Application of Hashing and Breadth First Search in the search for God's Number of the Pyraminx

Farhan Nafis Rayhan - 13522037[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*author@itb.ac.id*

***Abstract*—The Pyraminx is one of the most popular variation of the rubik's cube. It's pyramid like shape has captivated enthusiasts with its complexity and diverse solving strategies. One problem in the twisty puzzle community that relates heavily to discrete maths is the God's Number, the maximum number of moves needed to solve any puzzle. In this paper, we will delve into the exploration of God's Number for Pyraminx, and show how it can be found using Breadth First Search and hashing.**

***Keywords*—Pyraminx, Hashing, BFS, Graph**

I. Introduction

The Pyraminx is a twisty puzzle invented in 1970 by the German Puzzle designer, Uwe Mèffert [1]. In the same vein as it's sibling, the world famous Rubik's Cube, Pyraminx consisted of sticker colored puzzle pieces, yet instead of cube shaped, Pyraminx is a regular tetrahedron. The unique shape is the reason why Pyraminx is one of the most popular yet interesting among all other Rubik's cube variations. Pyraminx consists of 4 faces, each given different color. The puzzle is said to be in a solved state if every face shows same colored piece.

All 4 faces are split into 3 different layers, as shown in the picture below, which could be turned in a clockwise manner or otherwise. What follows is 16 different possible rotation to change the state of the puzzle. Any rotation from the solved state brings into an unsolved state, another rotation might discover a new state, and so on. This is why twisty puzzles like pyraminx are so beloved, every other rotation gives us a new scramble we had never encounter before, leading us a step further from the solved state.

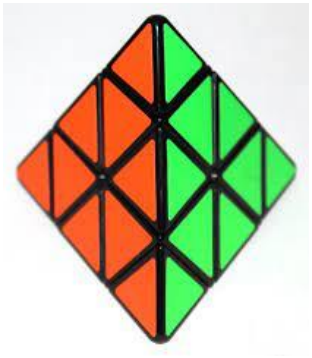

Fig 1.1. Solved state of the Pyraminx
(Taken from wikimedia.org)

Some sequence of moves actually brings us closer to solving the puzzle. These such sequences are called "Algorithm", not to be mistaken by the general definition of algorithm. Throughout the years, there are many algorithms and different methods discovered to solve the Pyraminx. However, some methods are known to be faster than the other, and the most effective one had been in constant search since the history of this puzzle. Among the fastest solution, there had been another mystery looming since the dawn of Rubik's cube.

What is the least number of moves needed, to ensure that we can solve the Pyraminx, from any of it's possible scramble? The answer of this problem is known as the God's Number. This is only one example of the intersections between the world of discrete mathematics and the famous puzzle. It took mathematician about 35 years of research and computer calculation to finally determine the answer for the original 3x3x3 Rubik's cube. On July 2010, it was determined that the God's number is 20 [2], proven by Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge. But for the Pyraminx , the God's Number haven't been clearly documented, or perhaps it there haven't been any resource of it's formal proof. Therefore, it is the subject matter of this paper.

The formal proof for the 3x3x3 cube God's Number used a deep understanding of group theory and complex computer algorithms to search for optimal solving solutions. This is beyond the scope of this paper, so we will approach the problem differently. We will picture every state of the Pyraminx as a graph vertice, where the edges are moves that would connect each state. The solution could be achieved by searching for the shortest path between each state and the solved state. Breadth First Search would be used to do the task. For optimization and representing the state of the puzzle, we will utilize a hash algorithm.

II. Graph Theory

A. Definition

Graph are discrete structures that is represented by vertices and edges that connects them. Formally, graph is defined as $G = (V, E)$ where V is a nonempty set of vertices and E is a set of edges [3]. Each edges are connected to at least 1 vertices, also called end points. For directed graph, each edge is assumed as an ordered pair (v_s, v_e) which signifies that an edge starts from vertex v_s and ends at v_e.

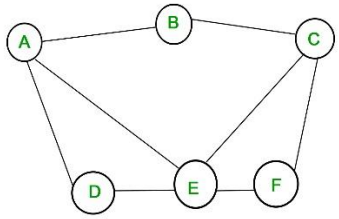

Fig 2.1 Graph Illustration
(Taken from geeksforgeeks.org)

To further deepens the understanding, we will be using graph terminologies described below.

