Block Blast Puzzle Solver Using the A* Search Algorithm

Penyelesaian Puzzle Block Blast Menggunakan Algoritma Pencarian A*

Aliya Husna Fayyaza - 13523062
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: aliyahfayyaza@gmail.com[1], 13523062@std.stei.itb.ac.id[2]

Abstract—*This paper presents an implementation of the A* search algorithm to solve the Block Blast puzzle. The game involves placing a fixed number of randomly shaped blocks into a two-dimensional board without overlapping or exceeding the board boundaries. The algorithm explores possible placements by evaluating board states based on the number of blocks placed and a heuristic estimation of usable space. The heuristic function computes the total area of empty regions large enough to fit the remaining blocks, ensuring admissibility by never overestimating the potential for block placement. Experimental results show that the A* algorithm is both effective and efficient in finding valid solutions or determining when no solution exists. This approach demonstrates the applicability of informed search algorithms in solving logic puzzles.*

Keywords—*block puzzle, heuristic search, A* algorithm, game solver, pathfinding.*

I. Introduction

There are many algorithms that can be used to solve logical puzzle games, ranging from brute force approaches to more advanced informed search techniques such as Greedy Best-First Search, Uniform Cost Search (UCS), and the A* (A Star) algorithm. Among these, A* is widely regarded as one of the most effective and versatile algorithms due to its ability to combine the advantages of UCS and Greedy Search. By using a heuristic to estimate the remaining cost, A* can efficiently guide the search process toward a valid solution while minimizing unnecessary explorations.

One puzzle game that can benefit from this approach is Block Blast, a tile-based puzzle game where players are given a limited set of block pieces and must place them strategically on a board to maximize coverage and avoid running out of space.

This paper presents a solution to the Block Blast puzzle using the A* algorithm. By modelling the board state and available pieces as nodes and transitions in a search space, A* can intelligently explore possible placements, aiming to find a complete solution where all pieces fit within the board. The following sections describe the formulation of the problem, the design of the heuristic, and the implementation of the solver.

II. Basic Theory

A. Graph

A graph G is formally defined as an ordered pair $G = (V,E)$ where V is a non-empty set of vertices and E is a non-empty set of edges connecting pairs of vertices. For a structure to qualify as a graph, it must contain at least one vertex and one edge. The number of edges incident to a vertex is referred to as the degree of that vertex.

Graphs can be classified based on their structural properties. A simple graph is one that contains no loops (edges connecting a vertex to itself) and no multiple edges between the same pair of vertices. In contrast, an unsimple graph allows for loops and/or multiple edges. Unsimple graphs are further categorized into two types: multigraphs, which permit multiple edges between the same pair of vertices, and pseudographs, which include both loops and multiple edges. [1]

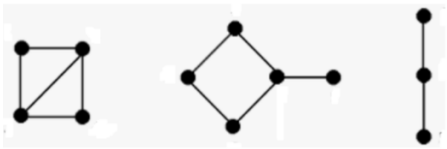

Fig 1. Simple graph examples (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024 - 2025/20-Graf-Bagian1-2024.pdf)

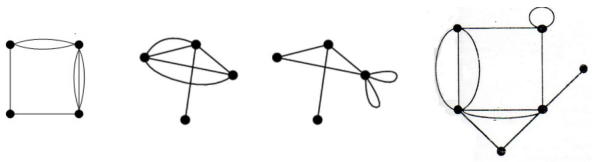

Fig 2. Unsimple graph examples (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024 - 2025/20-Graf-Bagian1-2024.pdf)

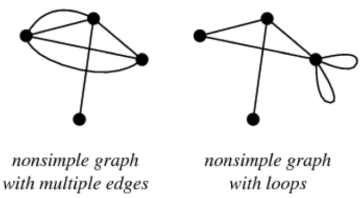

Fig 3. Unsimple graph categories
(source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024 - 2025/20-Graf-Bagian1-2024.pdf)

Another way to classify graphs is based on the directionality of their edges. An undirected graph has edges that do not point in any specific direction, meaning the connection between two vertices is bidirectional. On the other hand, a directed graph or digraph uses arrows to indicate the direction of each edge, signifying a one-way relationship between vertices.[1]

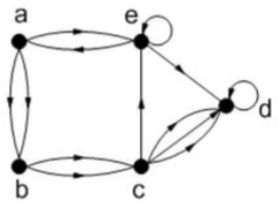

Fig 4. Directed graph example

(source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024 - 2025/20-Graf-Bagian1-2024.pdf)

