Solving Rubik's Cube using Branch & Bound Algorithm

Ignatius David Partogi - 13518014
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): ignatiusdavidpartogi@gmail.com

***Abstract*—Rubik's Cube is a popular puzzle game where the player must solve a 3x3x3 cube by rotating the sides of the cube to change its state. Rubik's Cube can be useful to train the player's brain. This paper provides a way to solve a Rubik's Cube using Branch & Bound algorithm. It also explains how the algorithm works and the testing of the program using different test cases.**

Keywords—Rubik's Cube, Puzzle Game, Rotation, State, Train, Branch & Bound, Algorithm.

I. INTRODUCTION

Rubik's Cube is one of the most popular puzzle games in the world. The puzzle game is called Rubik's Cube because it was invented by Ernõ Rubik, a Hungarian sculptor and professor of architecture, in 1974. It was originally called the Magic Cube. The invention quickly rose in popularity and became the world's best-selling toy. Rubik's Cube became a fun and brain-stimulating puzzle game, but also a target for some enthusiasts looking to find algorithms to solve it. Since its popularity, variations of Rubik's Cube start to emerge, from 2x2x2 cube to non-cubical designs. Rubik's Cube was inspired by China's Luo Book.[1]

Rubik's Cube is a 3D combination puzzle game. It consists of a 3x3x3 cube configuration formed by 26 cubes (called cubelets). Rubik's Cube has approximately 43,252,003,274,489,856,000 (43 quintillion) possible states.

The amount of possible states a Rubik's Cube has means solving the cube requires a lot of computing power and RAM. To solve this problem, algorithms are implemented into the solving program to reduce both the computing power requirement and execution time. One of the algorithms that can be implemented into a Rubik's Cube solving program is Branch & Bound algorithm.

Figure 1. Rubik's Cube (source: https://ruwix.com)

II. THEORIES

This section discusses about the theories and explanations of some terms used in this paper. Terms used in this paper that will be explained in this section consist of Branch & Bound, Graph, Tree, and Rooted Tree.

A. Breadth First Search (BFS)

BFS is one of the two basic uninformed search algorithms called graph traversal algorithm, alongside Depth First Search (DFS). BFS and DFS are done with an assumption that all nodes are connected.

BFS works as a simple Brute Force-like search algorithms. The way BFS works is as follows: First, determine an initial point of traversal by choosing a node. Check the initial node. If it does not match the desired search result, visit and check all unchecked neighboring nodes of the node first, then check all the unchecked neighboring nodes of each neighboring node of the initial node. This process is repeated until a node matches the search result desired. BFS works in a similar way to a queue, with First In First Out (FIFO) method.

Figure 2 is an illustration of how BFS algorithm works. First, we determine the initial node, numbered by 1. After searching node 1, we look at all unchecked neighboring nodes of node 1, numbered by 2 and 3. Then, after searching nodes 2 and 3, look at all neighboring nodes of node 2, numbered by 4 and 5, and search them first. Next, search all the neighboring nodes of node 3, numbered by 6 and 7. Subsequently, search all the unchecked neighbors of node 4, numbered by 8. The search ends with the termination trigger being no nodes fit the desired search result because there are no unchecked nodes left in the graph.

B. Branch & Bound

Branch & Bound is an algorithm that is used for optimization problems. Branch & Bound is used to minimize or maximize an objective function without violating the problem's constraints. How Branch & Bound works is the same as Breadth First Search (BFS), but with the addition of least cost search. Each node in a

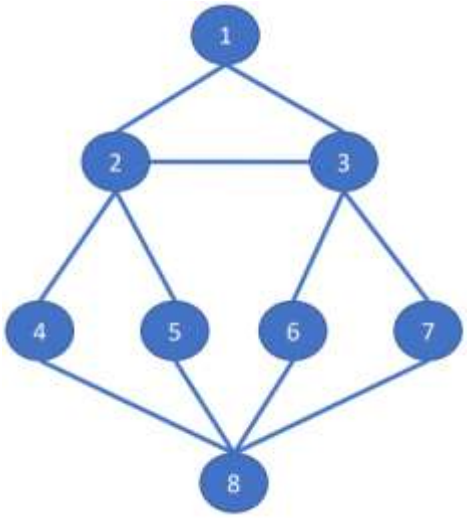

Figure 2. Illustration graph of BFS (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf)

set is given a specified cost based on the context of the problem and at the end of each iteration, after the child nodes have been generated, the set is arranged based on the cost in ascending order.