B. Terminology

Adjacency

If 2 vertices *u* and *v* are endpoints of an edge, where *(u, v)*, then *u* is said to be adjacent to *v*, while *v* is adjacent from *u*. Especially in a graph with directed edges, *u* is also named as the initial vertex, and *v* is the terminal vertex.

In-degree & out-degree

For a directed graph, the in-degree of *v* is the number of edges with *v* as it's terminal. On the other hand, the out-degree of *v* is the number of edges with *v* as it's initial vertex. In-degree and out-degree of *v* is denoted by $deg^{-}(v)$ and $deg^{+}(v)$ repectively. For example in the figure below, vertex 3 has 1 out-degree and 3 in-degree.

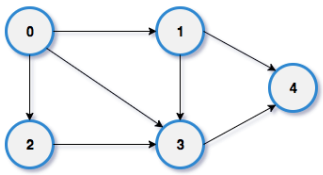

Fig 2.2 In-degree & Out-degree example
(Taken from log2base2.com)

Multiple Edges

If there is a pair of edges that connects the same vertex. In a directed graph, if there is 2 different edges that assumes the same pair *(u, v)*, therefore this pair of edges is called multiple edges.

Self Loop

An edge that connects vertex *(u, u)*, in other words it's and edge where the initial vertex is also the terminal.

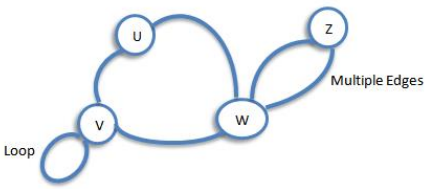

Fig 2.3 Multiple Edges & Self Loop
(Taken from scanftree.com)

Path

A sequence of edges that starts at a vertex of a graph and walks along the edges according to the direction, visiting vertices that acts as the terminal of those edges. For a path with length *n*, the notation is $x_0, x_1, x_2, \ldots x_n$ which is the sequence of *n+1* vertices it runs into.

Cycle

Special type of paths that starts and ends at the same vertex, in other words, for a path with length *n* denoted by $x_0, x_1, x_2, \ldots x_n$, we must have $x_0 = x_n$.

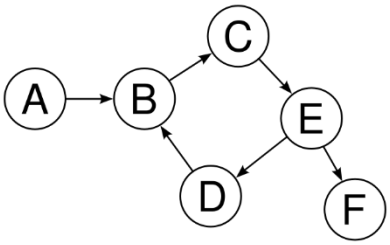

Fig 2.4 Cycle B,C,E,D,B in a directed graph
(Taken from codingninjas.com)

Simple Graph

Any graph that doesn't contain multiple edges or self loops is called a simple graph. Otherwise, it is said to be non-simple. Usually non-simple graph are divided even further into 2 different class depending on the existence of multiple edges or the self loop (Multi-graph & Pseudo-graph) but we won't delve into it further since these 2 classification is beyond the scope of this paper.

In the figure below, we can classify the first graph as a simple graph since it doesn't contain neither self loops nor multiple edges. The second graph is non-simple due to the existence of 2 different edges connecting the same pair of vertices, one of which is the horizontal pair of vertices. Finally the last graph is also considered to be non-simple for the existence of 2 self loops at the right most vertex.

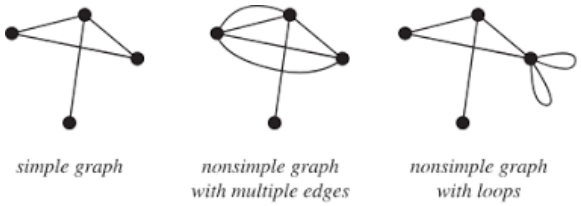

Fig 2.5 Examples of simple and non-simple graphs
(Taken from mathworld.wolfram.com)

III. Background on the Pyraminx

A. Pieces

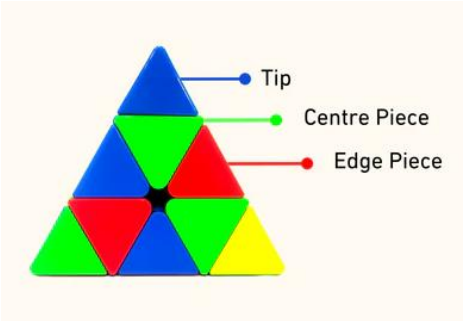

Fig 3.1. Pyraminx pieces names
(Taken from cubelelo.com/blogs)

Every Pyraminx consist of 3 type of pieces. The pieces shaped like an upside down triangle is called the Center, due to the fact that they can only rotate around an axis. Each center have 3 colors corresponding to which face it is in, and it can't move to any other position. Everytime we turn a layer, the center would also turn in the same direction.

