Optimasi Penyelesaian *Rubik's Cube*: Penerapan Algoritma *Iterative Deepening A** pada Metode Kociemba untuk Menemukan Solusi Dalam 30 Langkah

Albertus Christian Poandy - 13523077[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]achrisp05@gmail.com, 13523077@std.stei.itb.ac.id

***Abstrak*—Makalah ini membahas penerapan algoritma *Iterative Deepening A** (IDA*) pada metode penyelesaian *Rubik's Cube* oleh Herbert Kociemba, yaitu *two-phase-algorithm*, dengan langkah maksimal 30. Metode ini terbagi menjadi dua fase: *phase 1*, untuk mencapai suatu subset G1, dan *phase 2*, untuk menyelesaikan kubus rubik sepenuhnya. Dengan menggunakan *pruning table* sebagai *heuristic function*, algoritma IDA* mampu membatasi ruang pencarian, sehingga proses penemuan solusi menjadi lebih efisien.**

***Kata kunci*— Heuristic function, Iterative Deepening A*, Rubik's Cube, two-phase-algorithm.**

I. PENDAHULUAN

Rubik's cube atau kubus rubik adalah sebuah permainan mekanis klasik yang berbentuk kubus 6 sisi, dengan masing-masing sisi terdiri dari 3x3 kotak kecil yang berwarna. Tujuan utama dari permainan ini adalah mengumpulkan warna-warna yang sama sehingga tiap sisi kubus akan memiliki kotak-kotak dengan warna yang seragam.

Terdapat banyak metode yang dapat digunakan untuk menyelesaikan permainan ini. Salah satu metode populer yang sering digunakan adalah *Friedrich's Rubik's Cube Method*, yang melibatkan 4 tahap, CFOP, yaitu *Cross*, F2L (*First Two Layer*), OLL (*Orient Last Layer*), dan PLL (*Permute Last Layer*). Metode ini digunakan oleh banyak *speedcuber* di dunia, terutama karena metode ini memungkinkan kubus rubik untuk diselesaikan dalam kurang dari 6 detik.

Namun, bagaimanakah cara untuk menyelesaikan sebuah kubus rubik menggunakan program komputer? Salah satu ide yang langsung muncul mungkin adalah menggunakan pendekatan *brute force*. Cukup lakukan segala kemungkinan gerakan yang dapat dilakukan, hingga mencapai *state* kubus rubik yang telah terselesaikan. Namun, sebuah kubus rubik memiliki 43.252.003.274.489.856.000 (43 quintillion) kemungkinan posisi yang berbeda. Waktu komputasi yang dibutuhkan untuk memroses seluruh kemungkinan posisi yang ada adalah sekitar seribu tahun. Tentu saja pendekatan ini bukanlah opsi yang optimal.

Sebuah fakta yang sangat menarik dari sebuah kubus rubik adalah: setiap posisi pada kubus rubik dapat diselesaikan dalam maksimal 20 langkah. Angka ini biasanya disebut dengan *God's Number*. *God's number* ditemukan oleh peneliti dari *Google* menggunakan metode Kociemba, yaitu *two-phase-algorithm*. *Two-phase-algorithm* adalah metode yang dikembangkan oleh Herbert Kociemba untuk menyelesaikan sebuah kubus rubik yang dapat diterapkan pada program komputer secara efisien. Metode ini melibatkan algoritma graf *Iterative Deepening A* (IDA*)* dalam pencarian solusi sebuah kubus rubik untuk mengefisienkan waktu komputasi yang dibutuhkan.

Makalah ini akan membahas bagaimana cara kerja dari *two-phase-algorithm*, tetapi akan berfokus pada peran penting algoritma graf IDA* dalam menurunkan sumber daya komputasi yang diperlukan untuk mencari solusi. Makalah ini terbagi menjadi 5 bagian utama, yaitu: Bagian 1 adalah pendahuluan. Bagian 2 membahas teori graf, yaitu dasar-dasar graf serta istilah/terminologi yang sering digunakan. Bagian 3 akan menjelaskan algoritma graf Iterative Deepening A*. Bagian 4 akan membahas metode Kociemba untuk menyelesaikan kubus rubik menggunakan *two-phase-algorithm*. Terakhir, bagian 5 akan memberikan kesimpulan yang didapatkan.

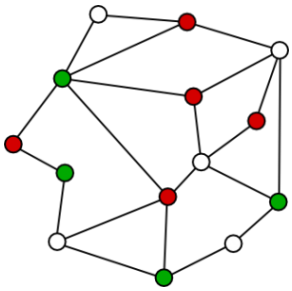
II. GRAPH THEORY

A. Definisi

Graf adalah struktur yang dapat digunakan untuk merepresentasikan objek-objek diskrit dan hubungan antara objek-objek tersebut. Graf terdiri dari dua komponen utama, yaitu simpul (*vertex*) dan sisi (*edge*). Simpul, yang sering juga disebut dengan *node*, adalah objek diskrit yang diwakili dalam graf. Sisi adalah komponen yang menghubungkan sepasang simpul.

