Implementation of Backtracking Algorithm to Solve the Sootopolis City Gym Puzzle in Pokémon Emerald

Jethro Jens Norbert Simatupang - 13523081
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: jethrojsimatupang@gmail.com , 13523081@std.stei.itb.ac.id

***Abstract*—The Sootopolis City Gym Puzzle in *Pokémon Emerald* challenges players to step on all the ice tiles on a stage without stepping on any tile more than once, in order to unlock the path to the Gym Leader. This problem can be modeled as a pathfinding task on a graph and solved using a backtracking algorithm. This paper implements the algorithm recursively by exploring all possible paths and pruning when rules are violated. The implementation results show that backtracking is effective in finding the solutions, although complexity increases with stage size. This approach proves to be relevant for solving logic-based challenges in the context of video games.**

Keywords—Solution Space Exploration, Backtracking Algorithm, Constraint Satisfaction, Sootopolis City Gym Puzzle

I. Introduction

Video games are electronic games that involve interaction between users and hardware through input devices to produce visual and audio output. Modern video games often not only offer entertainment, but also also hone the logical and strategic thinking skills of their players through various challenges. One such challenge commonly found in video games is a puzzle that players must solve before progressing further. One video game that features a variety of engaging puzzles is *Pokémon Emerald*, a popular RPG released by Nintendo in 2004.

In *Pokémon Emerald*, players are faced with various logic-based challenges spread across various Gyms. One of the more interesting challenges is the Sootopolis City Gym Puzzle, in which players must traverse a series of ice tiles, in order to make their way to the Gym leader. While the puzzle may appear simple at first, it becomes increasingly challenging as the size and complexity of the tile layout grow. This challenge resembles classic problems in computer science, particularly in the domains of pathfinding and solution space exploration. The problem can be represented as a graph, with each tile as a node and movements between tiles as edges. This modeling opens up opportunities to apply systematic search algorithms to find an optimal solution.

In solving solution finding problems, the backtracking algorithm is a suitable choice, especially for problems with multiple constraints that require trial and error, such as the Sootopolis City Gym Puzzle. The backtracking algorithm builds solutions by recursively exploring all possible paths, with the ability to backtrack when necessary. When a path being constructed proves not to lead to a valid solution or violates the bounding function, the algorithm backtracks and tries an alternative path, known as pruning. This ability to prune invalid paths makes the backtracking algorithm powerful and effective for finding solutions for solution finding problems with bounding functions and large solution spaces.

This paper implements the backtracking algorithm to solve the Sootopolis City Gym Puzzle in the game *Pokémon Emerald*. The main focus lies in how the problem is modeled, how the algorithm is applied efficiently, and how it performs across various puzzle configurations. This paper aims to demonstrate the potential of the backtracking approach in solving logic-based challenges within the context of video games, as well as to provide insight into the practical application of the backtracking algorithm in real-world problem solving.

II. Theoritical Basis

A. Solution Space Exploration

In computer science, many problems can be represented as a process of searching for a solution within a state space. This space contains a set of all possible state that can be reached from an initial state through a series of actions or steps. Each possible state is typically represented as a node in a tree, and the steps taken to reach those states are represented as edges. Solution space exploration is the systematic process of finding one or more solutions that meet specific criteria or constraints within that space.

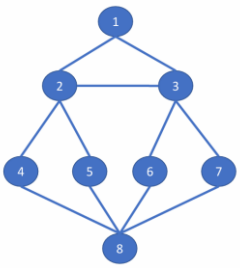


Figure 1. Solution space exploration illustration
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf

One fundamental approach to solving problems in a solution space is exhaustive search. This method works by systematically trying all possible solutions. While exhaustive search guarantees that a solution will be found if one exists, it is inefficient for large solution spaces due to its exponential time and space complexity.

Several commonly used search algorithms in solution space exploration include Depth-First Search (DFS), Breadth-First Search (BFS), and Brute Force Search. These algorithms work by exhaustively evaluating all possible steps, but they differ in their traversal order and efficiency. For example, DFS explores a path deeply until it reaches the maximum depth before backtracking, while BFS explores all paths at the same level before moving to deeper levels. Brute force, on the other hand, evaluates all possibilities without employing any strategy to reduce the search space.