The only pieces that could move to another position are the edges. Edges consist of 2 colors each, corresponding to faces it should be in. Everytime a layer is rotated, all the edges on that layer would permute in a 3-cycle. Not only it can permute, edges orientation should also be considered.

The last pieces are the tip, which is as it's named, tips located at every tip of the 4 vertices. A tip consists of 3 different colors, corresponding to the center it's next to. That tip shares an axis with the center, but tips could be rotated independently without interrupting any other pieces. Due to this reason, the tip will be ignored in this paper, since it could be considered to just be combined with it's center as a same piece. Calculation using the tip is deemed unnecessary and would take too much space at implementation. Even adjustment to the actual god's number could easily be done at the end.

B. Notation

Notation for the moves of Pyraminx should be defined clearly early on. This is due to the fact that we will be using the moves as edges in our graph, bridging between states of the puzzle. Throughout this paper, we will assume that the puzzle is hold in a way such that the red face is up front, and the blue face at the bottom.

Rotation in twisty puzzles is usually notated by a letter [1]. In the Pyraminx, there are 4 different axis that can be turned, which is shown by the location of the center pieces. Each rotation in a clockwise manner is notated by a capital letter, signified by the first letter of the axis turned (**u**p, **r**ight, **b**ack, **l**eft). The axis can be turned the other way around too, such counter clockwise moves is notated with an apostrophe (') after the letter.

So far we have defined 8 different possible moves from any state. It should be noted that since the tip could be rotated independently, they have their own moves that affects no other pieces. The rules are not much different than the usual rotation, however we use non-capital letters instead. As stated before, tip would be ignored, so we would eliminate the 8 possible moves from consideration for the rest of the paper. Thus, we are only using 8 moves to work with the Pyraminx, as viasualized below.

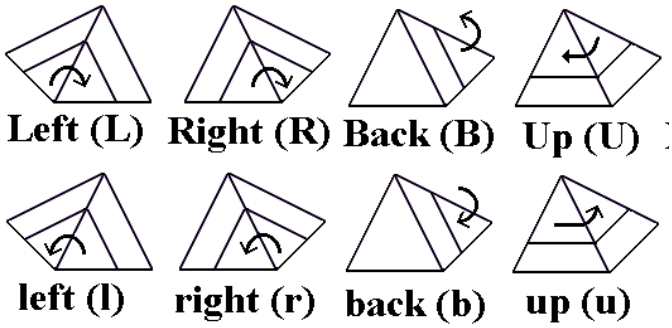

Fig 3.2. Pyraminx move notations
(Taken from solitairelaboratory.com)

IV. Algorithm

There are several algorithms that we will use to search for the God's Number. The concepts and time complexity will be elaborated below, while the implementation for this paper will be described in later chapters.

A. Hashing

Hashing is a process of generating a fixed and small size output from a large and variably sized input [4]. This is done with a specially chosen function called the Hash function. In computer science, hash is usually used to determine the first index to store our data in a data structure. The index is optimized to be hashed as small as possible such it's not using as much space, yet also leaves as many space as possible for the data to fit.

A data that is hashed usually paired with a key, which is unique and would be use to recognize a single data. The selected hash function will convert the key into a number or address in the size small enough for the data structure. Hash is also utilized to optimize the retrieval process of data. Since the hash function return the address of the data, we can also determine whatever data is paired with a certain key just by inserting the key into the function. Optimally, a perfect hash function would run in *O(1)* time, which speeds up memory look up significantly.

However, hashes is not without it's complication, since a collision might occur. Collision is a condition where a different key value returns a same hash value instead. There are several method for collision resolution policy, such as open addressing or double hashing. Fortunately in our case, selecting a perfect hash function could avoid collisions completely. Thus, we won't delve further into collisions and their problems.

B. Breadth First Search (BFS)

Breadth First Search, or more commonly abbreviated as BFS, is an efficient algorithm to traverse complexly connected data structures such as graphs [5]. This algorithm will start at a starting vertex called a root, and visits vertex at increasing order of distances. Different from it's sibling, the Depth First Search (DFS), which traverse the graph such that we get to the end of the path before we backtrack, BFS is the opposite manner. We

will be walk through a path 1 step at a time, moving on into the next path instead of continuing our current path. This is why the algorithm is called Breadth first, a representation of our approach to walk through said graph.