Using Branch & Bound algorithm to solve Rubik's Cube problem is similar to using it to solve 15-puzzle problem. In this case, the total cost consists of two different types of costs, displayed by this equation:

$$c(i) = f(i) + g(i) \quad (1)$$

$c(i)$ is the total cost, $f(i)$ is the amount of moves that the cube has made since the initial state, and $g(i)$ is the total difference between the current state and the finished state of the cube. $g(i)$ is initialized by 0 and increments by 1 for each square whose color does not match the color of the finished state's square.

C. Graph

Graph is a representation of discrete objects and their connections[4]. There are two components of a graph, which are displayed in this equation:

$$G = (V, E) \quad (2)$$

With G being the graph, V being a nonempty set of nodes, and E being a set of edges. A node is a discrete object that is represented in the graph, and an edge is a connection (usually drawn in a line) that connects a pair of nodes.

D. Tree

Tree is a fully-connected, directionless graph that has no cycles[5]. Based on the definition, for a graph to be considered a tree, the graph must fulfill three conditions:

- The graph does not have any cycle in it. This means that, if an initial point is determined randomly from any node, that point must not be able to travel to a visited node without backtracking its path.

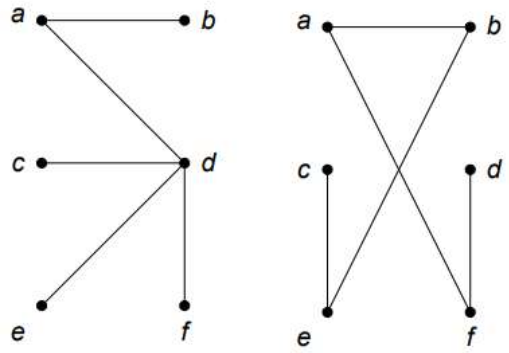

Figure 3. Examples of graphs that are also trees (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf)

- There is no directional edge in the graph
- All nodes in the graph must be connected. In other words, if an initial point is determined randomly from any node, that point must be able to travel to all nodes in the graph by going through the edges.

E. Rooted Tree

Rooted tree is a tree in which a node is considered the root and the edges are given direction away from the root[6]. Since a rooted tree is a tree, the direction sign of the edges can be omitted, but the direction of each edge is still clear, away from the root. There are 10 terms used in rooted tree:

1. Child

   A node is a child of another node if the former has lower level than the latter and they are directly connected by an edge. Using Figure 5 as an example, nodes 1 and 2 are children of node 0; nodes 3 and 4 are children of node 1.

2. Parent

   A node is a parent of other nodes if the former has higher level than the latter and they are connected by an edge. Using Figure 5 as an example, node 0 is the

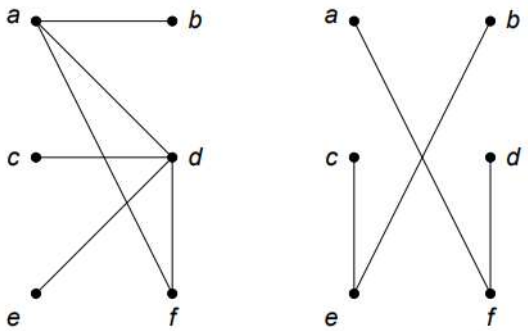

Figure 4. Examples of graphs that are not trees (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf)

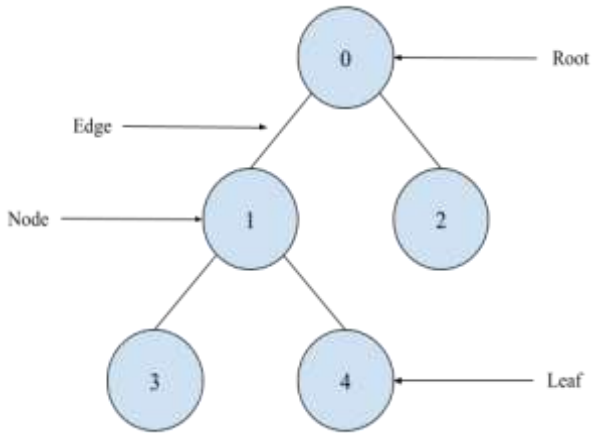

Figure 5. Rooted Tree

parent of nodes 1 and 2; node 1 is the parent of nodes 3 and 4.