Sebuah graf G dapat didefinisikan sebagai $G = (V, E)$, yang dalam hal ini: V adalah himpunan tidak kosong dari simpul-simpul; E adalah himpunan sisi-sisi yang menghubungkan simpul. Perhatikan bahwa himpunan V tidak boleh kosong,

sehingga tidak mungkin ada sebuah graf yang tidak mengandung simpul, tetapi himpunan E boleh kosong, sehingga sebuah graf bisa saja tidak memiliki satu pun sisi.

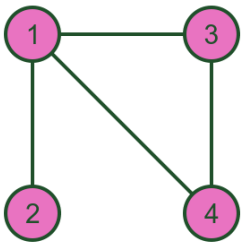


Gambar 1. Ilustrasi Contoh Graf
Sumber: https://www.open.ac.uk/blogs/is/?p=16

B. Terminologi

B.1. Ketetangaan (adjacent)

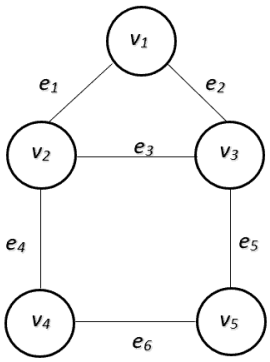
Dua buah simpul yang berbeda dikatakan saling bertetangga (*adjacent*) jika dan hanya jika keduanya terhubung secara langsung. Pada contoh Gambar 2. di bawah, dapat dikatakan bahwa simpul 1 bertetangga (*adjacent*) dengan simpul 2, simpul 4 bertetangga dengan simpul 3, tetapi simpul 2 tidak bertetangga dengan simpul 4.



Gambar 2. Ilustrasi Contoh Graf dengan Simpul Bertetangga
Sumber: https://graphicmaths.com/computer-science/graph-theory/graphs/

B.2. Bersisian (incidency)

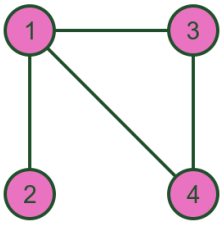
Suatu simpul dan dan suatu sisi dikatakan bersisian (*incident*) jika dan hanya jika simpul tersebut terhubung secara langsung dengan sisi tersebut. Untuk suatu sisi $e = (n_i, n_j)$, dikatakan bahwa sisi e bersisian dengan dengan simpul n_i, atau sisi e bersisian dengan simpul n_j. Pada contoh Gambar 3. di bawah, sisi e_1 dikatakan bersisian dengan simpul v_1, sisi e_2 dikatakan bersisian dengan simpul v_3, tetapi sisi e_1 tidak bersisian dengan simpul v_3.



Gambar 3. Ilustrasi Contoh Graf dengan Simpul dan Sisi bersisian
Sumber: https://graphicmaths.com/computer-science/graph-theory/graphs/

B.3. Derajat (degree)

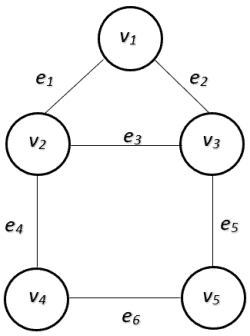
Derajat (*degree*) dari suatu simpul adalah jumlah sisi yang bersisian dengan simpul tersebut. Derajat dari suatu simpul v dinotasikan dengan $d(v)$. Pada contoh Gambar 4. di bawah, derajat dari simpul 1 adalah 3, $d(1) = 3$, derajat dari simpul 2 adalah 1, $d(2) = 1$, dan derajat dari simpul 3 sama dengan derajat dari simpul 4 yaitu 2, $d(3) = d(4) = 2$.



Gambar 4. Ilustrasi Contoh Graf
Sumber: https://graphicmaths.com/computer-science/graph-theory/graphs/

B.4. Lintasan (path)

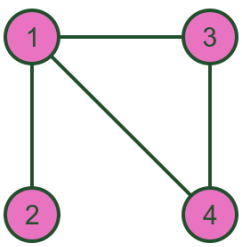
Lintasan (*path*) dari simpul awal v_0 menuju simpul v_n adalah barisan selang-seling antara simpul-simpul dan sisi-sisi dengan bantuk $v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n$, sedemikian sehingga e_1 bersisian dengan v_0 dan v_1, e_2 bersisian dengan v_1 dan v_2, ..., e_n bersisian dengan v_{n-1} dan v_n, dengan n adalah panjang dari lintasan. Panjang dari sebuah lintasan adalah jumlah sisi dalam lintasan tersebut. Pada Gambar 5. di bawah, lintasan v_1, v_3, v_2, v_4 memiliki panjang 3.



Gambar 5. Ilustrasi Contoh Lintasan Dalam Graf
Sumber: https://graphicmaths.com/computer-science/graph-theory/graphs/

B.5. Siklus (cycle) atau Sirkuit (circuit)

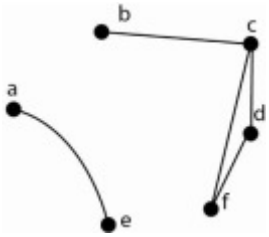
Siklus (*cycle*) atau sirkuit (*circuit*) adalah lintasan dengan simpul awal dan simpul tujuan yang sama. Dalam kata lain, siklus atau sirkuit adalah lintasan yang berawal dan berakhir pada simpul yang sama. Pada Gambar 6. di bawah, lintasan $1, 3, 4, 1$ adalah sebuah siklus.