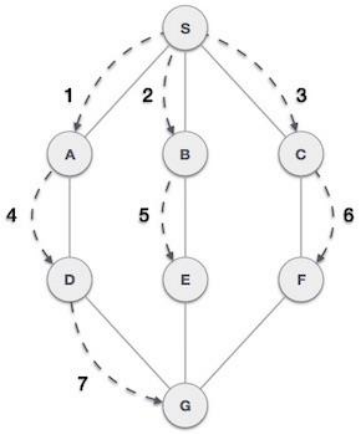

Fig 4.1. Breadth First Search Illustration
(Taken from tutorialspoint.com)

The Breadth First Search algorithm utilizes a first in first out (FIFO) structure. Queue is an excellent choice for this algorithm since it's implementation is usually efficient enough. The queue will be used to store the order of vertex exploration, at each step the front of the queue is processed. Another structure needed for storing the truth value of whether a vertex has been visited before. A Boolean typed array sized exactly the number of vertices works fine. We also need to store the distance of a vertex from the root, where an integer typed array is sufficient. Below is a pseudocode explaining how the algorithm works.

```
q: queue
visited: array of boolean with size [1..numberOfVertex]
distance: array of integer with size [1..numberOfVertex

procedure breadthFirstSearch(input x: vertex)
    v: vertex
    initializeVisitedToFalse()
    visited[x] <- true
    distance[x] <- 0
    while(not isQueueEmpty(q)) do
        v <- pop(q)
        process(q)
        for (u <- vertices of v) do
            if (not visited[u])
                visited[u] <- true
                distance[u] <- distance[v] + 1
                push(q, u)
```

Fig 4.2. Breadth First Search Pseudo Code

For a graph $G = (V, E)$, the time complexity for Breadth First Search on graph G is *O(V+E)*.

V. Lower Bound

To set a standard for the solution we will look for later, it is a good idea to know the lower bound of the God's Number first. Using a naive approach with combinatorics is sufficient.

A. Number of Combinations

We will look for the number of all possible shuffle of the Pyraminx. As stated before, the tip won't be accounted in this paper. Notice that for each center, their position is always fixed on an axis, but they can be rotated up to 3 faces. Since there are 4 independent centers, the number of combinations are 3^4. The edges would have more combinations since we won't only count it's orientation, but also it's permutation. Each edge have 2 stickers, meaning there are 2^6 possible orientation. While we have 6! ways to permute the edges location within the Pyraminx.

However, limitations on the structure of the puzzle needs us to consider parity cases for both permutation and orientation. The problem is due to the 3-cycle nature of pyraminx edges permutation, it is impossible to have a 2-cycle edges. Thus, we must eliminate it by dividing the final calculation by 2. The same thing applies for orientation too. Note that due to the nature of the puzzle, it is impossible to have a single flipped edge on a Pyraminx. This case can also be eliminated by dividing by 2. Therefore, the actual number of combinations possibly achieved on a Pyraminx is

$$\frac{3^4 \times 2^6 \times 6!}{2 \times 2} = 933120$$

B. Important Assumption

Consider that at every state of the puzzle we may use 8 different moves (U, U', R, R', B, B', L, L'). Assume that every time we do a move we will guaranteed to discover a new, unvisited before state. This assumption is obviously not always true, since there might be a cycle in the graph of possibility we're making, bringing us back into a state visited before instead. But on the other hand, this assumption results int the overcalculation of states we visit at every number of moves, which is important for the following calculation.

Starting from the solved state, we have 8 possible turns to move into a different state. For the next move, we only have 7 choices, since one of the moves here is just the inverse of what we just done, making our effort before useless. We will keep on moving to a new state every round, in each we have 7 choices of turns. Based on our assumption above, we will at one point reach a round where the number of states visited is actually more than what's possible (remember that we will overshoot the number of visited states). This is our lower bound, since in actuality we might need more rounds to finally visited all of them. We arrived at this inequality.

$$8 \times 7^{n-1} \geq 933120$$

Where *n* is the number of moves made. Therefore, the lower bound of the God's Number is the lowest possible *n*. Which turns out to be *7* moves.