B. Backtracking Algorithm

The backtracking algorithm is one of the systematic search techniques suitable for solving problems that have many possible solutions, but are constrained by specific rules. Backtracking is an improvement of exhaustive search. Unlike exhaustive search, which explores all possible solutions, backtracking only explores choices that lead toward a solution. Choices that do not lead to a solution are discarded by pruning nodes that do not contribute to a valid outcome.

To solve a problem using the backtracking algorithm, the problem needs to be modeled by considering several key properties of the backtracking approach:

1. Solution Space
   All possible solutions to the problem.
2. Node (State)
   A node or state represents a partial solution's current condition. Each node stores the current status during the search process.
3. Edge
   An edge is a step or action that moves the algorithm from one node to another. For example, by selecting a value to add to the partial solution. In graph form, it represents an edge connecting state A to state B.
4. Root Node and Goal Node
   The root node is the starting point of the search, typically representing an empty or partially filled solution. The goal node is a leaf node that represents a complete and valid solution.
5. Problom Solution
   The solution is expressed as an n-tuple vector: $X = (x_1, x_2, ..., x_n)$, where $x_i \in S_i$. Generally, $S_1 = S_2 = ... = S_n$.
6. Generator Function
   The generator function is expressed as T(). $T(x_1, x_2, ..., x_{k-1})$ generates a value for x_k, which is a component of the solution vector.
7. Bounding Function
   Expressed as a predicate $B(x_1, x_2, ..., x_k)$ that returns true or false. B returns true if $(x_1, x_2, ..., x_k)$ leads to a solution, meaning it does not violate any constraints. If true, the generation of a value for x_{k+1} continues; if false, then $(x_1, x_2, ..., x_k)$ is discarded.

The principle of solution search using the backtracking algorithm is carried out by generating state nodes that form a path from the root to the leaves in a search tree. The generation of these nodes follows a depth-first order (DFS). Nodes that have been generated are called live nodes, while the live node currently being expanded is referred to as the E-node (expand node). Each time an E-node is expanded, the path it forms becomes longer. However, if the path does not lead to a solution, the E-node is "terminated" and becomes a dead node by applying a bounding function. When a node becomes dead, its child nodes are implicitly pruned as well.

If the path being formed ends in a dead node, the search process will backtrack to a node at the previous level and attempt to generate another child node. This new node then becomes the new E-node. The search process stops once a goal node has been found.

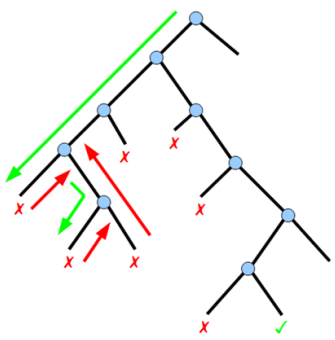


Figure 2. Backtracking illustration
Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/15-Algoritma-backtracking-(2025)-Bagian1.pdf

C. Pokémon Emerald

Pokémon Emerald is one of the games in the Pokémon series developed by Game Freak and published by Nintendo for the Game Boy Advance console. This game belongs to the role-playing game (RPG) genre, where players take on the role of a Pokémon Trainer exploring the Hoenn region to catch Pokémon, battle other trainers, and overcome various challenges. The game not only relies on strategy in battles but also incorporates puzzle and problem-solving elements that require logic and planning.

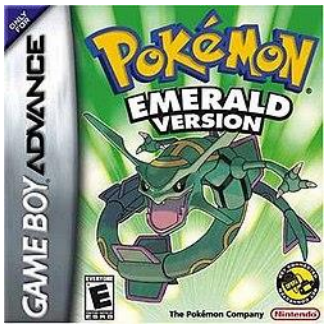


Figure 3. North American box art for *Pokémon Emerald*, depicting the Legendary Pokémon Rayquaza
Source: https://en.wikipedia.org/wiki/Pok%C3%A9mon_Emerald

One important aspect of *Pokémon Emerald* is the challenge posed by the eight Gym Leaders scattered throughout the Hoenn region. Before being allowed to challenge a Gym Leader, players are required to complete a puzzle or obstacle, which is often designed as an integral part of the Gym itself and serves as the path leading to the Gym Leader.

D. *Sootopolis City Gym Puzzle*

One of the most intriguing puzzles in *Pokémon Emerald* is the Sootopolis City Gym Puzzle, which challenges players to walk across all of the ice tiles present without stepping on the same tile more than once. Each ice tile cracks the first time it is stepped on, and if stepped on again, it shatters, causing the player to fall to the lower floor and start over. Additionally, the puzzle contains rocks that cannot be crossed, acting as obstacles and adding difficulty in determining the correct path.