Gambar 6. Ilustrasi Contoh Graf dengan Sebuah Sirkuit
Sumber: https://graphicmaths.com/computer-science/graph-theory/graphs/

B.6. Keterhubungan (connected)

Dua buah simpul berbeda v_i dan v_j dikatakan saling terhubung (*connected*) jika dan hanya jika terdapat lintasan dari v_i ke v_j. Sebuah graf G disebut graf terhubung jika untuk setiap pasan simpul v_i dan v_j yang berada pada graf, terdapat lintasan dari v_i ke v_j. Pada contoh Gambar 7. di bawah, simpul a dan e dikatakan terhubung, simpul b dan c dikatakan terhubung, tetapi simpul a dan b tidak terhubung.



Gambar 7. Ilustrasi Contoh Graf
Sumber: https://www.sciencedirect.com/topics/computer-science/connected-graph

III. ITERATIVE DEEPENING A* (IDA*)

A. Definisi

Algoritma *Iterative Deepening A** (IDA*) adalah algoritma pencarian lintasan/rute terpendek pada graf yang menggabungkan dua algoritma pencarian dalam graf, yaitu algoritma *Iterative Deepening Search* (IDS) dan algoritma *A**. Algoritma ini memiliki keunggulan dalam penggunaan memori, karena algoritma ini menggunakan memori yang proporsional dengan *depth* atau kedalaman pencarian. Algoritma ini didesain untuk mencari lintasan terpendek pada graf dengan mempertahankan efisiensi memori yang digunakan, yang membuatnya cocok untuk pencarian pada ruang yang besar, di mana penyimpanan informasi terkait simpul yang telah dikunjungi tidak mungkin dilakukan.

B. Iterative Deepening Search (IDS) Algorithm

Algoritma *Iterative Deepening Search* adalah algoritma pencarian dalam graf yang menggabungkan efisiensi memori dari *Depth First Search* (DFS) dan kecepatan pencarian *Bread First Search* (BFS) untuk simpul yang berada dekat dengan simpul sumber. Seperti DFS, algoritma ini menggunakan memori yang proposional dengan tingkat kedalaman pencarian, bukan ukuran dari graf yang ada.

Algoritma ini akan melakukan algoritma DFS secara berulang-ulang sambil meningkatkan batas kedalaman pada tiap iterasinya. Hal ini berarti simpul sumber (simpul awal) dan simpul-simpul yang dekat dengan sumber akan diproses berulang-ulang, serta simpul yang berada pada ujung graf hanya akan diproses satu kali. Pemrosesan yang berulang ini dapat menjadi proses komputasi yang cukup mahal pada faktor waktu. Namun, pada banyak kasus, hal tersebut tidak akan berpengaruh terlalu signifikan, terutama jika simpul-simpul pada graf berada lebih banyak pada bagian ujung (contohnya pada tree).

```python
def IDS(source, target, max_depth):
    for depth in range(max_depth):
        if (DLS(source, target, depth)):
            return True
    return False

def DLS(source, target, depth):
    if source == target:
        return True

    if depth == 0:
        return False

    for neighbor in neighbors(source):
        if (DLS(neighbor, target, depth - 1)):
            return True

    return False
```

Gambar 8. Potongan Kode Implementasi IDS
Sumber: arsip pengguna

C. A* (A-Star) Algorithm

Algoritma *A** (dibaca A-Star) adalah salah satu algoritma pencarian lintasan dalam graf yang paling populer dan paling banyak digunakan karena kecepatan dan efisiensi waktu yang ditawarkannya. Algoritma ini mencari lintasan terpendek dengan memanfaatkan suatu nilai estimasi pada tiap simpul yang disebut *heuristic function*.

Pada tiap tahap pencarian, algoritma *A** akan mengevaluasi simpul-simpul yang dapat dikunjungi selanjutnya melalui fungsi biaya:

$$f(n) = g(n) + h(n) \quad (1)$$

Di mana:

- $g(n)$: Biaya yang dibutuhkan untuk mencapai simpul n ini dari simpul awal
- $h(n)$: Biaya estimasi untuk mencapai simpul target dari simpul n
- $f(n)$: Total biaya untuk lintasan yang melalui simpul n

Algoritma ini akan memprioritaskan simpul dengan nilai $f(n)$ yang paling rendah. Dalam kata lain, algoritma ini akan fokus pada lintasan yang diestimasikan akan menghasilkan biaya paling rendah.

Algoritma ini menawarkan efisiensi waktu, dengan meminimalisir jumlah simpul yang akan dijelajahi, dan memprioritaskan simpul yang lebih menjanjikan. Namun, algoritma ini dapat menjadi mahal pada aspek memori karena diperlukannya penyimpanan terhadap simpul-simpul yang telah dijelajahi dan simpul-simpul yang dapat dijelajahi pada tahap selanjutnya.