VI. Proposed Method

The proposed method to search for Pyraminx's God's number would involve hash and breadth first search. Further details about the approach is as follows.

A. State Notation

Every state of the puzzle will be notated by a string of 16

integers. The notation is made with specific rules such that every possible state of the Pyraminx is uniquely notated. Each integers representing a single piece independently. Informations about center orientations, edge orientation, and edge permutation are included in this string.

The first 4 integers of the string represents orientation of each center. Numbered from 0 until 2, it's informing us how many clockwise rotation it is currently from the correct orientation. The centers are ordered from up, right, back, and left.

The next 6 integers are the orientation of the edges, which is either a 0 or 1. The following idea is borrowed from a lecture by Zachary Stillman & Bowen Shan [6], and expanded further to improve the handling for purposes of this paper. For each edge piece, a certain sticker is selected and labeled as by a special symbol '+'. The integers corresponding to an edge position is '1' if the edge located here has it's '+' on the correct orientation as the '+' symbol of this position. At the solved state, all 6 integers are '0' since every edge is in it's correct orientation. Notice that the number of 1's here should be even, which is an effect of the orientation parity. The scheme of the special stickers is illustrated below.

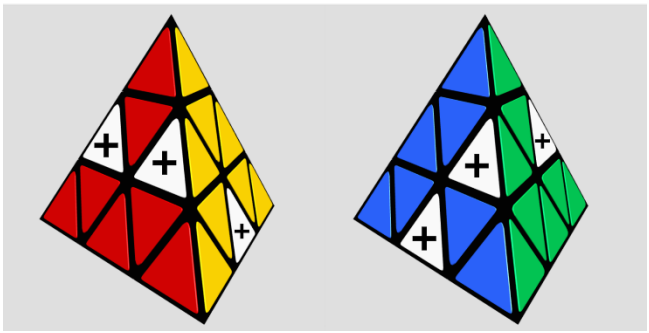

Fig 6.1. Special stickers scheme

The last integers of the string depicts the permutation of edges. Once again, this idea is adopted from the lecture "Group Theory and the Pyraminx" [6]. The integers are from the range of 0-5, which is analogous to the number of edges. Each edges are given a number, and each number in the strings shows what number is the edge located here. In the solved state, the string is simply "012345".

In simple terms, the last 6 numbers is a permutation of numbers from 0-5, with a little adjustment. One important thing is the fact that Pyraminx edges can only permute in 3-cycles. This means a single 2-cycle is impossible to be achieved, same as a 4-cycle and 5-cycle. Thus, the last 6 numbers must be constrained such that it satisfies the cycle condition. This can be done by swapping the last 2 numbers if the cycle condition is violated. The numbering scheme for the edges as follows

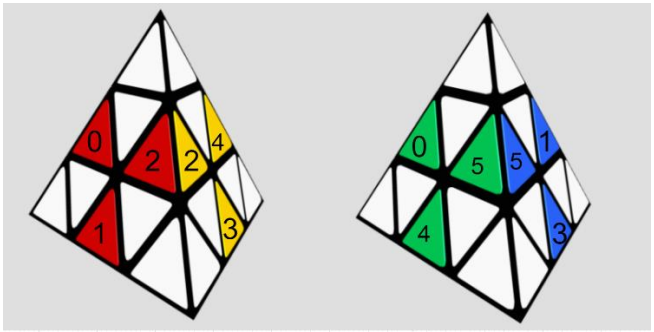

Fig 6.2. Edges Numbering scheme

For better understanding, the solved state would is notated "0000000000012345"

B. Hash Function

Hash is not only needed as a simpler representation of the states, it's also crucial for space optimization in our implementation. The hash function in this paper is perfected such that no collision would ever happen. This function converts the state notation string described above into an integer in the range of 0-933119, which is the number of all possible state of Pyraminx.

The first 4 integers are converted into an integer using the concepts of Base 3, which becomes a number between 0-80, assume it's A. The next 4 are 6 digits of 0's and 1's. Remember that the parity of the number of 1's should be even, so we can ignore the last digit and only look at the first five. This string of 5 can be assumed as a binary and converted easily into a number between 0-31, For example B.

Lastly we need to convert a permutation of numbers between 0-6 into an integer in the range 0-360, call it C. As stated before, the last 2 numbers could've been swapped to satisfy the 3-cycle condition. Therefore, we only need to focus at the first 4 numbers (since the last 2 are determined from them). To convert these 4 numbers into a single integer is analogous to finding out what order a permutation is if all are sorted lexicographically.