3. Path

   A path from a node to another means the list of nodes required to be traveled to in order to go from the former to the latter. A length of a path is the number of the nodes in the path. Using Figure 5 as an example, a path from node 0 to node 3 is 0, 1, 3, with the length of the path being 3.

4. Sibling

   A node is a sibling of another node if both are different children of the same parent. Using Figure 5 as an example, node 1 is a sibling of node 2 and node 3 is a sibling of node 4.

5. Subtree

   A tree is a subtree of a node if the tree's root is a child of the node. Using Figure 5 as an example, a tree which consists of nodes 1, 3, and 4 is a subtree of node 0.

6. Degree

   A degree of a node is the amount of subtrees or children of the node. Using Figure 5 as an example, the degree of node 0 is 2, the degree of node 1 is 2, the degrees of nodes 2, 3, and 4 are 0.

7. Leaf

   A leaf of a tree is a node of the tree that has no children. Using Figure 5 as an example, nodes 2, 3, and 4 are leaves of the tree.

8. Internal nodes

   An internal node is a non-root node of the tree that has children. Using Figure 5 as an example, node 1 is an internal node.

9. Level

   The level of a node in a tree starts from 0 at the root, then increments by 1 for each node traveled from the root until a leaf is reached. Using Figure 5 as an example, the level of node 0 is 0, the level of nodes 1 and 2 are 1, the level of nodes 3 and 4 are 2.

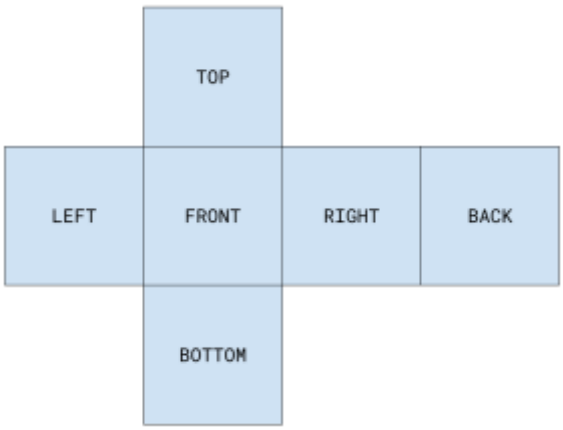

Figure 6. Flattened Rubik's Cube sides

10. Height/Depth

    The height or depth of a tree is the maximum level of a leaf node. Using Figure 5 as an example, the height of the tree is 2.

III. Implementation

This section discusses everything about the program. The things explained here are cube layout, the implementation of the algorithm, multiprocessing, and heuristic techniques implemented.

The first thing discussed in this section is the cube layout. The cube layout part of this section explains about how to see a Rubik's Cube for this paper's context, how each square is indexed to prevent confusion, and how the cube's layout translates to the program.

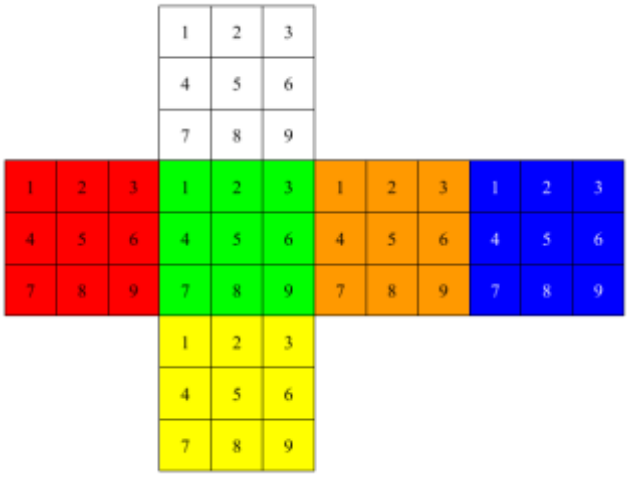

Figure 7. Flattened Rubik's Cube's squares, numbered

```
[[[1,1,1],[1,1,1],[1,1,1]],
 [[2,2,2],[2,2,2],[2,2,2]],
 [[3,3,3],[3,3,3],[3,3,3]],
 [[4,4,4],[4,4,4],[4,4,4]],
 [[5,5,5],[5,5,5],[5,5,5]],
 [[6,6,6],[6,6,6],[6,6,6]]]
```

Figure 8. How Rubik's Cube is translated into the program

Next, the section discusses about algorithm implementation. The algorithm implementation part of this section explains about all 18 moves a Rubik's Cube can do, how they are implemented into the program, the input and the output of the program, and how the program works.

