# Algoritma *Depth First Search* dan *Brute Force* untuk menyelesaikan Rubik Cube 3x3

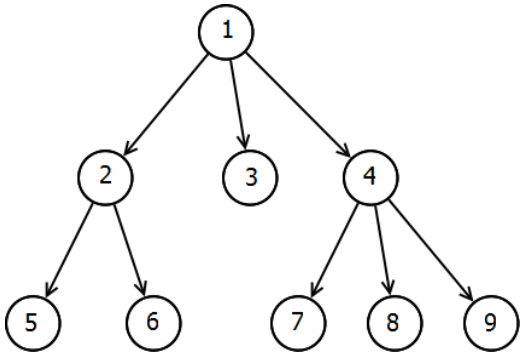
Jofiandy Leonata Pratama, 13517135
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
*13517135@std.stei.itb.ac.id*

***Abstract*— Terdapat berbagai jenis algoritma penyelesaian untuk suatu permasalahan komputasi. Salah satu algoritma solusi yang paling sering digunakan adalah algoritma *brute force*. Namun sering kali algoritma *brute force* ini kurang mangkus dikarenakan lamanya waktu komputasi ketika sudah menyelesaikan permasalahan dengan banyak percobaan (n). Ini menyebabkan munculnya banyak algoritma penyelesaian lainnya yang lebih mangkus untuk menyelesaikan suatu permasalahan komputasi. Salah satunya adalah algoritma *DFS (Depth First Search)*. Algoritma *DFS* menggunakan tipe susunan data berbentuk *tree graph* dan akan mencari atau mengeksplorasi solusi yang memenuhi. Algoritma *DFS* ini sering kali digunakan untuk menyelesaikan permasalahan yang berkembang secara dinamik. Makalah ini akan menjelaskan mengenai kinerja algoritma penyelesaian *Depth First Search* dan penerapannya untuk menyelesaikan Rubik Cube 3x3**

**Kata Kunci:** ***algoritma brute force, algoritma DFS, implementasi DFS, Rubik Cube Problem***

## I. Pendahuluan

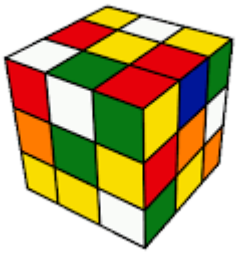
Suatu data dapat di bentuk menjadi berbagai macam struktur data. Struktur data menjelaskan bagaimana data disusun sedemikian sehingga dapat diselesaikan oleh suatu algoritma. Terdapat beberapa jenis struktur data yang sering didengan seperti *array*, *graph*, *stack* dan lain-lain. Struktur data yang sering digunakan pada beberapa permasalah atau kasus ini merupakan algoritma *Depth First Search.* Struktur data pada algoritma *DFS* ini disusun dalam bentuk pohon atau *tree*. Struktur ini membentuk data sehingga terdapat simpul dan anak simpul kiri dan kanan yang menyebabkan penelusuran solusi menjadi lebih mudah.




Gambar 1.1 Struktur Data Tree

Algoritma DFS merupakan salah satu algoritma penyelesaian permasalahan komputasi yang sering digunakan di masa sekarang. Algoritma DFS merupakan algoritma yang menggunakan struktur data *graph tree* sehingga penyelesaiannya dapat dilakukan secara sistematis. Algoritma ini akan menulusri node-node yang terbentuk dari struktur data yang berbentuk *graph tree.* Algoritma ini lalu akan mencoba kemungkinan solusi yang sudah dipertimbangkan sehingga akan menghemat waktu pencarian solusi tersebut.

Menyelesaikan Rubik Cube merupakan salah satu permasalahan yang implementasi dapat melakukan algoritma DFS. Permaslahan Rubik Cube ini menyangkut bagaimana menyelesaikan Rubik Cube sehingga semua warna berada pada sisi yang sama. Permasalahan yang diselesaikan dengan DFS ini dibantu dengan metode-metode menyelesaikan Rubik sehingga apabila ditemukan suatu pola maka akan digunakan solusi dari pola tersebut. Hal ini berarti terdapat pola-pola yang perlu ditempuh agar dicapai Rubik yang selesai.