All these steps results into 3 integers as stated before. We can combine them all into one number using the equation

$$Hash\,Value = 2592 \times C + 81 \times B + A$$

this hash value is guaranteed to be between 0-933119. To retrieve a state string from the hash value, we can create a new function which is the inverse of the hash function, the steps are similar as above but reversed. Pseudo code for this algorithm provided below

```
function scrambleIntoInt(arr: array of short) -> int
    num <- 0
    permutation <- 0
    i <- 10
    p <- 0
    j <- 0
    k <- 60
    six <- [0, 1, 2, 3, 4, 5]
    used <- [false, false, false, false, false, false]
    while i < 14 do
        j <- 0
        p <- 0

        while six[p] is not equal to arr[i] do
            if not used[p] then
                j <- j + 1
            p <- p + 1
        used[p] <- true
        permutation <- permutation + (k * j)
        k <- k / (15 - i)
        i <- i + 1
    orientation <- 0
    i <- 8
    while i > 3 do
        orientation <- orientation << 1
        orientation <- orientation + arr[i]
        i <- i - 1
    center <- 0
    while i > -1 do
        center <- center * 3
        center <- center + arr[i]
        i <- i - 1
    num <- permutation
    num <- num * 32
    num <- num + orientation
    num <- num * 81
    num <- num + center
    -> num
```

Fig 6.3. Hash Function Pseudo Code

C. Graph Representation

The graph used is a simple directed graph that represents every possible scramble which can be achieved from a solved state, meaning there are 933120 vertices. All vertex fundamentally stores 4 values. The hashed value of the state represented which used for identification, a mark on whether this vertex is visited before, the distance from the solved state, and the last move done by it's neighbor to reach this state.

On the other hand, the directed edges connects a vertex to other states it can reach using only a move. In theory, each vertex must have out-degree 8 since there are 8 possible moves. But notice that doing moves in the same axis as the one did last is useless, since we are either cancelling the last move or going to a neighbor of the last state (which is guaranteed to have been visited before). Thus, for every vertex we only create 6 edges which is every moves in axis other than the last.

D. Breadth First Search

Set the solved state as the root of our search. The algorithm is no different than the pseudo code for BFS provided before, except for the process that we do for each vertex. Essentially, the process that we do to each visited vertex is we hash a state created by a valid move, then we would update the distance of that vertex to be 1 more than the vertex we're in.

At one point, we will visit every state and the search is stopped. Since we started from the solved state, and visited every possible scramble of the puzzle, we can finally find the answer. The God's Number for the Pyraminx is the highest distance value of all vertex.

Reminder that we are excluding the tips at our calculation, which means the number resulted is not exactly the god's number. Fortunately it is quite easy to calculate, since the tips are independent from any other pieces. We can assume the worst case scenario of each tip is wrongly oriented. This is needs at most 4 moves, where each tip can be fixed with just 1 move. Therefore, the final answer found from the Breadth First Search must be increased by 4 to return the actual God's Number.

E. Time Complexity Analysis

The hash function is made to run as fast as possible. In fact this hash function complexity is simply *O(1)*. This is true since every part of the answer is determined with calculation, and every loop size is constant.

Since every vertex essentially has 6 neighbors, and we won't visit any visited vertex again, we can say we visit each vertex only once. Noticed that we also visited every edges in our graph. In our graph $G = (V, E)$, we have $|v| = 933120$ and $|e| = |v| \times 6$. Thus we must have

$$T(n) = |v| + |e| = 7 \times |v|$$

and implies that the complexity of the search is *O(|v|)*. Which in theory should run in mere seconds even for a graph that big.

VII. Program Implementation

I used C++ for the implementation of this program since I want the program to have a fast runtime. But it turns out that the available memory for this language is quite limited. Therefore, space optimization is heavily needed for the implementation. Such as the use of smaller data types like short and deallocating used structure as fast as possible. The program simply utilizes command line interface. The full source code could be viewed at my git repository [7].