Graphs are widely used in various fields due to their ability to model relationships and connections. Applications include electrical circuit design, chemical compound structures, routing and scheduling, medical imaging, transportation networks, social media platforms, and computer network topologies.[6]

B. Tree

A tree is a connected, undirected graph with no cycles. Formally, a graph $G = (V,E)$ with n vertices is a tree if it satisfies any of the following equivalent conditions:

1. G is connected and acyclic.
2. Any two vertices are connected by exactly one simple path.
3. G has $n-1$ edges and is connected.
4. G has $n-1$ edges and no cycles.
5. Adding any edge creates exactly one cycle.
6. All edges are bridges.

Furthermore, a forest is a collection of disjoint trees, or an acyclic graph that is not necessarily connected.[2]

In computer science, trees are used to represent hierarchical data. Their acyclic and connected nature makes them ideal for structures like game development, databases, and machine learning.[7] Trees are also used as the basis for many data structures such as binary trees, heaps, and syntax trees.

C. Uninformed and Informed Search in Pathfinding

Search strategies are generally divided into two categories: uninformed or blind search and informed or heuristic-based search. Uninformed search algorithms do not utilize any information about the proximity of a state to the goal. They explore the search space blindly, typically using simple strategies such as Breadth-First Search (BFS), Depth-First Search (DFS), or Uniform Cost Search (UCS).

In contrast, informed search algorithms incorporate additional knowledge in the form of heuristics to estimate the cost of reaching the goal from a given node. For example, there are the Greedy Best-First Search algorithm where it only consider the heuristic function to determine its expansion and the A* Algorithm where it also take the actual cost into account. This allows Greedy Best-First Search and A* to prioritize more promising nodes and significantly reduce the number of states explored during the search process[3] .Yet, because of its heuristic-only consideration, the Greedy Best-First Search does not guaranteed to find the shortest path[5].

C. A* Algorithm

The A* (A star) algorithm is one of the most widely used informed search strategies in artificial intelligence and algorithm design. It is valued for its completeness and optimality, as long as the heuristic function employed is admissible—meaning it never overestimates the true cost to reach the goal. A* operates by evaluating each node using the cost function:

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

where $g(n)$ represents the actual cost from the start node to the current node n, and $h(n)$ is a heuristic estimate of the cost from node n to the goal. By combining both real and estimated costs, A* is able to strike a balance between depth-first exploitation of known good paths and breadth-first exploration of new possibilities.

The algorithm proceeds by always expanding the node with the lowest $f(n)$ value from a priority queue, ensuring that paths with the most promising cost estimates are explored first. The exploration algorithm itself uses the Best-First Search algorithm. This strategy guarantees that the first solution found is the optimal one, as long as the heuristic is admissible and consistent.[5]

A* possesses the following key properties:

- Completeness: A* is complete as long as the branching factor is finite and all edge costs are greater than zero. It will eventually find a solution if one exists, unless there are infinitely many nodes with cost $f(n) \leq f(G)$, where G is the goal node.
- Time Complexity: In the worst case, time complexity is exponential, specifically $O(bm)$, where b is the branching factor and m is the depth of the shallowest goal.

- Space Complexity: A* maintains all generated nodes in memory to guarantee optimality, which leads to a space complexity of $O(bm)$ as well.
- Optimality: A* is optimal provided that the heuristic function is admissible. It always finds the least-cost path to the goal.[3]

Due to these characteristics, A* is extensively applied in various domains such as pathfinding, puzzle solving, and route planning. Its performance, however, is heavily dependent on the heuristic function used.

Several heuristic functions are commonly used in search algorithms, each suited to different types of problems depending on the structure of the state space. One widely used heuristic is the Euclidean distance, which calculates the straight-line distance between two points. This is particularly effective in continuous, geometric spaces where diagonal movement is allowed and cost is proportional to physical distance.

Another popular heuristic is the Manhattan distance, which sums the absolute differences of the horizontal coordinate and the vertical coordinate. It is commonly used in grid-based environments where movement is restricted to horizontal and vertical directions, such as in mazes or tile-based puzzles. The Chebyshev distance is another variant, defined as the maximum difference between the coordinates of two points. This heuristic is useful when movement in all eight directions is allowed, and each step has equal cost.[4]

Choosing the right heuristic function depends on the nature of the problem and the allowed movements in the state space. A well-designed heuristic helps reduce the search space while preserving admissibility and optimality.