Gambar 1.2 Permasalahan Rubik Cube

Pada makalah ini, kita akan menjelaskan bagaimana data disusun dalam beberapa bentuk struktur data, bagaimana algoritma penyelesaian dari *DFS* dan juga bagaimana pendekatan permasalahan Rubik Cube dengan beberapa algoritma penyelesaiaan seperti algoritma *brute force* dan algoritma *DFS.*

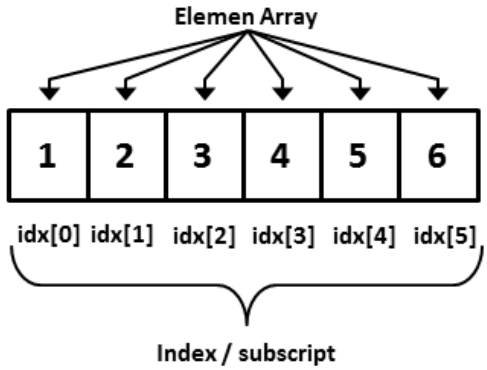
# II. Landasan Teori

## 1. Struktur Data

Struktur data digunakan untuk menyimpan dan merepresentasikan bagaimana data disimpan dan diolah. Pada bagian ini akan dibahas beberapa jenis struktur data seperti *Array*, *Graph* dan *Tree*

### *A. Array*

Struktur data *Array* merupakan struktur data yang paling umum ditemukan ketika menyimpan suatu data. *Array* menyimpan elemen-elemen data yang betipe data yang sama dan tersimpan secara sekuensial. Hal ini berarti *array* sudah terdefinisikan besarnya dan memiliki *index* dan *value*. *Index* mendefinisikan lokasi dari data tersebut disimpan dalam *array* dan *value* mendefinisikan nilai yang disimpan pada lokasi dari data tersebut.




Gambar 2.1 Struktur Data *Array*

*Array* dapat didefinisikan menjadi satu dimensi maupun dua dimensi.

1. *Array* satu dimensi

   *Array* satu dimensi seperti *array* yang umum ditemukan. *Array* ini tersusun dalam satu baris dan memiliki tipe data yang sama

   Bentuk umum dari *Array* satu dimensi adalah:

   NamaArray[n] = {el0,el1,el2,…,n}
   dimana n = jumlah elemen

2. *Array* dua dimensi

   *Array* dua dimensi digambarkan sebagai sebuah matriks. *Array* dua dimensi ini merupakan perluasan dari *Array* satu dimensi yang berarti tiap kolom elemen dari *array* tersebut memiliki *array* sendiri juga. Sehingga terbentuk baris dari setiap kolom elemen tesebut.

| | Column 0 | Column 1 | Column 2 |
|---|---|---|---|
| Row 0 | x[0][0] | x[0][1] | x[0][2] |
| Row 1 | x[1][0] | x[1][1] | x[1][2] |
| Row 2 | x[2][0] | x[2][1] | x[2][2] |

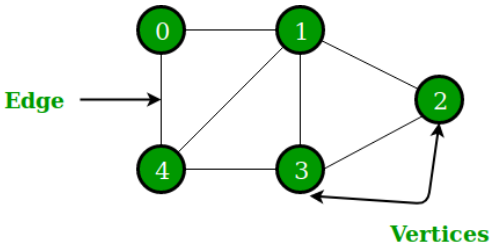
Gambar 2.2 Struktur Data Matriks

Bentuk umum dari Matriks adalah:

NamaArray[m][n] = { {a,b,c}, {d,e,f}}
dengan m dan n merupakan jumlah elemen

### *B. Graph*

Struktur data *Graph* merupakan salah satu struktur data yang sering digunakan ketika suatu data atau objek memiliki hubungan dengan data lainnya. *Graph* biasanya direpresentasikan dalam dua dimensi dengan garis (sisi). Hal ini menyebabkan representasi graph menjadi objek dengan *node* (*vertices)* dan *edge* (sisi).