D. Mengombinasikan IDS dan A*

Algoritma IDA* mampu mengatasi keterbatasan memori dari algoritma *A** dengan melakukan penjelajahan secara iteratif seperti pada algoritma IDS. Algoritma ini juga tetap mempertahankan optimalitas dari *A** dengan menggunakan bantuan estimasi *heuristic function*. Perbedaan proses penjelajahan pada algoritma IDA* dan IDS adalah maksimum kedalaman pencarian pada algoritma IDA* dicapai ketika nilai $f(n) = g(n) + h(n)$ melebihi *threshold*.

Langkah-langkah pencarian yang dilakukan oleh algoritma IDA* adalah:

1. Inisialisasi *threshold*, yaitu nilai *heuristic estimate* $h(n)$ dari simpul awal.
2. Lakukan DFS: Jelajahi graf dengan algoritma DFS. Maksimum *depth* pada proses penjelajahan dicapai ketika $f(n) > threshold$. Jika *maximum depth* tercapa, lakukan *pruning* pada simpul tersebut.
3. Jika simpul target ditemukan pada proses DFS, maka lintasan telah ditemukan.
4. Jika simpul target tidak ditemukan, *update* nilai *threshold* dengan nilai $f(n)$ minimum dari simpul-simpul yang telah di-*prune* sebelumnya.
5. Ulangi proses hingga mencapai simpul target atau hingga seluruh kemungkinan jalur telah dicoba (simpul target tidak ditemukan).

```
def IDA(start, goal):
    threshold = heuristic(start)
    while True:
        result = DLS(start, goal, threshold, 0)
        if (result == "FOUND"):
            return True
        if (result == "NOT FOUND"):
            return False
        threshold = result
```

Gambar 9. Potongan Kode Implementasi IDA* (1)
Sumber: arsip pengguna

```
def DLS(node, goal, threshold, g):
    f = g + heuristic(node)
    if node == goal:
        return "FOUND"
    if f > threshold:
        return f

    minimum = float('inf')
    for neighbor in neighbors(node):
        result = DLS(neighbor, goal, threshold, g + cost(node, neighbor))
        if result == "FOUND":
            return "FOUND"
        minimum = min(minimum, result)

    if minimum == float('inf'):
        return "NOT FOUND"

    return minimum
```

Gambar 10. Potongan Kode Implementasi IDA* (2)
Sumber: arsip pengguna

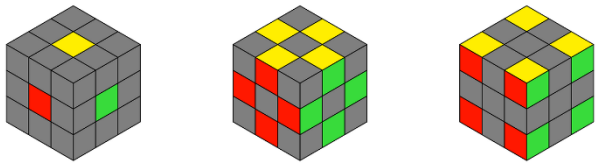
IV. TWO-PHASE-ALGORITHM

A. Definisi

Two-phase-algorithm yang diperkenalkan oleh Herbert Kociemba adalah metode untuk menyelesaikan sebuah kubus rubik secara efisien dengan memisahkan proses penyelesaian ke dalam dua fase yang berbeda. Fase pertama adalah proses mentransformasikan kubus rubik yang teracak menjadi sebuah *subset* yang disebut G1. Fase yang ke-dua adalah proses menyelesaikan kubus rubik dari *subset* G1. *Two-phase-algorithm* didesain untuk mencari solusi yang sangat dekat dengan solusi optimal, yang dapat menyelesaikan kubus rubik dalam 20 langkah atau bahkan kurang dari 20 langkah.

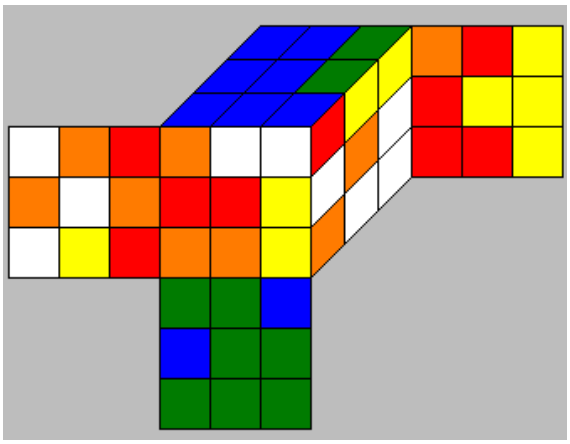
Sebuah kubus rubik mempunyai 6 sisi, yaitu sisi U (*Up*), D (*Down*), L (*Left*), R (*Right*), F (*Front*), dan B (*Back*). Notasi gerakan pada sebuah kubus rubik juga menggunakan penamaan yang sama, di mana notasi U berarti memutar sisi atas kubus rubik 90° searah jarum jam, notasi U' berarti memutar sisi atas kubus rubik 90° berlawanan arah jarum jam, dan notasi U2 berarti memutar sisi atas kubus rubik sejauh 180°. Notasi ini juga berlaku pada seluruh sisi kubus.

Sebuah kubus rubik memiliki 3 bagian, yaitu *corner*, *edge*, dan *centre*. *Corner* adalah bagian dari kubus rubik yang berada di sudut, yang terdiri dari tiga warna berbeda. Terdapat 8 buah *corner* pada sebuah kubus rubik. *Edge* adalah bagian dari kubus rubik yang berada di antara 2 *corner* berbeda, yang terdiri dari dua warna berbeda. Terdapat 12 buah *edge* pada sebuah kubus rubik. *Centre* adalah kotak paling tengah dari sebuah sisi pada sebuah kubus rubik yang hanya memiliki satu warna. Terdapat 6 buah *centre* pada sebuah kubus rubik.