D. Admissible Heuristic

An admissible heuristic is a heuristic function that never overestimates the actual minimum cost from any node to the goal. Formally, for every node *n*, it must satisfy the condition:

$$h(n) \leq h^*(n)$$

where *h*(n)* is the true cost of the shortest path from node *n* to the goal. This characteristic makes an admissible heuristic optimistic, as it always estimates the remaining cost as equal to or less than the actual cost. The use of an admissible heuristic is critical for ensuring that the A* algorithm finds the optimal path. According to the admissibility theorem, if *h(n)* is admissible, then A* with tree-search is guaranteed to return an optimal solution, assuming all step costs are non-negative.

A classic example of an admissible heuristic is the straight-line distance (SLD) used in route-finding problems. SLD never overestimates the actual travel distance on a road map, making it a reliable heuristic for geographical search problems.[3]

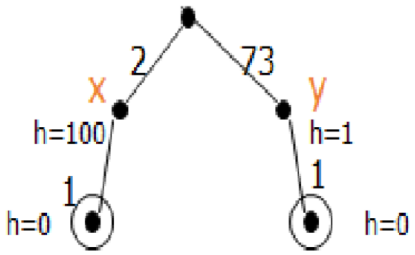

Fig 5. Inadmissible heuristic function example (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf)

If a heuristic is not admissible, A* may fail to find the optimal path. For example, if a node *Y* has *g(Y)+h(Y)=74,* while another node *X* in the optimal path has *g(X)+h(X)=102* due to an overestimated heuristic *h(X)*, the algorithm will wrongly prioritize expanding node *Y*, and the optimal path through *X* may be ignored or delayed. Therefore, selecting a good admissible heuristic is crucial in practice. It not only guarantees correctness but also significantly reduces the search space, making A* both sound and efficient.

E. Block Blast Game

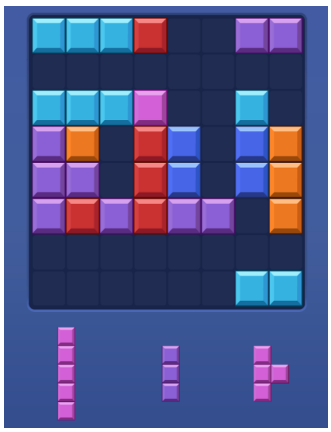

Fig 6. Block Blast game interface (source: https://blockblastonline.com)

Block Blast is a tile-based puzzle game played on a fixed 8×8 square grid. The player is presented with three blocks of random shapes, which must be placed onto the board. The objective is to fill entire rows or columns to clear them and score points. Once placed, blocks cannot be moved or rotated, making every decision permanent and strategic because the order of the placement of the blocks also matters.

The game ends when no combination of the placement order of the three available blocks can fit into the remaining empty spaces on the board. This constraint encourages players to think several moves ahead and manage space efficiently.

This game can be modelled as a search problem, where:

- The current configuration of the board represents the state.

- Placing a block at a legal position is considered an action.
- The goal state is any configuration where all three blocks can be successfully placed.

By framing the game in this way, Block Blast becomes a suitable domain for applying search algorithms such as A*, particularly when evaluating different strategies for block placement and estimating usable space through heuristic functions.

III. Heuristic Function Used

In the A* search algorithm, the heuristic function plays a critical role in estimating the desirability of a given state in relation to reaching the goal. In this implementation, the heuristic is designed to assess how much usable space remains on the board for placing the remaining blocks. The heuristic works by identifying all contiguous regions of empty cells on the board, summing the sizes of those regions that are at least large enough to fit the smallest remaining block, and then negating this value so that states with more usable space receive lower *f(n)* scores and are prioritized by A*.

Despite the unconventional approach of negating a space-based measurement, this heuristic remains admissible because it never overestimates the true cost to reach the goal. The key insight is that the chosen heuristic does not actually estimate cost, instead, it provides the quality of solution represented by the node. The heuristic makes no assumptions about whether that space will be used successfully, nor does it guarantee that the available space can accommodate all remaining blocks in a valid configuration. It simply counts empty regions without considering placement constraints, overlapping requirements, or the complex spatial relationships between different blocks (the row or column clearing constraint). Since the heuristic only provides an optimistic upper bound on usable space, it never suggests that a goal state is closer than it actually is. Even when negated, the heuristic maintains its conservative nature by avoiding any overestimation of progress toward the goal. The negative values merely serve as a prioritization mechanism, ensuring that more spacious states are explored first, while the underlying measurement remains a safe underestimate of the actual solution complexity. This preservation of admissibility ensures that A* will find an optimal solution if one exists, while benefiting from informed search efficiency.