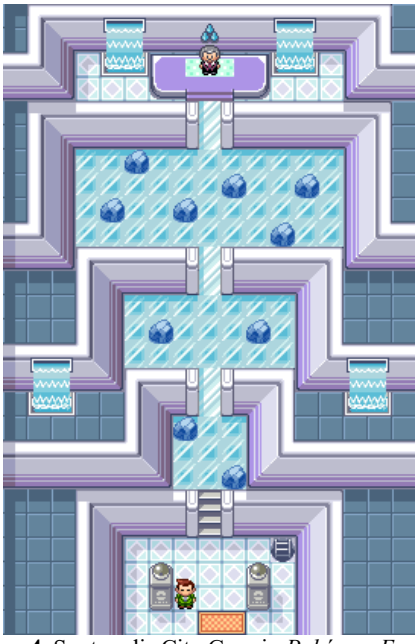


Figure 4. Sootopolis City Gym in *Pokémon Emerald*
Source: https://pokemon.fandom.com/wiki/Sootopolis_City_Gym

This puzzle consists of three separate stages at three different floors, each connected by a staircase. At the beginning, the staircase to the next floor is closed and players begin on the first tile in front of the previous staircase. The objective of each stage is to step on all the tiles in the correct sequence, ending with the final step on the tile directly in front of the staircase to the next floor. Only by fulfilling this condition will the stairs to the next stage unlock and allow the player to climb up the ladder without the need to step on additional tiles. This puzzle exhibits the characteristics of a solution space exploration problem, making it well-suited for analysis and resolution using algorithmic approaches such as backtracking to find a path that satisfies all given constraints.

III. Implementation

A. *Problem Modeling*

The Sootopolis City Gym Puzzle can be modeled as a two-dimensional board in the form of a matrix, where each cell represents a specific condition:

- 'S' marks the player's starting point, which is the first tile in front of the staircase,
- 'G' marks the goal point, which is the tile directly beneath the staircase,
- '.' represents a tile that can be stepped on once,
- 'X' represents a rock that cannot be crossed.

This problem can be formulated as a pathfinding task from 'S' to 'G' under the following conditions:

1. All walkable tiles ('.' and 'G') must be visited exactly once,
2. The player must end the path at the 'G' position,
3. The player must not step on any tile more than once.

This problem falls into the category of exponential solution space exploration, making a brute-force approach highly inefficient. Therefore, a backtracking algorithm is used, along with a pruning strategy to avoid exploring invalid paths.

The definitions of the backtracking algorithm properties for this problem are as follows:

1. Solution Space
   The solution space consists of all possible paths where no tile is stepped on more than once.
2. Node (State)
   Each state is represented by a matrix coordinate, consistent with the puzzle's modeling as a matrix-based board.
3. Edge
   Edges in this problem model represent the steps taken, which can be in four directions: west, north, east, and south.
4. Root Node and Goal Node
   The root node is represented by the starting point of the search, marked by the position 'S', while the goal node is represented by the endpoint of the search, marked by the position 'G'.
5. Problem Solution
   The problem solution is represented as a vector of step sequences $X = (x_1, x_2, ..., x_n)$, where each x_i is a coordinate (r, c) of a tile visited. Each element in this vector represents a single step in the chosen path, and its values are taken from the domain S_i, which consists of valid neighboring cells that can be visited from the previous step.
6. Generator Function
   The generator function takes the last position from the partial solution vector and checks four possible directions of exploration (west, north, east, and south). Only cells that have not been visited, do not contain 'X', and are within the grid boundaries will be generated as candidates for the next step.
7. Bounding Function
   The bounding function is used to decide whether the partial solution is still valid and should be continued, or whether it should be pruned. This function returns false if the partial solution meets any of the following conditions:
   (a) The player steps on the 'G' tile before all required tiles have been visited,

(b) The player is trapped between 'X' tiles and cannot move before reaching the goal 'G',
(c) There is a '.' tile that becomes unreachable (e.g., surrounded by 'X' tiles, making it impossible for the player to step on it).

This problem also has a deterministic nature and doesn't involve random elements, meaning the solution path is entirely determined by the initial stage/board conditions and movement rules. This allows for systematic exploration of the solution space and consistent replication of results. Furthermore, because all information is available from the outset, no learning or adaptation process is required during the solution search, which makes search-based approaches like backtracking highly suitable for solving this type of puzzle.