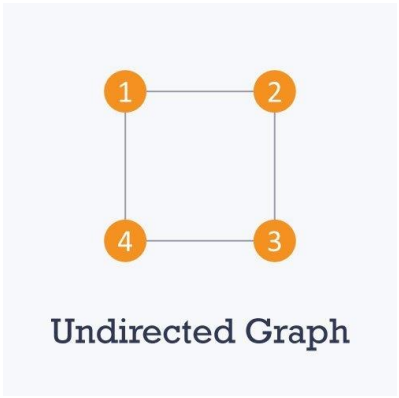



Gambar 2.3 Struktur Data Graph

Berikut merupakan jenis *Graph* berdasarkan arahnya:

1. *Undirected Graph*

   *Undirected Graph* merupakan *graph* yang tidak memiliki arah. Hal ini menyebabkan antar 2 node hanya memiliki relasi tetangga namun tidak diketahui arah dari node mana menuju kemana.

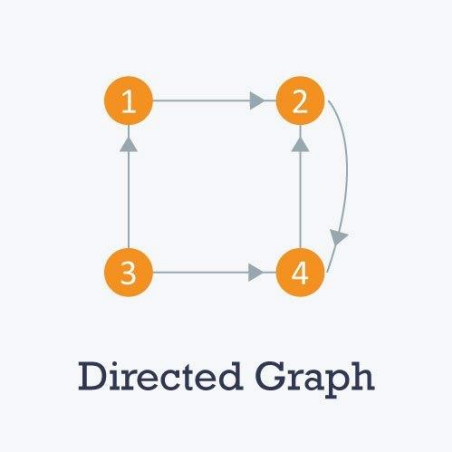



Gambar 2.4 Graph tidak berarah

2. *Directed Graph*

   *Directed Graph* merupakan graph yang miliki arah. *Directed Graph* ini mendefinisikan bagaimana beberapa node saling terhubung dan

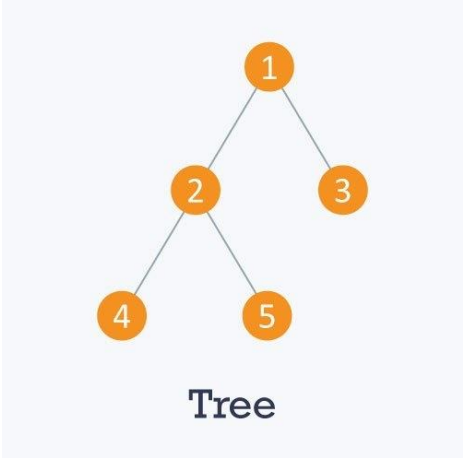
arah yang ditempuh ketika terbentuk lintasan dari graph tersebut.




Gambar 2.5 Graph berarah

## C. *Tree*

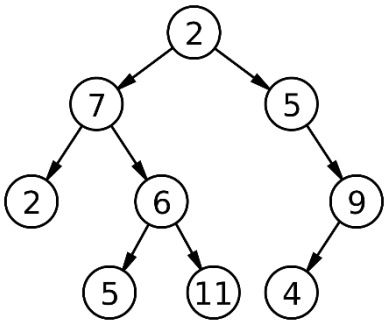
*Tree* merupakan salah satu bentuk undirected graph yang memiliki direpresentasikan dengan parent node dan child node. *Tree* merupakan *acyclic* graf dan setiap node di *tree* hanya terdiri atas satu *parent* node.




Gambar 2.6 Graph bentuk Tree

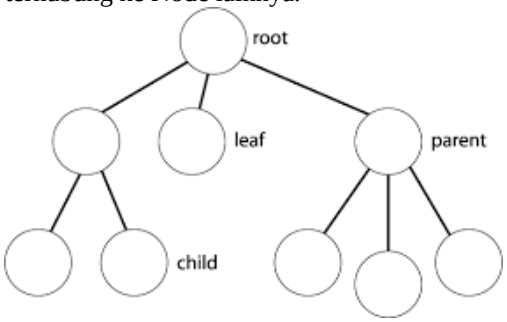
Pada struktur data *Tree*, terdapat beberapa jenis *Tree* berdasarkan jumlah node dari suatu *parent*.