IV. Implementation

The implementation of the Block Blast solver using the A* algorithm is structured into several modular components that handle input parsing, state representation, heuristic evaluation, search execution, and output visualization. Python is chosen due to its concise syntax and powerful support for data manipulation, which is especially beneficial for grid-based puzzle problems like Block Blast.

A. Required Dependencies

In this implementation, several Python libraries are used to support core functionalities of the solver:

1. numpy:
   Used extensively for numerical and matrix operations. In this implementation, numpy is responsible for representing the game board and blocks as 2D arrays, enabling efficient manipulation of grid-based data.
2. heapq:
   Provides a min-heap priority queue structure, which is essential for the A* algorithm. It ensures that the next most promising state (with the lowest f*(n)* = *g* + *h*) is always selected for expansion.
3. typing: Used for type annotations such as List, Tuple, and Optional, which improve code readability, maintainability, and safety by clarifying the expected input and output types of functions.

These libraries work together to manage grid representation, algorithmic state control, and type safety, forming the backbone of the puzzle-solving logic.

B. Problem Representation

The Block Blast problem is modeled using two input text files: map.txt representing the current game state and blocks.txt containing the three blocks to be placed. The map file contains a rectangular grid where cells are either empty (.) or occupied (*), with asterisks internally converted to hashes (#) for standardization. Blocks are defined in blocks.txt using unique uppercase characters (A, B, C), where the parser identifies connected components and constructs minimal bounding box arrays for each block using numpy.

Puzzle states are represented by the State class, which stores the current board configuration as a 2D numpy array, a list of integers (remaining_indices) indicating which blocks are yet to be placed, the cost so far g, and a heuristic value h that combines reachable empty cells. The A* algorithm uses the total cost of *f(n)* to prioritize states in the search queue, with the overridden __lt__ operator ensuring that states with lower total costs are explored first, guiding the search toward optimal solutions that efficiently place all blocks while maximizing line clears.

C. Heuristic Function Implementation

The heuristic function is defined to estimate how promising a given board state is in terms of future block placements. The key idea is to avoid getting stuck with unreachable or fragmented board regions that cannot fit any remaining blocks.

The reachable_empty_cells function calculates the number of empty cells that are part of a contiguous region large enough to fit at least the smallest remaining block. It does so by performing a Depth-First Search (DFS) over the board, starting from each unvisited empty cell. If the connected region meets the size requirement, its size is added to the heuristic count. This encourages the algorithm to favour board states that remain open and accessible for the remaining pieces.

```
def reachable_empty_cells(board: np.ndarray, blocks: List[np.ndarray]) -> int:
    if not blocks:
        return 0

    min_size = min(np.count_nonzero(block != '.') for block in blocks)
    visited = np.zeros(board.shape, dtype=bool)
    h, w = board.shape
    count = 0

    def dfs(x, y):
        if x < 0 or x >= h or y < 0 or y >= w:
            return 0
        if visited[x, y] or board[x, y] != '.':
            return 0
        visited[x, y] = True
        size = 1
        for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
            size += dfs(x+dx, y+dy)
        return size

    for i in range(h):
        for j in range(w):
            if board[i,j] == '.' and not visited[i,j]:
                area = dfs(i,j)
                if area >= min_size:
                    count += area
    return count
```

Fig. 7 Heuristic function implementation
(source: author's source code)

This heuristic is both simple and effective, serving as a guiding estimate without being overly optimistic. It avoids overestimating the board's flexibility, which could otherwise lead the search astray.

D. A Search Algorithm*

The core of the solver lies in the astar_solver function, which implements the A* search algorithm. It begins with the initial board state and pushes it into a priority queue. At each iteration, the state with the lowest *f(n)* value is expanded. If all blocks are placed, the current board is returned as the solution.

Each state expansion involves trying to place any remaining block at all valid positions on the board, taking into account that different order of the placement of the blocks could generate different results. The validity of a placement is checked using the can_place function, which verifies that every non-empty cell in the block corresponds to an empty cell on the board and does not exceed boundaries.

If a block can be placed, the place_block function creates a new board configuration by copying the current board, overlaying the block at the specified position, and then clearing any full rows or columns. These new configurations are wrapped in new State objects and pushed into the heap for further exploration. The algorithm continues until either a valid full placement is found or the search space is exhausted.