Gambar 11. *Corner*, *edge*, dan *centre* dari sebuah kubus rubik
Sumber: https://medium.com/swlh/how-i-learned-to-solve-the-rubiks-cube-in-30-seconds-afff9292b030

Sebuah *subset* G1 = <U, D, R2, L2, F2, B2> adalah kondisi di mana sebuah kubus rubik dapat diselesaikan hanya dengan 6 jenis gerakan, yaitu U, D, R2, L2, F2, dan B2. *Subset* ini adalah *subset* yang spesial, karena orientasi dari tiap *corner* dan tiap *edge* pada subset ini tidak dapat berubah, serta 4 *edges* yang berada di antara sisi U dan sisi D (UD-*slice*) akan selalu berada pada *slice* tersebut. Salah satu cara untuk mencapai *subset* ini adalah dengen memulai dari kubus rubik yang telah terselesaikan, dan mengacaknya tanpa menggunakan gerakan R, R', L, L', F, F', B, dan B'.



Gambar 12. Ilustrasi Contoh Subset G1
Sumber: https://kociemba.org/math/twophase.htm

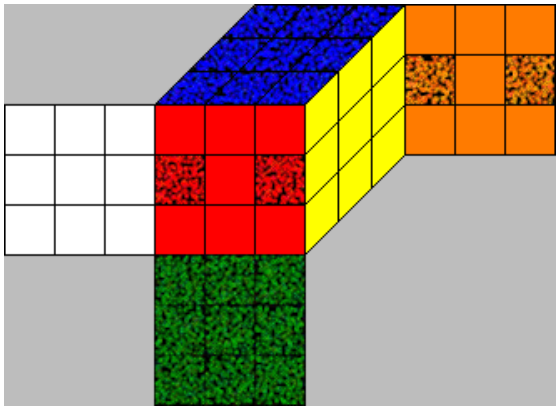
B. Coordinates Representation

Untuk menyelesaikan sebuah kubus rubik menggunakan program komputer, diperlukan sebuah cara untuk

merepresentasikan kondisi dari sebuah kubus rubik. Merepresentasikan sebuah kubus rubik dengan menyimpan posisi dari tiap warna tentu saja tidak optimal, terutama karena efisiensi adalah kunci utama dalam penggunaan *two-phase-algorithm*. Dalam *two-phase-algorithm*, kondisi dari sebuah kubus rubik akan direpresentasikan dalam bentuk koordinat (x, y, z).

B.1. Corner Orientation Coordinate (CO)

Corner Orientation Coordinate (CO) adalah koordinat untuk merepresentasikan orientasi dari delapan *corner* pada sebuah kubus rubik. Orientasi dari sebuah *corner* ditentukan relatif terhadap sebuah referensi. Pada *two-phase-algorithm*, referensi yang digunakan adalah sebagai berikut:



Gambar 13. Referensi untuk Orientasi Kubus Rubik
Sumber: https://kociemba.org/math/cubielevel.htm

Untuk merepresentasikan orientasi dari sebuah *corner*, digunakan tiga angka: 0, berarti *corner* telah memiliki orientasi yang benar (sesuai relatif terhadap referensi); 1, berarti *corner* memiliki orientasi terputar searah jarum jam; 2, berarti *corner* memiliki orientasi terputar berlawanan arah jarum jam. Maka, representasi orientasi dari delapan buah *corner* adalah delapan angka berurutan, dengan tiap digitnya dapat berupa angka 0, 1, atau 2. Bentuk ini adalah bentuk dari sebuah bilangan dalam basis 3, sehingga bentuk ini dapat direpresentasikan dalam bilangan desimal, dengan cara mengonversikan bilangan dengan basis 3 menuju bilangan desimal.

$$CO = \sum_{i=0}^{6} orientasi_i \times 3^i \qquad (2)$$

Maka, total kemungkinan koordinat orientasi *corner* yang berbeda adalah $3^7 = 2187$, dengan tiap koordinat direpresentasikan dengan sebuah angka dari 0 hingga 2186. Perhatikan bahwa orientasi dari *corner* ke-delapan tidak perlu disimpan, karena orientasinya dapat ditentukan oleh orientasi dari 7 *corner* lainnya, di mana:

$$\sum_{i=0}^{7} orientasi_i \equiv 0 (mod\ 3) \qquad (3)$$

B.2. Edge Orientation Coordinate (EO)

Edge Orientation Coordinate (EO) adalah koordinat yang digunakan untuk merepresentasikan orientasi dari dari 12 *edge* yang berada pada sebuah kubus rubik. Cara representasi ini sangat mirip dengan cara representasi CO. Yang membedakannya adalah hanya terdapat dua orientasi dari sebuah *edge*, yaitu: 0, jika *edge* telah terorientasi dengan benar relatif terhadap referensi; 1, jika *edge* terorientasi secara terbalik.