After algorithm implementation, the section discusses about the implementation of multiprocessing into the program. The multiprocessing part of this section explains about the benefits of implementing multiprocessing into the program.

The last thing that the section discusses is the heuristic techniques used in the program. This part of the section explains how and where the heuristic technique is implemented into the program.

A. Cube Layout

The Rubik's cube layout is represented by that of a flattened cube. The figures in this chapter show how the cube is perceived. The flattened cube in Figure 6 shows each side of the cube and what side it represents before it is flattened. Figure 7 shows the layout of the squares on each side of the cube. Figure 8 shows how the cube is implemented in the program. The cube is implemented as a three-dimensional array with 6x3x3 size. The first dimension represents the sides of the cube, numbered by 0 to 5 and arranged as following: Top, Front, Left, Back, Right, and Bottom. The second dimension represents the rows of each side of the cube, numbered by 0 to 2 and arranged from the top (number 1-3 on Figure 7) to the bottom row (number 7-9 on Figure 7). The third dimension represents the columns of each row, numbered by 0 to 2 and arranged from the left (number 1, 4, and 7 on Figure 7) to the right column (number 3, 6, and 9 on Figure 7). For example, using Figures 6 and 7 as a reference, rubik[3][0][2] represents the square number 3 on the back side of the cube.

Each side is represented by a color, and each color is represented by a number, from 1 to 6. A solved cube is represented by Figure 8.

```
Enter scramble notation text file path here:
../test/testcase1.txt
What method would you like to use to solve the puzzle? Type the number only
1. BFS
2. Branch & Bound
2
```

Figure 10. Input of the program

B. Algorithm Implementation

The Rubik's Cube has 18 types of rotations[2]:

- F: Rotate the front side of the cube clockwise
- R: Rotate the right side of the cube clockwise
- U: Rotate the top side of the cube clockwise
- B: Rotate the back side of the cube clockwise
- L: Rotate the left side of the cube clockwise
- D: Rotate the bottom side of the cube clockwise
- Fi: Rotate the front side of the cube counter-clockwise
- Ri: Rotate the right side of the cube counter-clockwise
- Ui: Rotate the top side of the cube counter-clockwise
- Bi: Rotate the back side of the cube counter-clockwise
- Li: Rotate the left side of the cube counter-clockwise
- Di: Rotate the bottom side of the cube counter-clockwise
- F2: Rotate the front side of the cube (regardless of clockwise motion) twice
- R2: Rotate the right side of the cube (regardless of clockwise motion) twice
- U2: Rotate the top side of the cube (regardless of clockwise motion) twice
- B2: Rotate the back side of the cube (regardless of clockwise motion) twice
- L2: Rotate the left side of the cube (regardless of clockwise motion) twice
- D2: Rotate the bottom side of the cube (regardless of clockwise motion) twice

In the program, all 18 types of rotations are implemented. The rotations F2 to D2 are implemented by doing the single type of the rotations (F to D) twice, but are counted as one move instead of two.

The program can accept an input and print an output. The input of the program is a .txt file that consists of rotation notations meant to scramble the cube, and the output is the rotation notations needed to solve the cube and the execution time of the solving process.

After scrambling the cube, the current state of the cube is used as the first live node. The first live node is removed from