1. *Binary Tree*
   Pada *Binary Tree*, tiap *Parent* hanya memiliki 2 buah *child* node, kiri dan kanan.




Gambar 2.7 Binary Tree

2. *General Tree*
   Pada *General Tree*, tiap *Parent* dapat memiliki 0 atau lebih dari 1 Child Node. Dengan data struktur ini, tidak terbatasi berapa banyak Node yang terhubung ke Node lainnya.




Gambar 2.8 General Tree

Pada permasalahan ini, kita menggunakan Struktur Data berupa *General Tree*. Dalam penyelesaian rubiks, terdapat beberapa metode solusi yang diimplementasikan ketika ditemukan sebuah pola pada rubiks tersebut. Sehingga representasi struktur data yang paling baik untuk permasalahan ini adalah *General Tree*.

## 2. Algoritma Penyelesaian

Algoritma penyelesaian yang paling umum digunakan adalah algoritma *brute force*. Namun terkadang algoritma *brute force* tidak menghasilkan solusi yang mangkus. Apabila menemui jumlah data yang sangat besar, algoritma *brute force* membutuhkan waktu yang sangat lama untuk menemukan solusi tersebut. Hal ini menyebabkan munculnya algoritma-algoritma lainnya yang lebih mangkus ketika ingin menyelesaikan suatu bentuk permasalahan. Salah satunya adalah *Depth First Search* (DFS)

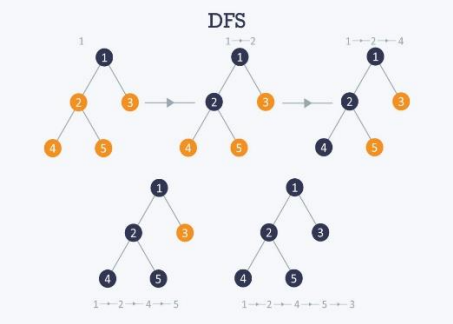
### *A. Brute force*

*Brute force* merupakan algoritma yang umum diimplementasikan dalam menyelesaikan suatu permasalahan. Kelebihan dari algoritma *brute force* adalah dapatnya ditemukan solusi dari permasalahan yang ada dalam bentuk struktur data apapun.

Namun, Algoritma *brute force* tidak menggunakan pendekatan singkat dalam mencari solusinya melainkan melakukan rekursif terhadap semua kemungkinan yang ada. Hal ini mempengaruhi performa dari algoritma *brute force* terutama pada permasalahan waktu ketika menemukan jumlah data yang sangat besat atau langkah rekursif yang sangat besar.

B. *Depth First Search (DFS)*

Algoritma *Depth First Search (DFS)* merupakan salah satu metode penyelesaian permasalahan untuk struktur data Tree. Algoritma DFS mengimplementasikan konsep rekursif dengan bantuan *backtracking*. Konsep DFS adalah algoritma ini akan menelusuri lintasan pada suatu node hingga tidak ditemukan jalur lain dari lintasan tersebut dan akan melakukan *backtracking* untuk menelusuri node lain untuk menemukan jalur lain. Proses ini akan direkursif sampai ditemukan solusi yang memenuhi atau yang dicari.



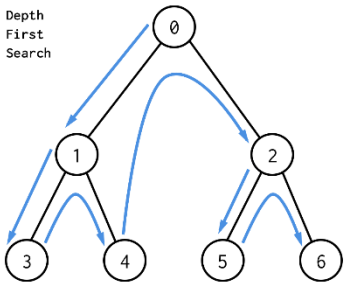

Gambar 3.1 Algoritma DFS

Algoritma penyelesaian secara DFS mengimplementasikan 2 metode dalam pencarian solusinya:

1. Traversal

   Algoritma DFS yang menggunakan metode Traversal akan menggunakan bantuan rekursif. Node akan ditelusuri satu demi satu dan akan ditandai dengan *boolean* untuk menunjukkan bahwa node itu sudah terkunjungi.