Maka, representasi ini dapat dibuat dalam bentuk bilangan dalam basis 2 (bilangan biner). Total kemungkinan koordinat orientasi *edge* yang berbeda adalah $2^{11} = 2048$, dengan tiap koordinat direpresentasikan dengan sebuah angka dari 0 hingga 2047. Perhatikan bahwa orientasi dari *edge* ke-12 tidak perlu disimpan, karena orientasinya dapat ditentukan oleh orientasi 11 *edge* lainnya, di mana:

$$\sum_{i=0}^{11} orientasi_i \equiv 0 (mod\ 2) \qquad (4)$$

B.3. Corner Pemutation Coordinate (CP)

Corner Permutation Coordinate (CP) adalah koordinat untuk merepresentasikan permutasi (posisi) dari delapan *corner* pada sebuah kubus rubik. Koordinat ini hanya fokus pada posisi dari kedelapan *corner*, dan tidak memerhatikan orientasi dari *corner*. Terdapat 8 *corner* pada sebuah kubus rubik, sehingga total permutasi yang mungkin adalah $8! = 40320$, dengan tiap koordinat direpresentasikan dengan sebuah angka dari 0 hingga 40319. Representasi ini dilakukan dengan menggunakan sistem faktorial:

$$CP = \sum_{i=0}^{7} rank_i \times i! \qquad (5)$$

Di mana $rank_i$ menyatakan jumlah *corner* pada indeks yang lebih besar dari i dengan nilai yang lebih besar dari *corner* i. Nilai dari sebuah *corner* didapatkan dengan urutan: URF < UFL < ULB < UBR < DFR < DLF < DBL < DRB.

B.4. Edge Permutation Coordinate (EP)

Edge Permutation Coordinate (EP) adalah koordinat untuk merepresentasikan permutasi (posisi) dari dua belas *edge* pada sebuah kubus rubik. Koordinat ini hanya fokus pada posisi dari kedua-belas *edge*, dan tidak memerhatikan orientasi dari *edge*. Representasi ini mirip dengan representasi CP, yang membedakannya hanyalah terdapat 12 *edge* pada sebuah kubus rubik, sehingga total permutasi yang mungkin adalah 12!, dengan tiap koordinat direpresentasikan dengan sebuah angka dari 0 hingga $12! - 1$. Representasi ini dilakukan dengan menggunakan sistem faktorial:

$$EP = \sum_{i=0}^{11} rank_i \times i! \qquad (6)$$

B.5. UD-Slice Coordinate

UD-Slice Coordinate adalah koordinat untuk merepresentasikan posisi dari 4 buah *edge* yang akan berada di antara sisi U dan sisi D (UD-*Slice*) pada kubus rubik yang telah diselesaikan. Dari 12 *edge* yang berada pada kubus rubik, kita memilih 4 *edge* untuk berada di UD-slice, sehingga total kemungkinan kombinasi koordinat berbeda yang mungkin adalah $\binom{12}{4} = 495$, dengan tiap koordinat direpresentasikan dengan sebuah angka dari 0 hingga 494. Representasi ini ditentukan dengan cara:

$$UD - SliceCoord = \sum_{i=0}^{11} isUDSlice_i \times \binom{i}{count} \qquad (7)$$

Di mana:

- $isUDSlice_i$ adalah 1 jika *edge* i yang seharusnya berada pada UD-Slice, 0 jika sebaliknya.
- $\binom{i}{count}$ adalah kombinasi i dengan jumlah *edge* UD-Slice yang telah diproses.

B.6. Phase 2 UD-Slice Coordinate

Phase 2 UD-Slice Coordinate adalah koordinat untuk memastikan bahwa 4 *edge* UD-Slice telah berada pada posisi yang sesuai. *UD-Slice Coordinate* sebelumnya hanya memastikan bahwa 4 *edge* yang akan berada pada UD-slice telah berada pada *layer* tengah, tanpa memperhatikan posisinya sudah tepat atau belum. Maka, total kemungkinan koordinat yang berbeda adalah $4! = 24$, dengan tiap koordinat direpresentasikan dengan angka dari 0 hingga 23.

C. Basic Move

Pada *phase 1* dari *two-phase-algorithm*, koordinat yang digunakan untuk merepresentasikan *state* kubus rubik adalah *Corner Orientation* (CO), *Edge Orientation* (EO), dan UD-*Slice Coordinate*. Dalam kata lain, sebuah kubus rubik direpresentasikan dalam sebuah triplet (x, y, z), dengan: x adalah koordinat CO; y adalah koordinat EO; z adalah koordinat UD-slice. Pada *phase 2*, koordinat yang digunakan adalah *Corner Permutation* (CP), *Edge Permutation* (EP), dan *Phase 2 UD-Slice Coordinate*. Perhatikan bahwa EP yang digunakan pada *phase 2* hanya fokus pada 8 *edges*, sehingga total kemungkinan bukan lagi 12!, melainkan $8! = 40320$.

Setelah merepresentasikan kondisi sebuah kubus rubik dengan sistem koordinat, proses komputasi menjadi lebih efisien. Sebuah gerakan M sekarang pada dasarnya adalah gerakan yang mengubah triplet koordinat (x_1, y_1, z_1) menjadi koordinat yang berbeda (x_2, y_2, z_2). Untuk efisiensi, dapat digunakan sebuah *Lookup Table*, yaitu tabel yang menyimpan efek dari suatu gerakan dasar pada koordinat. Sebagai contoh, pada *phase 1*, akan disimpan *CO Move Table, EO Move Table,* dan *UD-Slice Move Table* yang telah di-*precompute* sebelumnya.

