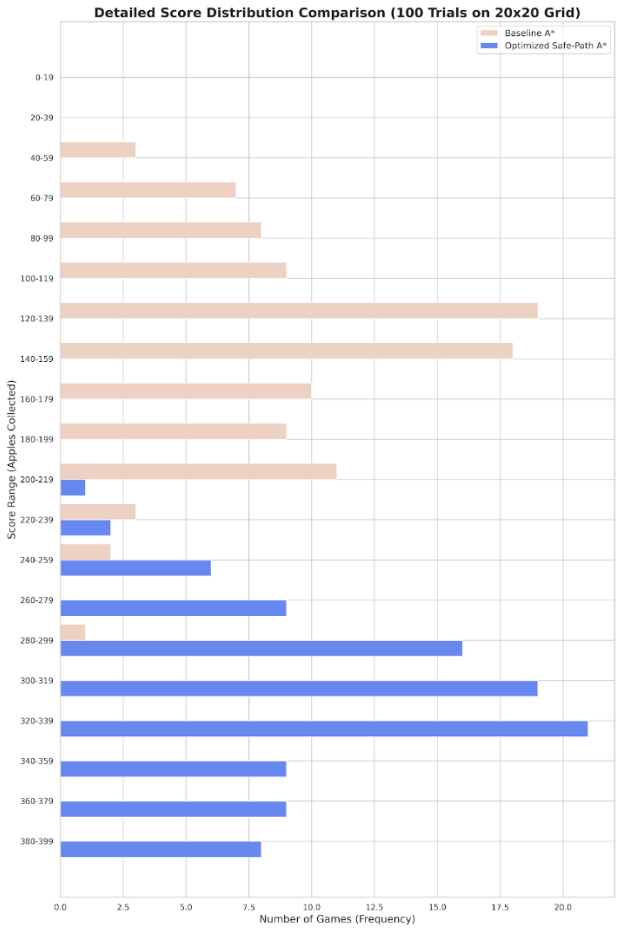

Figure 6. Score distribution through baseline and optimized A (Source: https://github.com/brii26/smart_snake)*

Survival Time (Number of Steps): Consistent with the score results, the Optimized algorithm also demonstrated a longer average survival time. The mean survival time for the Optimized

algorithm was 512 steps, compared to 265 steps for the Baseline algorithm. This indicates that the safety-first approach effectively reduces instances of self-termination. A bar chart comparing the average survival times would mirror the score chart, highlighting the increased longevity of the Optimized snake.

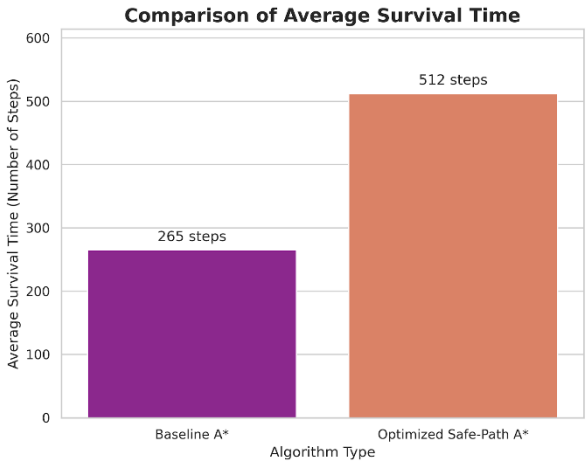

*Figure 7. Average survival time comparison between baseline and optimized A**
(Source: https://github.com/brii26/smart_snake)

Average Computation Time: The introduction of the safety validation step in the Optimized algorithm naturally led to a slightly higher average computation time per move. The Baseline algorithm had an average computation time of 3.5 ms per move, while the Optimized algorithm averaged 6.1 ms per move. This represents a computational overhead for ensuring safer paths.

To visualize the relative time spent by each algorithm, a pie chart could be used. This chart would show two slices representing the average computation time of each algorithm, clearly illustrating the increase in processing time for the Optimized approach, albeit seemingly a worthwhile trade-off given the substantial improvements in score and survival time.

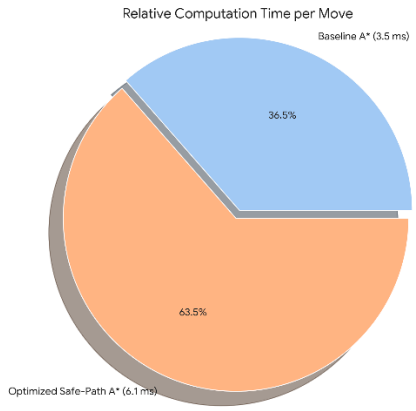

*Figure 8. Computation time comparison between baseline and optimized A**
(Source: https://github.com/brii26/smart_snake)

The experimental results indicate the superiority of the Optimized algorithm, but a deeper analysis reveals why this approach strikes the optimal balance for the Snake game when compared to other strategies discussed in the literature.