   Dengan metode ini, akan dituliskan satu demi satu tetangga atau *adjacent* dari suatu node dan akan ditelusuri semua node yang belum dikunjungi sampai ditemukan solusi

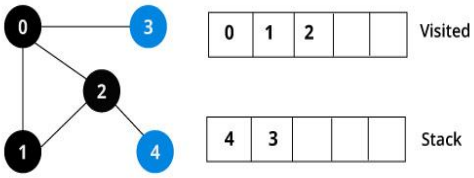



Gambar 3.2 DFS dengan Traversal

2. Stack

   Algoritma DFS yang menggunakan metode *Stack* akan menggunakan bantuan berupa *Stack*. Metode ini diimplementasikan dengan cara:

   a. Pilih node awal yang akan dikunjungi dan akan push semua *adjacent* node ke dalam *stack*.
   b. *Pop* node yang akan dikunjugin selanjutnya dan push semua *adjacent* node yang di-*pop* ini.
   c. Node yang sudah di-*pop* atau dikunjungi akan ditandai dengan *boolean* agar tidak dikunjungi dua kali
   d. Lakukan proses ini terus-menerus hingga stacknya tidak berisi lagi atau ditemukan solusi yang memenuhi

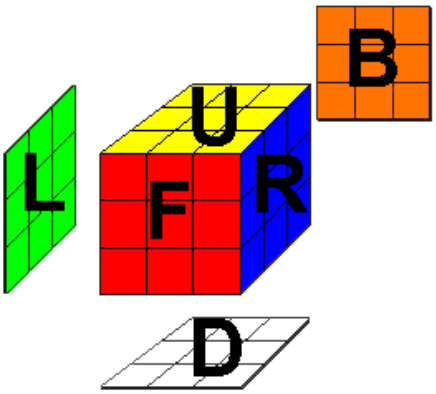



Gambar 3.3 DFS dengan Stack

Pada permasalahan ini, akan digunakan algoritma penyelesaian DFS (*Depth First Search*) dengan metode traversal. DFS digunakan untuk menyelesaikan permasalahan Rubik dengan struktur data *Tree* dikarenakan struktur data ini mempermudah untuk bisa menentukan langkah yang tepat ketika menemukan sebuah pola dalam permasalah Rubik ini. Dengan struktur data ini, algoritma yang paling cocok untuk menyelesaikannya adalah DFS dikarenakan *tree* akan dtelusuri dan dapat melakukan *backtracking* sehingga lebih cepat ditemukan solusi yang memenuhi tanpa harus menelusuri semua kemungkinan yang ada.

## III. Penyelesaian Permasalahan Rubik

Rubik merupakan suatu permainan yang dimainkan dengan cara memutar sisi-sisi pada Rubik sehingga didapatkan semua sisi memiliki warna yang sama. Pada permasalahan ini, Rubik yang digunakan merupakan Rubik 3x3. Penyelesaian permasalahan ini dibantu dengan Teknik penyelesaian Rubik 3x3 yang berlandaskan pola yang dimiliki pada Rubik tersebut. Berikut merupakan istilah sisi dari sebuah rubik



Gambar 3.1 Sisi pada Rubik

Pendekatan yang dilakukan untuk menyelesaikan permasalahan rubik ini adalah dengan membentuk pola-pola pada tiap bagian dari rubik. Metode menyelesaikan rubik ini dilakukan dengan:

1. Membuat "*White*" *Cross* pada Sisi D
2. Membuat *"White" Corner* pada Sisi D
3. Membuat Bagian Tengah (Baris 2) pada sisi F,,L,R dan B
4. Membuat *Yellow Face* pada sisi U
5. *Finishing* yang bagian yang tersisa

Algoritma pendekatan yang akan dibahas untuk menyelesaikan permasalahan ini melalui dua metode yaitu algoritma *brute force* dan algoritma DFS.