Misal, koordinat saat ini adalah CO = 2024, EO = 12, UD-Slice = 31, dan akan dilakukan gerakan R pada kubus rubik. Maka, transformasi menjadi koordinat baru adalah:

- CO' = $lookup(CO, R)$
- EO' = $lookup(EO, R)$
- UD' = $lookup(UD, R)$

D. Pruning Table

Pruning Table adalah salah satu elemen utama dalam *two-phase-algorithm* yang digunakan untuk mengefisienkan waktu komputasi dalam pencarian solusi kubus rubik. *Pruning table* pada dasarnya adalah *heuristic estimation* yang akan digunakan untuk membatasi ruang pencarian pada algoritma *Iterative Deepening A** (IDA*) saat proses pencarian solusi. *Pruning table* adalah tabel yang di-*precompute* (telah dihitung terlebih dahulu), yang berisi informasi terkait jarak minimum dari suatu posisi/*state* kubus rubik menuju posisi/*state* target tertentu.

Pruning Table dibangun menggunakan algoritma *Breadth-First Search* (BFS), yaitu salah satu algoritma pencarian pada graf yang populer. Langkah-langkah algoritma untuk membangun *pruning table* adalah:

1. Mulai dari posisi target: Inisialisasi posisi target dengan nilai 0.
2. Lakukan seluruh 18 gerakan yang mungkin pada kubus rubik, yaitu R, R', R2, L, L', L2, ..., B, B', B2.
3. Pada tiap gerakan yang dilakukan, jika posisi baru yang didapatkan belum pernah dikunjungi sebelumnya, isi posisi dengan nilai jarak 1.
4. Ulangi langkah 2 dan 3 pada seluruh posisi yang baru dikunjungi, sambil meng-*increment* nilai jarak pada tiap tahap.

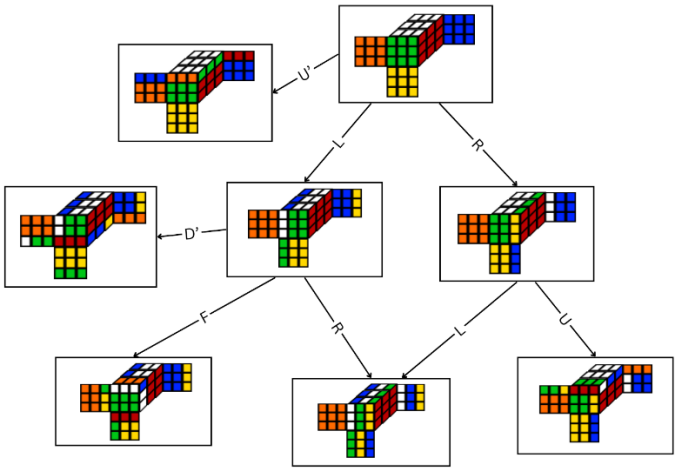
E. Solving The Cube

Akhirnya, setelah berhasil merepresentasikan *state* sebuah kubus rubik secara efisien, yaitu dengan sebuah triplet koordinat, serta setelah membangun *pruning table* yang akan digunakan sebagai *heuristic function*, kubus rubik sekarang siap untuk diselesaikan secara efisien, yaitu melalui algoritma *Iterative Deepening A** (IDA*).

Ingat kembali bahwa *two-phase-algorithm* terbagi menjadi dua fase, yaitu *phase 1* dan *phase 2*. Pada *phase 1*, kondisi target adalah sebuah subset yang disebut subset G1, yaitu subset dengan seluruh orientasi *corner* dan *edge* yang tepat, dan UD-*slice edges* telah berada pada *layer* tengah. Artinya, subset G1 berhasil dicapai apabila triplet koordinat pada *phase 1* telah menjadi (0,0,0),yaitu $CO = 0$, $EO = 0$, dan UD-*Slice Coordinate* = 0. Kemudian, setelah *phase 1* berakhir, yaitu telah dicapai subset G1, *phase 2* dimulai. Pada *phase 2*, sebuah kubus rubik dapat dinyatakan berhasil disolve ketika triple kooridinat pada *phase 2* juga menjadi (0, 0, 0), yaitu $CP = 0$, $EP = 0$, dan *Phase 2 UD-Slice Coordinate* = 0.

Proses mencari solusi pada dengan *two-phase-algorithm* pada sebuah kubus rubik acak menggunakan algoritma *Iterative Deepening A* (IDA*)* dilakukan dengan menganggap seluruh ruang pencarian sebagai suatu graf. Simpul pada graf didefinisikan sebagai tripel koordinat *state* kubus rubik. Terdapat sebuah sisi di antara dua simpul v_i dan v_j apabila *state* v_i dapat mencapai *state* v_j hanya dalam satu gerakan. Perhatikan gambar di bawah sebagai ilustrasi. (Catatan: ilustrasi di bawah

hanya menggambarkan sebagian gerakan yang mungkin, tidak secara keseluruhan).