B. Algorithm Implementation

To solve the Sootopolis City Gym Puzzle challenge, the backtracking algorithm is implemented as a recursive function that explores all possible paths from the starting position to the goal, ensuring that all walkable tiles are visited exactly once. The solution is constructed using recursion and pruning to avoid searching paths that do not meet the criteria. The main function, `solve_gym_puzzle`, is responsible for processing the puzzle matrix, determining the starting and goal positions, and initializing the solution search using the backtracking technique.

```
def solve_gym_puzzle(grid):
   rows = len(grid)
   cols = len(grid[0]) if rows > 0 else 0

   # Temukan posisi awal (S) dan hitung total titik yang
harus dikunjungi
   start_pos = None
   target_positions = set()
   for i in range(rows):
      for j in range(cols):
         if grid[i][j] == 'S':
            start_pos = (i, j)
         if grid[i][j] == '.' or grid[i][j] == 'G':
            target_positions.add((i, j))

   if not start_pos:
      return None, 0

   target_count = len(target_positions)

   # Arah: kiri, kanan, atas, bawah
   directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]

   best_path = None
   nodes_visited = 0

   def is_valid(x, y, visited):
      return 0 <= x < rows and 0 <= y < cols and grid[x][y]
!= 'X' and not visited[x][y]

   def can_reach_all_targets(x, y, visited,
remaining_targets):
      # Buat salinan visited untuk flood fill
      visited_copy = [row[:] for row in visited]
      queue = [(x, y)]
      visited_copy[x][y] = True
      found_targets = set()

      while queue:
         cx, cy = queue.pop(0)
         if (cx, cy) in remaining_targets:
            found_targets.add((cx, cy))
            if len(found_targets) == len(remaining_targets):
               return True
         for dx, dy in directions:
            nx, ny = cx + dx, cy + dy
            if is_valid(nx, ny, visited_copy):
               visited_copy[nx][ny] = True
               queue.append((nx, ny))

      return len(found_targets) == len(remaining_targets)

   def backtrack(x, y, visited, path, remaining_targets):
      nonlocal best_path, nodes_visited
      nodes_visited += 1

      current_char = grid[x][y]
      new_remaining = remaining_targets.copy()

      # Jika menginjak target, hapus dari remaining_targets
      if (x, y) in new_remaining:
         new_remaining.remove((x, y))

      # Basis: mencapai G dan semua target terpenuhi
      if current_char == 'G' and not new_remaining:
         if best_path is None or len(path) + 1 <
len(best_path):
            best_path = path + [(x, y)]
         return

      # Prune jika tidak semua target bisa dicapai dari
posisi ini
      if not can_reach_all_targets(x, y, visited,
new_remaining):
         return

      # Tandai sudah dikunjungi
      visited[x][y] = True

      # Rekursi: coba semua arah
      for dx, dy in directions:
         nx, ny = x + dx, y + dy
         if is_valid(nx, ny, visited):
            backtrack(nx, ny, [row[:] for row in visited],
path + [(x, y)], new_remaining)

      # Unmark visited untuk backtracking
      visited[x][y] = False

   # Mulai backtracking dari posisi awal
   initial_visited = [[False for _ in range(cols)] for _ in
range(rows)]
   backtrack(start_pos[0], start_pos[1], initial_visited,
[], target_positions)

   return best_path, nodes_visited

def print_solution(grid, path, nodes_visited):
   if not path:
      print("Tidak ada solusi ditemukan")
      print(f"Jumlah node yang dikunjungi: {nodes_visited}")
      return

   solution_grid = [list(row) for row in grid]  # Buat
salinan grid
   for step, (x, y) in enumerate(path[1:], 1):
      if solution_grid[x][y] not in ['S', 'G']:
         # Format step dengan dua digit, jika step > 99 maka
akan menunjukkan tiga digit
         solution_grid[x][y] = f"{step:02d}"

   for row in solution_grid:
      print(' '.join(f"{str(c):>2}" for c in row))
   print(f"Jumlah node yang dikunjungi: {nodes_visited}")
   print()

]
```

The implementation above begins by reading the game layout/board and identifying the starting position ('S') as well as all the target tiles that must be visited (stepped), i.e. all

regular tiles ('.') and the final goal ('G'). The algorithm then maps the starting position and collects all target points into a set called `target_positions`.