Each state expansion involves trying to place any unplaced block at all valid positions on the board. The validity of a placement is checked using the can_place function:

```
def can_place(board: np.ndarray, block: np.ndarray, x: int, y: int) -> bool:
    h, w = block.shape
    if x + h > board.shape[0] or y + w > board.shape[1]:
        return False
    for i in range(h):
        for j in range(w):
            if block[i, j] != '.' and board[x + i, y + j] != '.':
                return False
    return True
```

Fig 8. can_place function
(source: author's source code)

This function checks that every non-empty cell in the block corresponds to an empty cell on the board and does not exceed the board's boundaries. If a block can be placed, the place_block function creates a new board configuration by copying the current board, overlaying the block at the specified position, and then automatically clearing any complete rows or columns. The line-clearing process checks for rows and columns that contain no empty cells (all filled with blocks) and removes them by setting all cells in those lines back to empty (.), continues iteratively until no more full lines can be found. This line-clearing feature is essential to the Block Blast gameplay, as it creates additional space for subsequent block placements and is often necessary to successfully place all blocks on the board.

```
def place_block(board: np.ndarray, block: np.ndarray, x: int, y: int) -> np.ndarray:
    new_board = board.copy()
    h, w = block.shape

    for i in range(h):
        for j in range(w):
            if block[i, j] != '.':
                new_board[x + i, y + j] = block[i, j]

    cleared_board = clear_full_lines(new_board)

    return cleared_board
```

Fig 9. place_block function
(source: author's source code)

These new configurations are wrapped in new State objects and pushed into the heap for further exploration. The algorithm continues until either a valid full placement is found or the search space is exhausted.

The main search process is handled by the astar_solver function, which implements the A* algorithm using a priority queue. The steps are as follows:

1. Initialization:

```
start = State(board, [False] * len(blocks), 0, blocks)
    heap = [start]
```

Fig 9. Initialization step of the A function*
(source: author's source code)

- The initial state is created using the starting board, a list of False flags indicating that no blocks are placed, and an initial cost g = 0.
- This state is pushed into a priority queue (heap) using heapq.

2. Search Loop
   - While the priority queue is not empty, the state with the lowest f(n) = g + h is popped from the heap.
   - If all blocks are placed (all(current.placed)), the current board configuration is returned as the solution.

3. State Tracking:

```
state_id = (tuple(map(tuple, current.board)), tuple(current.placed))
        if state_id in visited:
            continue
        visited.add(state_id)
```

Fig 10. State tracking step of the A function*
(source: author's source code)

- To avoid redundant computation, each state is uniquely identified by its board layout and placement status.
- If a state has been visited before, it is skipped.

4. Expansion

```
for x in range(board.shape[0]):
                for y in range(board.shape[1]):
                    if can_place(current.board, block, x, y):
                        new_board = place_block(current.board, block, x, y)
                        new_remaining = [idx for idx in current.remaining_indices if idx != block_idx]

                        new_state = State(new_board, new_remaining, current.g + 1, blocks)
                        heapq.heappush(heap, new_state)
```

Fig 10. Expansion step of the A function*
(source: author's source code)

- For each unplaced block, the algorithm tries every possible position (x, y) on the board.
- If the block can be placed (can_place returns True), a new board is generated (place_block) and a new state is created.
- This new state has $g + 1$ as its new path cost and recalculates the heuristic h. It is then pushed into the priority queue for exploration.

5. Failure Case

- If the heap is exhausted but still no complete configuration has been found, the function returns None, indicating that no valid placement exists for all blocks.

V. Testing and Analysis

Author tested few inputs of map.txt and blocks.txt:

1. Test Case 1

Input:

map.txt

```
1  ***.....
2  ***..*..
3  ***..*..
4  ***..*..
5  ****....
6  ****..**
7  ***.....
8  ........
```

blocks.txt

```
1  AAAA
2  BB
3  B
4  B
5  CCC
```

Fig 11. Test case 1 inputs
(source: author's source code)

Ouput:

```
***.....
***BB*..
***B.*..
***B.*..
****....
****..**
***.CCC.
..AAAA..
```

Fig 12. Test case 1 output
(source: author's source code)

2. Test Case 2

Input:

map.txt

```
1  ........
2  ...*....
3  ...*....
4  .***....
5  *..***..
6  ......*.
7  ......*.
8  ......*.
```

blocks.txt

```
1  AA
2  A
3  BBB
4  B
5  C
6  CCC
```

Fig 13. Test case 2 inputs
(source: author's source code)