Gambar 14. Contoh Representasi Graf Pencarian Solusi
Sumber: Arsip Pribadi

Tahapan dalam proses pencarian solusi kubus rubik menggunakan IDA* adalah sebagai berikut:

1. *Phase 1* dimulai, Inisialisasi *threshold* dengan nilai *heuristic estimates* pada koordinat *state* saat ini, yaitu estimasi langkah yang dibutuhkan untuk mencapai kondisi G1. *Heuristic function* ini didapatkan dari *pruning table* yang telah di-*precompute* sebelumnya.
2. Lakukan DFS: jelajahi graf sambil menghitung jarak dan *heuristic estimates*. Penjelajahan berhenti (di-*prune*) ketika nilai $f(n) = g(n) + h(n) > threshold$.
3. Jika simpul target ditemukan, yaitu *state* dengan tripel koordinat (0, 0, 0), *phase 1* telah selesai dan dilanjutkan dengan *phase 2*.
4. Jika simpul target tidak ditemukan, *update* nilai *threshold* dengan nilai $f(n)$ minimum dari simpul-simpul yang telah di-*prune* sebelumnya.
5. Ulangi langkah-langkah di atas hingga target ditemukan, yaitu subset G1 tercapai.
6. Setelah *phase 1* selesai, dilanjutkan dengan *phase 2*. Proses pencarian pada *phase 2* juga sama seperti *phase 1*, dengan catatan: gerakan yang dapat dilakukan hanya terbatas pada gerakan U, D, R2, L2, F2, dan B2.

Dapat dipastikan bahwa jumlah langkah maksimum untuk menyelesaikan *phase 1* dan mencapai G1 adalah 12 langkah, dan jumlah langkah maksimum untuk menyelesaikan *phase 2* adalah 18 langkah. Maka, langkah maksimum untuk menyelesaikan kubus rubik dengan *two-phase-algorithm* adalah sebanyak 30 langkah.

Untuk solusi yang lebih optimal, pencarian pada *phase 1* tidak akan berhenti ketika telah mencapai *state* G1. Pencarian akan terus dilanjutkan untuk mencari solusi dengan jumlah langkah yang lebih pendek pada *phase 2*. Sebagai contoh, awalnya ditemukan *state* G1 hanya dalam 9 langkah, kemudian *phase 2* berhasil diselesaikan dalam 12 langkah. Namun, pencarian tetap dilanjutkan, hingga menemukan *state* G1 dalam 11 langkah, tetapi *phase* 2 berhasil diselesaikan dalam 6 langkah.

V. KESIMPULAN

Makalah ini membahas tentang optimasi penyelesaian Rubik's Cube menggunakan algoritma *Iterative Deepening A** (IDA*) pada Metode yang diperkenalakan Herbert Kociemba, yaitu *two-phase-algorithm*. Metode ini membagi proses pencarian solusi menjadi dua fase, yaitu *phase 1* dan *phase 2*.

Pada *phase 1*, kubus rubik akan diubah menjadi subset G1 yang memastikan orientasi tiap *corner* dan *edge* benar dan *edge* UD-*Slice* telah berada di *layer* tengah. Langkah maksimum untuk menyelesaikan *phase 1* adalah sebanyak 12 langkah. Pada *phase 2*, kubus rubik akan diselesaikan sepenuhnya, dengan maksumum 18 langkah. Maka, langkah maksimum yang diperlukan untuk menyelesaikan kubus rubik menggunakan *two-phase-algorithm* adalah sebanyak 30 langkah.

Penggunaan algoritma *Iterative Deepening A** (IDA*) sangat mengefisienkan sumber daya komputasi yang dibutuhkan, terutama karena besarnya ruang eksplorasi untuk mencari solusi dari sebuah kubus rubik, terutama dengan pemanfaatan *pruning table* sebagai *heuristic function* yang dapat mengurangi jumlah simpul yang perlu dikunjungi secara signifikan.

VI. UCAPAN TERIMA KASIH

Terima kasih kepada Tuhan Yang Maha Esa karena berkat rahmat dan karunia-Nya, makalah yang berjudul "Optimasi Penyelesaian Rubik's Cube: Penerapan Algoritma Iterative Deepening A* pada Metode Kociemba untuk Menemukan Solusi Dalam 30 Langkah" dapat diselesaikan dengan lancar tanpa adanya hambatan. Terima kasih juga kepada dosen-dosen pengampu mata kuliah IF1220 Matematika Diskrit, terutama kepada Bapak Rinaldi Munir karena telah membagikan ilmu yang mendukung pembuatan makalah ini.

REFERENCES

[1] A. Reinefeld and T. A. Marsland, "Enhanced Iterative-Deepening Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 16, no. 7, pp. 701–710, Jul. 1994.

[2] H. Kociemba, "Two-Phase Algorithm for Solving Rubik's Cube," [Online]. Available: https://kociemba.org/cube.htm. [Accessed: Dec. 30, 2024].

[3] R. Munir, "Graf (Bagian 1)," Institut Teknologi Bandung, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. [Accessed: Dec. 30, 2024].

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, 30 Desember 2024

Albertus Christian Poandy