The helper function `is_valid` is crucial for navigating the puzzle. It ensures that a cell can be stepped on by checking two key conditions: first, that it is not a rock (represented by `grid[x][y] != 'X'`), and second, that it has not already been visited in the current path (`not visited[x][y]`). This strict validation helps to immediately rule out impossible moves and prevent redundant exploration, contributing to the overall efficiency of the backtracking process.

To avoid exploring paths that are clearly invalid from the start, the bounding function is applied strictly. One such boundinf function is that the player must not reach the 'G' tile before all targets are visited (the `backtrack` function only accepts a path to 'G' if `remaining_targets` is empty). This prevents invalid solutions as described in the bounding function (a).

Additionally, bounding function (b) and (c) are enforced through a flood fill mechanism implemented in the `can_reach_all_targets` function. This function verifies whether all remaining targets are still reachable from the current position. If not, the path is immediately terminated before proceeding to the next recursive step. This prevents scenarios in which the player becomes trapped in a closed-off area (e.g., surrounded by 'X') or when a '.' tile becomes unreachable due to a previous wrong move, as outlined in the bounding function (b) and (c). Such pruning is crucial for improving efficiency and avoiding exploration of branches that can never lead to a valid solution.

During the exploration process, the `backtrack` function attempts all movement directions (left, up, right, down) from the current position. Each step is executed recursively using a copy of the current visitation status and path. When a valid solution is found, i.e. reaching 'G' with all targets visited, that path is stored as the best candidate. After all possibilities have been explored, the solution is visualized using the `print_solution` function, which displays the stage along with the sequence of the player's steps in solving the puzzle. This approach demonstrates how strong and selective constraint functions can maintain the accuracy and efficiency of the backtracking algorithm.

C. Testing Result and Analysis

After the backtracking algorithm was successfully implemented, testing was conducted on three puzzle configurations corresponding to the three stages in the Sootopolis City Gym. Each stage presents an increasing level of difficulty, with stage structures varying from small to large in size. The following are the solutions produced by the algorithm for each puzzle:

```
# Puzzle 1
puzzle1 = [
   ['X', 'G', '.'],
   ['.', '.', '.'],
   ['.', 'S', 'X']
]
print("Solusi Puzzle 1:")
path1, nodes1 = solve_gym_puzzle(puzzle1)
print_solution(puzzle1, path1, nodes1)
```

```
Solusi Puzzle 1:
 X  G 05
02 03 04
01  S  X
Jumlah node yang dikunjungi: 14
```

Figure 5. Execution result of puzzle 1
Source: Author's document

```
# Puzzle 2
puzzle2 = [
   ['.', '.', '.', 'G', '.', '.', '.'],
   ['.', 'X', '.', '.', '.', 'X', '.'],
   ['.', '.', '.', 'S', '.', '.', '.']
]
print("Solusi Puzzle 2:")
path2, nodes2 = solve_gym_puzzle(puzzle2)
print_solution(puzzle2, path2, nodes2)
```

```
Solusi Puzzle 2:
05 06 07  G 17 16 15
04  X 08 09 10  X 14
03 02 01  S 11 12 13
Jumlah node yang dikunjungi: 219
```

Figure 6. Execution result of puzzle 2
Source: Author's document

```
# Puzzle 3
puzzle3 = [
   ['.', '.', 'X', '.', '.', 'G', '.', '.', '.', '.', '.'],
   ['.', '.', '.', '.', '.', '.', 'X', '.', '.', 'X', '.'],
   ['.', 'X', '.', '.', 'X', '.', '.', '.', '.', '.', '.'],
   ['.', '.', '.', '.', '.', 'S', '.', '.', 'X', '.', '.']
]
print("Solusi Puzzle 3:")
path3, nodes3 = solve_gym_puzzle(puzzle3)
print_solution(puzzle3, path3, nodes3)
```

```
Solusi Puzzle 3:
08 09  X 15 16  G 36 35 32 31 30
07 10 11 14 17 18  X 34 33  X 29
06  X 12 13  X 19 20 23 24 25 28
05 04 03 02 01  S 21 22  X 26 27
Jumlah node yang dikunjungi: 923
```

Figure 7. Execution result of puzzle 3
Source: Author's document

The testing results show that the backtracking approach, optimized with pruning, is capable of solving the puzzles across different level of difficulty, including the large-sized stage. All three stages of the puzzle were successfully completed in a relatively short execution time, showing the algorithm's correctness and its ability to traverse the search space efficiently under pruning constraints.