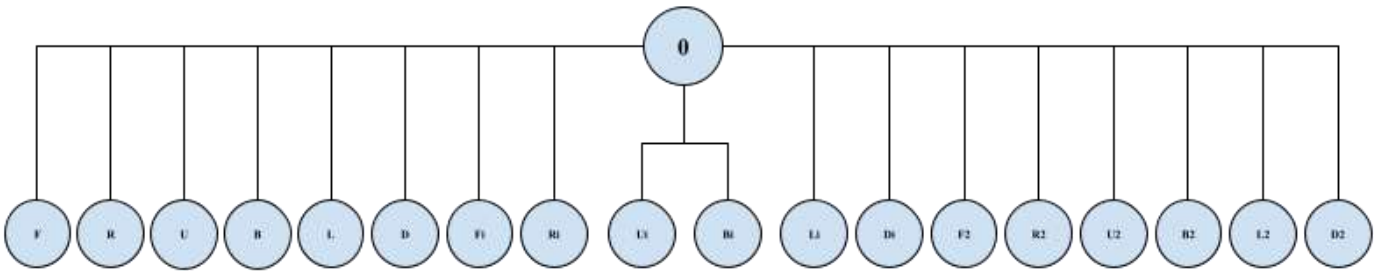

Figure 9. Rooted Tree of the first iteration of the program

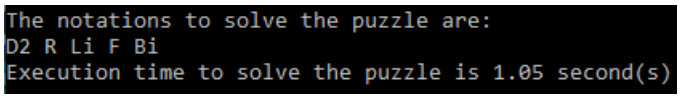

Figure 11. Output of the program

the live node list and used as the current node, and 18 child nodes (based on the 18 different types of rotations) are generated and inserted into the live node list. The list of live nodes is then arranged.

The program uses Branch & Bound algorithm to solve the cube. The algorithm is implemented on the live node list arrangement function. The function uses Merge Sort algorithm to arrange the live nodes and they are arranged based on the number of the sum of f(i) (the amount of rotations that the cube has done on that node) and g(i) (the difference between the current state and the solved state) of each node in ascending order.

The first element of the live node list after being arranged is picked as the next current node. Then, the same process as the previous current node is repeated until the g(i) of the current node is 0. The loop is then terminated, with the current node considered as the solution.

C. *Multiprocessing*

The program to solve Rubik's cube is created using Python programming language, which means it only uses one CPU core by default. To speed up the solving process, multiprocessing is implemented into the program. Multiprocessing allows the CPU to use all cores, instead of just one, to process a part of a program. Using multiprocessing on a compute-heavy process will reduce execution time significantly.

Multiprocessing changes the algorithm of the program a little bit. The first 18 child node generation is done using single core processing. After the 18 child nodes are in the live node list, multiprocessing is implemented, by computing all 18 child nodes at once. Each child node will have its own live node list and it will be arranged separately. This not only speeds up the solving process in general, but also divides the amount of live nodes in each list by 18. This will make arranging live node lists, one of the most compute-heavy parts of the program, a lot faster, allowing for storing significantly more nodes compared to single core processing.

Multiprocessing is implemented using Multiprocessing library, with functions Pool to create multiple processes and Starmap to map the solving function to different processes. Multiprocessing Queue is used to communicate between different processes. Once a process finds a solution, it will put an integer into the empty queue. All processes will stop if one manages to find the solution first, indicated by the queue not being empty.

D. *Heuristic Techniques*

To reduce the amount of states that the program must check, a heuristic is placed. The heuristic technique used is as follows: Depending on the last rotation of the current node, certain nodes are exempt from being generated, because the cube states of the nodes have been covered by other existing nodes. Here are some examples of the heuristic:

- If the last rotation of a node is F, a child node with F rotation is not generated. This is because the rotation makes the child node's cube state equivalent to a sibling of the parent node's with F2 as its last rotation, and is therefore redundant.
- If the last rotation of a node is F, a child node with Fi rotation is not generated. This is because the rotation makes the child node's cube state equivalent to the parent of the parent node's, and is therefore redundant.
- If the last rotation of a node is F, a child node with F2 rotation is not generated. This is because the rotation makes the child node's cube state equivalent to a sibling of the parent node's with Fi as its last rotation, and is therefore redundant.

IV. TESTING

This section discusses about the testing part of the paper. It consists of limitations put in place to reduce the amount of computing power the computer needs, the test cases and conditions, and the test results.

A. *Limitations*

Rubik's Cube has approximately 43,252,003,274,489,856,000 (43 quintillion) possible states[1]. Even with heuristics and multiprocessing, solving a randomly scrambled Rubik's cube using Branch & Bound on a computer will require a lot of RAM. Therefore, a limitation is set, in that the input test cases used in this test will only contain a maximum of 5 rotations as scramble notations for the cube. This limitation is set to reduce the amount of RAM that the computer requires to solve the puzzle.