A. Algoritma *brute force*

Algoritma *brute force* yang digunakan untuk menyelesaikan permasalahan rubik ini akan mencoba semua kemungkinan yang dapat dilakukan untuk mendapatkan pola-pola yang diinginkan berdasarkan metode yang diterapkan.

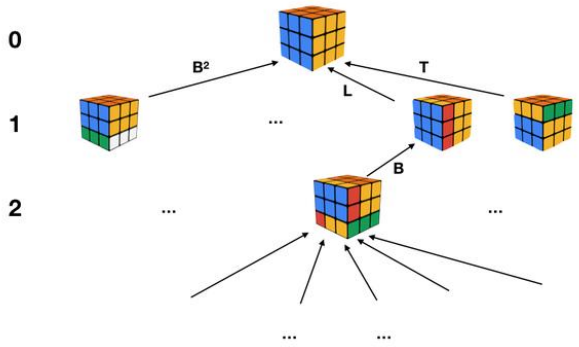
Berdasarkan hitungan dan jumlah kemungkinan yang ada, algoritma *brute force* sangat tidak dianjurkan pada permasalahan ini. Jumlah kemungkinan yang mungkin dengan algoritma ini bernilai $4.33\ x\ 10^{19}$ kemungkinan, yang diperkirakan membutuhkan waktu sekitar $2.44\ x\ 10^{13}$ tahun untuk menyelesaikannya.

Dari perhitungan jumlah kemungkinan dan waktu yang diperlukan untuk menyelesaikan permasalahan rubik 3x3, algoritma *brute force* sangat tidak dianjurkan untuk diimplementasikan untuk mencari solusinya

.

B. Algoritma *Depth First Search* (DFS)

Salah satu Algoritma yang paling dianjurkan untuk menyelesaikan permasalahan Rubik adalah algoritma *Depth First Search*. Berdasarkan metode yang kita rancang, struktur data *Tree* merupakan stuktur data yang paling memungkinkan. Hal ini dikarenakan pendekatan permasalahan ini akan dilakukan *step-by-step* untuk mencari suatu pola yang diharapkan. Hal lain yang mendukung algoritma DFS untuk diimplementasikan dalam menyelesaikan permasalahan ini adalah kemampuan DFS untuk memilih jalur yang akan dilalui dan kapan berhenti (ketika memang tidak memungkinkan lagi) dan kemampuan *backtracking* dari DFS sehingga pencarian solusi dapat ditemukan secara efisien.




Dari metode yang penyelesaian yang diajukan, berikut merupakan salah satu dari keseluruhan algoritma penyelesaian dari permaslaahan rubik

```
void DFS_Stage2 (int depth, List<Color>RemainingColors,
RubiksCube parent, List<RubiksCube> tree)
{
        if (depth == 0) // first iteration
        {
            RemainingColors. = new List<Color>();
            RemainingColors.Add(Cube.REDCOLOR);
            RemainingColors.Add(Cube.GREENCOLOR);
          RemainingColors.Add(Cube.ORANGECOLOR);
            RemainingColors.Add(Cube.BLUECOLOR);
        }

        for (int i = 0; i<RemainingColors.Count ; i++)
        {
           RubiksCube tempRC = parent.cloneCube();
           tempRC.turnCubeToFaceRGBOColorWithYell
           owOrWhiteOnTop(RemainingColors[i]);
          SolveWhiteSideCube(tempRC);
           List<Color> newRemainingColors = new
           List<Color>();
           for (int j = 0; j < RemainingColors.Count; j++){
           newRemainingColors.Add(RemainingColors[j])
           ;}
          newRemainingColors.RemoveAt(i);

          if(newRemainingColors.Count == 0){
                tree.Add(tempRC);
                return;
          }
          else
                DFS_Stage2(depth+1, newRemainingColors,
                tempRC,tree);
        }
```

```
        return;
}
```

Algoritma Penyelesaian Rubik

Algoritma Penyelesaian Rubik tersebut menyelesaikan pada metode nomor 3 (bagian tengah pada sisi F,L,R dan B). Pada Algoritma ini diperhatikan bahwa terdapat pemanggilan pada algoritma penyelesaian metode 1 dan 2 (yaitu menyelesaikan bagian *"White"* atau sisi D yang lalu dipakai pada algoritma "SolveWhiteSideCube". Ini merupakan salah satu langkah dalam mengimplementaskan algoritma DFS dimana akan ditemukan suatu pola tertentu sebelum dapat melanjutkan ke lintasan penyelesaian selanjutnya.