A. Pyraminx Data Type

The Pyraminx is basically implemented as an Abstract Data Type (ADT) called Node. This structure holds 2 values, first of which is the hash value called id, and the other is distance to record the number of moves it took to reach this state. Another structure used is Address, a pointer to a node. The function createNode is used to allocate and initialize a new node

```
...
#ifndef PYRAMINX_H
#define PYRAMINX_H

const int graphSize = 933120;

typedef struct node* Address;
...
typedef struct node{
    int id;
    short distance;
}Node;

Address createNode(int newId, short newDistance);

#endif
```

Fig 7.1. Pyraminx ADT

B. Moves.h

This header file is used to define all the moves and hash function. The matrix orientationUpdate is used to store every

edge orientation change due to a move, while permuationUpdate is handling edge permutation. The function scrambleToInt is essentially the hash function, converting any scramble notation into a hash value. While intIntoScramble is it's inverse hash function. doMove is used to apply a move to a state notation, then return the hash value of the resulting state.

```h
#ifndef MOVES_H
#define MOVES_H

#include "pyraminx.h"
#include <array>

// moveList = {U, U', R, R', B, B', L, L'}

const std::array<std::array<short, 6>, 8> orientationUpdate = {{
    {0, 0, 1, 0, 1, 0},
    {1, 0, 1, 0, 0, 0},
    {0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0},
    {0, 1, 0, 0, 0, 1},
    {1, 0, 0, 0, 0, 1}
}};

const std::array<std::array<short, 6>, 8> permutationTransition = {{
    {2, 1, 4, 3, 0, 5},
    {4, 1, 0, 3, 2, 5},
    {0, 3, 1, 2, 4, 5},
    {0, 2, 3, 1, 4, 5},
    {0, 1, 2, 5, 3, 4},
    {0, 1, 2, 4, 5, 3},
    {5, 0, 2, 3, 4, 1},
    {1, 5, 2, 3, 4, 0}
}};

void intIntoScramble(int num, short *arr);

int scrambleIntoInt(short* arr);

int doMove(short *state, short moveNum);
#endif
```

Fig 7.2. Moves header file

C. Search.h

This file contains the implementation of the Breadth First Search algorithm. One important thing to note is the queue data structure used is the one provided in the standard library. However, the element type it accepts is a bit different. For this queue, I construct it such as it accepts a pair of Node and short. The node at the top of the queue is the state that will be processed. While the short is used to store the last move done to reach this said node, such that we won't do moves at the same axis anymore.

```h
#include"moves.h"

#ifndef SEARCH_H
#define SEARCH_H

void bfs(short* answer, int* numNodesVisited);

#endif
```

Fig 7.3. Search header file

D. Main.cpp

This file is the driver file for the program. This file contains all the I/O displayed to the command line. Here is also where the bfs function is executed. The main program also prints several useful information such as the runtime of the program, also the amount of nodes found at each depth.

```cpp
#include<bits/stdc++.h>
#include"search.h"

using namespace std;

int main(){
    ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
    short godsNumber;
    int nodesVisited;
    std::cout << "
    std::cout << "
    std::cout << "
    std::cout << "
    std::cout << "
    std::cout << "
    std::cout << "
    std::cout << "
    cout << "Welcome to the Pyraminx God's Number finder"<< "\n";
    cout << "This Project was created by Farhan Nafis Rayhan as an implementation for his paper"<< "\n";
    cout << "The program utilizes algorithms such as Breadth First Search & Hashing"<< "\n";
    cout << "Please wait while the program looking for the answer..."<< "\n";
    auto start = std::chrono::high_resolution_clock::now();
    bfs(&godsNumber, &nodesVisited);
    auto end = std::chrono::high_resolution_clock::now();
    cout << "All, Nodes had been visited, the exact number of Nodes visited is " << nodesVisited << "\n";
    cout << "The God's Number for Pyraminx is " << godsNumber << "\n";
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    cout << "time taken by the program is " << (double) duration.count()/1000 << " seconds." << "\n";
    return 0;
}
```