B. *Testing Conditions*

The Rubik's cube program is tested against the exact same program, but with Breadth First Search (BFS) as its solving algorithm instead of Branch & Bound. There are three test cases used, each with 5 rotations as scramble notations. Here are the test cases:

- Test Case 1: B Fi L Ri D2
- Test Case 2: F L2 F Di R
- Test Case 3: Li Di B Ri Fi

There are two variables that are tested in this test, and both are measurements of the algorithm's performance on solving Rubik's cube puzzle. The first variable is the execution time. The less time the program requires to solve the cube, the more performant the program is at its job. The execution time is counted from right before the start of the solving function to right after the end of it. The execution time is measured by Time The second variable is the RAM usage. The less RAM the program uses, the more efficient the program is at its job. The RAM usage number is collected from Task Manager's display of the Python task.

The computer used for testing of the programs has these specifications:

- CPU: AMD Ryzen 7 5700U
- RAM: 7.4GB DDR4 (around 5.8-5.9GB free)

C. Test Results

No	*Algorithm + Test Case*	*Execution Time (s)*	*RAM Usage (MB)*
1	BFS + TC 1	50.1	5,185
2	Branch & Bound + TC 1	0.86	138
3	BFS + TC 2	FAIL (RAM full)	5,916
4	Branch & Bound + TC 2	0.85	168
5	BFS + TC 3	22.49	2,927
6	Branch & Bound + TC 3	0.11	102

TABLE I. TEST RESULTS OF BRANCH & BOUND AND BFS ACROSS THREE DIFFERENT TEST CASES

The table shows a clear and huge win for the solving program with Branch & Bound algorithm compared to BFS. For the first test case, the Branch & Bound algorithm manages to find the solution 58.26 times faster while consuming 37.57 times less RAM. Test case 2 is where things get interesting, as BFS algorithm fails to find the solution due to the computer not having enough RAM, while Branch & Bound algorithm manages to find the solution just fine with quick execution time and nowhere near full RAM usage. Test case 3 tells us that Branch & Bound algorithm manages to find the solution 204.45 times faster while consuming 28.7 times less RAM compared to BFS algorithm.

V. CONCLUSION

Branch & Bound algorithm is proven to be superior compared to BFS. This is due to the nature of Branch & Bound being an informed search algorithm. However, neither algorithm is effective enough to be able to completely solve a fully randomized Rubik's Cube without a ridiculous execution time and an unrealistic amount of RAM usage. Considering both Branch & Bound and BFS are more general algorithms (algorithms that work with a lot of different problem solving cases), a more specific Rubik's Cube solving algorithm is required to make this task possible.

VIDEO LINK AT YOUTUBE

https://youtu.be/7H0vdNhp61E

GITHUB REPOSITORY

https://github.com/13518014Ignatius/IF2211_Paper

ACKNOWLEDGMENT *(Heading 5)*

In this section, I would like to express my eternal gratitude towards God for allowing everything in my life to happen. I would also say my thanks to my family and friends for supporting me all the way up to the point that I manage to finish this paper. Special thanks also goes to Bandung Institute of Technology, STEI faculty, Informatics Engineering department, and Dr. Nur Ulfa Maulidevi, S.T, M.Sc. as my lecturer for the Algorithm Strategies course, for allowing this course to exist and helping me in learning about various algorithms necessary to finish this paper. Finally, I would like to apologize for any mistake in this paper.

REFERENCES

[1] Zeng et al., *Overview of Rubik's Cube and Reflections on Its Application in Mechanism* (2018), https://doi.org/10.1186/s10033-018-0269-7, accessed on May 6th, 2022.

[2] Ferenc, D., *Rubik's Cube Notations*, https://ruwix.com/the-rubiks-cube/notation/, accessed on May 4th, 2022.

[3] Munir, R., "Breadth/Depth First Search" (2021), https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf, Accessed on May 12th, 2022

[4] Munir, R., Maulidevi, N.U., Khodra, M.L., "Algoritma Branch & Bound (Bag.1)" (2021), https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Branch & Bound-2021-Bagian1.pdf, Accessed on May 12th, 2022.

[5] Munir, R., "Graf (Bag.1)" (2021), https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf, Accessed on May 11th, 2022.

[6] Munir, R., "Pohon (Bag.1)" (2021), https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf, Accessed on May 11th, 2022.

[7] Munir, R., "Pohon (Bag.2)" (2021), https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf, Accessed on May 6th, 2022.

STATEMENT

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

Bandung, 22 Mei 2022

Ignatius David Partogi, 13518014