Output:

```
........
...*....
...*....
.***....
*..***..
BBB...*.
.BAAC.*.
..ACCC*.
```

Fig 14. Test case 2 output
(source: author's source code)

3. Test Case 3

Input:

map.txt

```
1  ***..***
2  *..*.*..
3  *..*...*
4  .***...*
5  *..***.*
6  .**...*.
7  .***..*.
8  .*....*.
```

blocks.txt

```
1  AAA
2  A
3  BBB
4  B
5  C
6  CCC
```

Fig 15. Test case 3 inputs
(source: author's source code)

Output:

```
No solutions found
```

Fig 16. Test case 3 output
(source: author's source code)

4. Test Case 4

Input:

map.txt

```
1  *******.
2  *******.
3  *******.
4  *******.
5  *******.
6  *******.
7  *******.
8  ........
```

blocks.txt

```
1  B
2  B
3  B
4  B
5  B
6  B
7  AAAAAA
8  C
9  CCC
```

Fig 17. Test case 4 inputs
(source: author's source code)

Output:

Fig 18. Test case 4 output
(source: author's source code)

From the four test cases evaluated, the program successfully found solutions for different combinations of maps and blocks, including the combinations where the solution is found only if the block inserted is clearing a row or column first to make space for the other blocks. It also correctly identified cases where no valid solution exists.

This shows that the A algorithm works well and the solution search has been adapted to match the unique characteristics of Block Blast, including the line-clearing mechanic that helps create space for the next blocks.

V. Conclusion

In this project, author have successfully implemented a Block Blast puzzle solver using the A* search algorithm. The problem was formulated as a search over board configurations, where each state represents a specific arrangement of placed blocks. By combining the cost of actions (*g(n)*) and a custom heuristic function (*h(n)*) that estimates the remaining empty usable spaces, the A* algorithm is able to effectively explore the most promising paths in search of a complete solution.

Through several test cases, including unsolvable scenarios, it is demonstrated that this approach is both robust and adaptable. However, due to the lack of rotation and mirroring support, the algorithm is limited to straightforward block orientations. Future improvements may include enhancements such the search of a better heuristic function.

Overall, this implementation shows that A* is a viable and effective method for solving spatial logic puzzles like Block Blast, where optimal placement and constraint satisfaction are essential.

VI. Appendix

Github: https://github.com/aliyahusnaf/StimaPaper-13523062.git
Bonus video: https://youtu.be/Ej478xvBR7k

VII. Acknowledgment

Author would like to express sincere gratitude to God Almighty for His blessings and guidance that made the completion of this paper possible. The author also extends appreciation to all individuals who supports author through out the working of this paper.

Special thanks are given to Mr. Rinaldi, Mr. Monterico Adrian, S.T., M.T., and Dr. Nur Ulfa Maulidevi, S.T., M.Sc., as lecturers of the Algorithm Strategy course, for their support, encouragement, and clear explanation of the concepts, especially regarding the A* algorithm, which is the core of this paper. Lastly, the author acknowledges the various sources and references that have provided valuable information throughout the writing of this paper.

References

[1] MUNIR, RINALDI. 2024. "Graf (Bag. 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. Accessed 22 June 2025, 8:30 AM.

[2] MUNIR, RINALDI. 2024. "Pohon (Bag. 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf. Accessed 22 June 2025, 9:00 AM.

[3] MUNIR, RINALDI. 2024. "Route Planning (Bagian 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf. Accessed 22 June 2025, 11:30 AM.

[4] PATEL, AMIT. 2025. "Heuristic from Amit's Thoughts on Pathfinding". https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html. Accessed 23 June 2025, 8:30 AM.

[5] PATEL, AMIT. 2025. "Introduction to A* from Amit's Thoughts on Pathfinding". https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. Accessed 23 June 2025, 3:00 PM.

[6] Dutta, Jita. 2024. "Application of Graph Theory in real Life". https://ijnrd.org/papers/IJNRD2410012.pdf. Accessed 23 June 2025. 10.00 PM.

[7] Balding. 2024. "Real World Examples of Tree Structures". https://www.baeldung.com/cs/tree-examples/ Accessed 23 June 2025. 10.10 PM.

Personal Statement

I hereby declare that the paper I have written is my own work, not an adaptation or translation of someone else's paper, and not plagiarism.

Jakarta, June 22 2025

Aliya Husna Fayyaza/13523062