Fig 7.4. Main Program file

E. Test Run

```
PS C:\Users\Farhan Nafis Rayhan\Documents\IT8\Semester 3\God-s-Number-for-Pyraminx> ./main
Welcome to the Pyraminx God's Number finder
This Project was created by Farhan Nafis Rayhan as an implementation for his paper
The program utilizes algorithms such as Breadth First Search & Hashing
Please wait while the program looking for the answer...
 Number of nodes with distance 0 is 1
 Number of nodes with distance 1 is 8
 Number of nodes with distance 2 is 48
 Number of nodes with distance 3 is 287
 Number of nodes with distance 4 is 1705
 Number of nodes with distance 5 is 9920
 Number of nodes with distance 6 is 54789
 Number of nodes with distance 7 is 247381
 Number of nodes with distance 8 is 503475
 Number of nodes with distance 9 is 115336
 Number of nodes with distance 10 is 170
 Number of nodes with distance 11 is 0
All, Nodes had been visited, the exact number of Nodes visited is 933120
The God's Number for Pyraminx, if we include the tips, is 14
time taken by the program is 1.624 seconds.
PS C:\Users\Farhan Nafis Rayhan\Documents\IT8\Semester 3\God-s-Number-for-Pyraminx>
```

Fig 7.5. Test Run Result

Above is what the program displays when it is run. Clearly the program runs quite fast, averaging at about 1.56 s every run. The program also gives consistent results, printing out that the gods number is 14. Based on a sequence in Sloaine's database [9] (also known as Online Encyclopedia of Integer Sequences), there are 32 cases in the Pyraminx that needs 11 moves to solve. This implies that the actual God's Number is 15, and my implementation missed by 1.

As it could be seen below, the number of nodes at depth 3 on my program and the ones in the database differ by 1, and it would differ further more until it left no more nodes in the depth 11. This is a bug in my implementation that I suspect is due to my bfs implementation, since I had double checked that my hash function won't deliver any wrong results.

A079744 Number of positions that are exactly n moves from the starting position in the Pyraminx puzzle.

1, 8, 48, 288, 1728, 9896, 51808, 220111, 480467, 166276, 2457, 32 (list; graph; refs; listen; history; text; internal format)

OFFSET 0,2

COMMENTS This is the number of positions that can be reached in n moves from the start, but which cannot be reached in fewer than n moves.
A puzzle in the Rubik cube family. The total number of distinct positions is 933120. The trivial turns of the tips are ignored.
If tips are included the total number of positions is 933120 * 3^4 = 75582720.

REFERENCES Computed by John Francis and Louis Robichaud.
D. R. Hofstadter, Metamagical Themas, Basic Books, NY, 1985, p. 358.

Fig 7.6. Sequence of positions at depth n
(Taken from oeis.org)

VIII. CONCLUSION

Breadth First Search and Hashing could also be used as an alternative to find God's Number in a twisty puzzle. Although my implementation didn't quite give the exact answer, it was just close enough that it's safe to say the method is proven to be

successful. The error on the final answer is not to be looked on as a failure, but rather a room for improvement in the future. It could finally be concluded that the actual God's Number for the Pyraminx is exactly 15.

IV. ACKNOWLEDGMENT

I would like to thank first of all, to God for giving me all the ability and chance to finish this paper. I would also like to thank Mrs. Ulfa Nur Maulidevi, S.T, M.Sc as discrete mathematics lecturer for the knowledge shared in class throughout this semester and the guidance as I am writing my first paper. A special appreciation for all puzzle designers especially Uwe Mèffert for their work and innovation in creating puzzle that's also worth as variable study material. Appreciation also extends to the community of twisty puzzle community For their passion and dedication, especially in the interest of studying such puzzle's which shared so insight for me. In addition, I would also thank my family, friends, and everyone providing support while writing this paper.

REFERENCES

[1] Denes Ferenc., "Pyraminx," Ruwix. Available: https://ruwix.com/twisty-puzzles/pyraminx-triangle-rubiks-cube/. [Accessed: December 8, 2023].
[2] Herbert Kociemba et al., "God's Number is 20," cube20. Available: https://cube20.org/. [Accessed: December 8, 2023].
[3] K. H. Rosen, "Discrete Mathematics and Its Applications," 8th ed. New York, NY: McGraw Hill, 2019.
[4] R. Johnsonbaugh, "Discrete Mathematics," 8th ed. Upper Saddle River, NJ: Prentice Hall, 2018.
[5] J. Wengrow, "A Common-Sense Guide to Data Structures and Algorithms," New York.
[6] Zachary Stillman, Bowen Shan (2016). Group Theory and the Pyraminx. Cornell University.
[7] https://github.com/Farhannr28/God-s-Number-for-Pyraminx.
[8] Neil Sloane., "A079744," OEIS Foundation Inc. Available: https://oeis.org/A079744. [Accessed: December 11, 2023].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 11 Desember 2023

Farhan Nafis Rayhan - 13522037