However, although the backtracking algorithm successfully solved all three puzzles in a relatively short time, the results indicate that the number of visited nodes (explored states) for more complex puzzles (such as Puzzle 3, with a 4 × 11 board) was quite high, reaching 923 nodes. This is due to the exponential nature of the backtracking algorithm, with a time complexity of $O(b^d)$, where b is the branching factor and d is the depth of the solution. This means as the stage size and solution depth increase, the number of states to explore grows rapidly, making the algorithm less scalable despite pruning optimizations.

To optimize this algorithm, several additional approaches may be applied. For example, prioritizing tiles with limited movement options (only 1–2 directions) early in the search can

help reduce the branching factor. Additionally, the stage could be divided into smaller zones to be solved individually before combining them into a complete solution (divide and conquer). Another potential enhancement is applying a two-way search, where exploration begins simultaneously from both the start and goal positions, which may significantly speed up the pathfinding process.

IV. Conclusion

Based on the testing results, it can be concluded that the backtracking algorithm successfuly solved the Sootopolis City Gym Puzzle in *Pokémon Emerald* by exploring all possible paths while respecting constraints and pruning invalid branches to focus only on viable solutions. Although it guarantees finding a valid solution, its exponential complexity becomes a challenge in larger puzzles, as seen in stage 3 with a significant increase in visited nodes.

To enhance performance, techniques such as heuristic-based exploration or divide-and-conquer strategies may be applied. Overall, this paper demonstrates the effectiveness of backtracking in solving grid-based puzzles in video games, although scalability remains a challenge for more complex cases.

V. Suggestion

The author's suggestion for future researchers who wish to continue this topic is to enhance the backtracking algorithm by applying specific heuristics, such as prioritizing tiles with limited movement options early in the search, to reduce the branching factor. Additionally, a hybrid approach that combines backtracking with divide-and-conquer algorithms, such as dividing the stage into smaller zones to be solved separately and then merging the solutions through two connecting tiles, could also be a promising strategy. An alternative improvement that involves a bidirectional search, where the exploration starts simultaneously from both the starting point and the goal, could also potentially accelerate the pathfinding process considerably. This backtracking-based solution search approach also has potential applications in other types of puzzles that involve a set of constraints and can be similarly modeled to find solutions from a set of decisions.

VI. Appendix

The GitHub repository for this paper can be accessed at https://github.com/JethroJNS/Stima_Makalah_13523081.git
and the explanatory video for this paper can be accessed at https://youtu.be/ZmyhBRjQqyk?si=j3BKzLu9mKDnmWhI

Acknowledgment

The author expresses heartfelt gratitude to our Father in Heaven, whose boundless grace, wisdom, and strength made the completion of this paper possible. It was through His divine guidance and provision that every step of this work was successfully accomplished. Deep appreciation is also extended to the esteemed lecturer, Dr. Ir. Rinaldi Munir, M.T., whose invaluable guidance, encouragement, and support greatly enhanced the quality of this research. The author further thanks their family for their unwavering love, prayers, and faith, which provided strength and encouragement throughout the making of this paper. Lastly, sincere gratitude goes to friends, whose fellowship and support enriched this meaningful experience.

References

[1] Munir, R., & Maulidevi, N. U. 2025. "Breadth/Depth First Search (BFS/DFS) (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf (accessed on 22nd June 2025).

[2] Munir, Rinaldi. 2025. "Algoritma Runut-balik (Backtracking) (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/15-Algoritma-backtracking-(2025)-Bagian1.pdf (accessed on 22nd June 2025).

[3] Munir, Rinaldi. 2025. "Algoritma Runut-balik (Backtracking) (Bagian 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/16-Algoritma-backtracking-(2025)-Bagian2.pdf (accessed on 22nd June 2025).

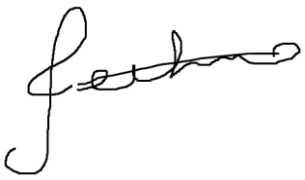
[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[5] Levitin, A. 2011. *Introduction to the Design and Analysis of Algorithms*, 3rd ed. Addison-Wesley.

STATEMENT

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

Bandung, 22nd June 2025



Jethro Jens Norbert Simatupang
13523081