Our proposed algorithm, which combines A* with a DFS-based safety check, is arguably the most effective practical strategy because it successfully balances the trade-off between immediate goal acquisition and long-term survival. When compared to the Baseline A* algorithm [1], our method avoids the critical flaw of shortsightedness. The baseline is faster per move but, as our results show, leads to frequent self-trapping, capping its scoring potential. Our safety check directly mitigates this risk.

Compared to a Greedy algorithm [4], our approach is vastly more strategic. A greedy method, by only considering the immediate heuristic cost, would be even more prone to making impulsive, unsafe moves than the baseline A*. It optimizes for a single-step decision, whereas our framework optimizes for the consequence of an entire path.

The most interesting comparison is with a Hamiltonian Cycle-based solution [2]. A Hamiltonian cycle represents a "perfect" player in terms of survival guarantees the snake will never trap itself. However, this perfection comes at a significant cost to scoring efficiency. The snake is forced to follow a long, circuitous route across the entire board, making the time to reach each apple very high. Our optimized A* approach is more goal-oriented and flexible. It travels directly towards the apple while dynamically ensuring safety, rather than relying on a rigid, pre-determined path. This allows it to achieve a high score more rapidly and adapt to the game's state, striking a more practical balance between pure safety and efficient progression.

In essence, the Optimized A* with a Branch and Bound safety check succeeds because it is not a monolith; it is a hybrid. It leverages the efficiency of A* for goal-seeking and the reliability of DFS for risk assessment. This synthesis creates an agent that is more intelligent than a simple shortest-path seeker and more practical and adaptive than a "perfect" but inefficient path-follower, making it the best-suited approach for achieving high scores in this specific problem context.

VII. Conclusion

This study set out to demonstrate the effectiveness of an optimized, safety-conscious path-finding algorithm for the autonomous Snake game by comparing it against a standard shortest-path-first strategy. By implementing a Branch and Bound framework where an A* generated path is validated for safety using a DFS tail-reachability check, we have shown that strategic foresight provides a definitive performance advantage.

The experimental results clearly indicate that the Optimized algorithm dramatically outperforms the Baseline, achieving approximately double the average score and survival time. This confirms our central hypothesis: in a dynamic environment where the agent's own body becomes an obstacle, prioritizing path safety is more critical for long-term success than simply minimizing path length to an immediate goal. The modest increase in computation time required for the safety check is a worthwhile trade-off for the substantial gains in performance and robustness.

While this study confirms the validity of our approach, there are limitations to acknowledge. The experiments were conducted on a fixed grid size, and the safety check was a binary condition of tail reachability. Future work could expand on this

research in several directions. First, the algorithm's performance could be tested across various grid sizes and against more complex environmental constraints. Second, more sophisticated safety heuristics could be developed, such as evaluating the total number of free cells accessible after a move, rather than just tail reachability. Finally, a valuable comparative analysis could be conducted to benchmark this optimized A* strategy against entirely different paradigms, such as the pre-determined Hamiltonian Cycle approach, to better understand the trade-offs between dynamic path-finding and fixed-path strategies.

Overall, this research successfully demonstrates that by augmenting a classic path-finding algorithm like A* with a strategic safety-driven framework, an agent's performance can be significantly improved, highlighting a key principle in the design of intelligent agents for dynamic systems.

VII. Appendix

- YouTube video explaining the paper: https://youtu.be/ubDm1z3rmuI
- GitHub repository for the project: https://github.com/brii26/smart_snake

References

[1] Soetandio, A. A., & Lim, A. (2024). Penerapan algoritma A* (A-Star) untuk mencari jalur terpendek dalam kecerdasan buatan (studi kasus: game snake). Jurnal Paradigma Informatika, 16(1), 55-61. https://doi.org/10.31294/paradigma.v16i1.19662 [accessed 24 June 2025]

[2] Chen, Y. F., Lan, T. W., & Hsieh, C. H. (2011). A Hamiltonian-Cycle-Based Solution for the Snake Game. 2011 First National Conference on Web-based Education. https://doi.org/10.1109/NCWE.2011.23 [accessed 24 June 2025]

[3] Wang, W., & Li, W. (2020). An improved A-star algorithm for the snake game. Proceedings of the 2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE), 1324-1328. https://doi.org/10.1109/ICMCCE51767.2020.00253 [accessed 24 June 2025]

[4] Pal, A., Bhattacharya, S., & Das, A. K. (2018). An approach to solve the snake game by using greedy algorithm. International Journal of Computer Sciences and Engineering, 6(5), 701-705. https://www.ijcse.net/docs/IJCSE18-06-05-111.pdf [accessed 24 June 2025]

[5] Kosiński, W. (2013). A Comparison of Pathfinding Algorithms for the Game of Snake. Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on, 425-430. https://doi.org/10.15439/2013F358 [accessed 24 June 2025]

Statement

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Sumedang, 24 June 2025

Brian Ricardo Tamin, 13523126