Dengan menggunakan Algoritma DFS ini, di dapatkan bahwa jumlah langkah yang diperlukan untuk menyelesaikan permasalahan Rubik ini berjumlah sekitar 140-160 langkah dari rubik dalam kondisi berantakan hingga terselesaikan.

## IV. Analisa

Melalui dua metode penyelesaian terhadap permasalahan rubik, dilihat dari ruang sample dan waktu yang diperlukan untuk menyelesaikan masalah, disimpulkan bahwa algoritma *Depth First Search* (DFS) lebih mangkus dalam menyelesaikan permasalahannya.

Algoritma DFS ini lebih diunggulkan dikarenakan beberapa hal:

1. Stuktur data yang disusun untuk menggunakan algoritma DFS ini. Struktur Data *Tree* sangat berperan dalam menyelesaikan permasalahan ini karena melalui *Tree*, metode penyelesaiannya dapat mengetahui langkah mana yang harus ditempuh agar mendapatkan pola atau jalur yang diharapkan.
2. Kemampuan *backtracking* yang membantu menghemat ruang *sample*. Apabila memang tidak dapat ditemukan solusi melalui suatu jalur, maka algoritma DFS dapat melakukan *backtracking* sehingga tidak perlu melakukan pengecekan terhadap semua solusi. Kemampuan ini menyingkat waktu yang diperlukan untuk menemukan solusi yang diharapkan
3. Algoritma DFS juga membagi persoalan menjadi beberapa bagian berdasarkan kemampuan *backtracking*-nya dan algoritma pencariannya. Hal ini menyebabkan DFS mengetahui jalur mana yang harus ditempuh agar menempuh lintasan yang dapat mencapai solusi sehingga efisien pada waktu dan ruang.

Beberapa kemampuan ini menyebabkan Algoritma DFS unggul pada permasalahan ini dibandingkan dengan Algoritma *brute force*.

## V. Kesimpulan

Berdasarkan analisis yang telah dilakukan, dapat disimpulkan bahwa algoritma DFS unggul dalam menyelesaikan permasalahan rubik ini. Hal ini dikarenakan kemampuan DFS dalam *backtracking* dan struktur data *Tree* yang digunakan.

Algoritma *brute force* kurang dianjurkan pada permasalahan ini dikarenakan ruang *sample* percobaan yang perlu dilalui terlalu besar sehingga menyebabkan waktu yang sangat lama dalam pencarian solusi.

## Ucapan Terima Kasih

Pertama-tama, penulis mengucapkan syukur kepada Tuhan Yanng Maha Esa karena karunia-Nya, penulis dapat menyelesaikan makalah berjudul "Algoritma *Depth First Search* dan *Brute Force* untuk menyelesaikan Rubik Cube 3x3". Ucapan terima kasih juga diberikan kepada Bapak Dr. Ir. Rinaldi Munir selaku dosen mata kuliah IF2211 Strategi Algoritma untuk K-3 yang telah membimbing dan memberikan materi mengenai algoritma penyelesaian suatu permasalahan selama proses pembelajaran mata kuliah Strategi Algoritma.

## Referensi

[1] Munir, Rinaldi. Diktat Kuliah Strategi Algoritma. Unpublished

[2] Harpreet Kaur. Algorithms for solving Rubik's Cube

(http://www.diva-portal.org/smash/get/diva2:816583/FULLTEXT01.pdf)

[3] Lawrence L. Larmore, DFS and BFS Algorithms using Stacks and Queues

(http://www.egr.unlv.edu/~larmore/Courses/CSC477/bfsDfs.pdf)

# 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, 26 April 2019

Jofiandy Leonata Pratama