

# **Pencarian Solusi Diagonal Magic Cube dengan Local Search**

Laporan Tugas Besar 1  
IF3170 Intelelegensi Artifisial



**Disusun Oleh Kelompok 18:**

|          |                            |
|----------|----------------------------|
| 12821046 | Fardhan Indrayesa          |
| 13522037 | Farhan Nafis Rayhan        |
| 13522091 | Raden Francisco Trianto B. |
| 18321011 | Wikan Priambudi            |

**Program Studi Teknik Informatika**

**Sekolah Teknik Elektro dan Informatika**

**Institut Teknologi Bandung**

**2024**

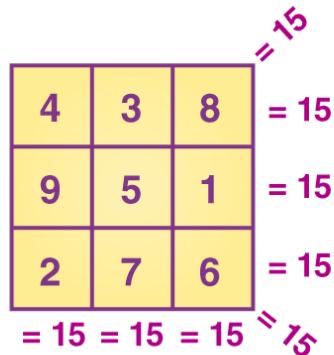
## DAFTAR ISI

|   |           |
|---|-----------|
| <b>DAFTAR ISI.....</b>                        | <b>2</b>  |
| <b>BAB 1 DESKRIPSI PERSOALAN.....</b>         | <b>3</b>  |
| <b>BAB 2 PEMBAHASAN.....</b>                  | <b>5</b>  |
| 2.1. Objective Function.....                  | 5         |
| 2.2. Implementasi Algoritma Local Search..... | 6         |
| 2.2.1. Diagonal Magic Cube.....               | 6         |
| 2.2.2. Objective Function.....                | 9         |
| 2.2.3. Hill-Climbing Steepest Ascent.....     | 10        |
| 2.2.4. Hill-Climbing with Sideways Move.....  | 12        |
| 2.2.5. Random Restart Hill-Climbing.....      | 14        |
| 2.2.6. Stochastic Hill Climbing.....          | 17        |
| 2.2.7. Simulated Annealing.....               | 19        |
| 2.2.8. Genetic Algorithm.....                 | 22        |
| <b>BAB 3 HASIL DAN ANALISIS.....</b>          | <b>25</b> |
| 3.1. Hasil Eksperimen.....                    | 25        |
| 3.1.1. Hill-Climbing Steepest Ascent.....     | 25        |
| 3.1.2. Hill-Climbing with Sideways Move.....  | 29        |
| 3.1.3. Random Restart Hill-Climbing.....      | 33        |
| 3.1.4. Stochastic Hill-Climbing.....          | 37        |
| 3.1.5. Simulated Annealing.....               | 43        |
| 3.1.6. Genetic Algorithm.....                 | 49        |
| 3.2. Analisis.....                            | 58        |
| <b>BAB 4 KESIMPULAN DAN SARAN.....</b>        | <b>62</b> |
| 4.1. Kesimpulan.....                          | 62        |
| 4.2. Saran.....                               | 62        |
| <b>PEMBAGIAN TUGAS.....</b>                   | <b>63</b> |
| <b>REFERENSI.....</b>                         | <b>65</b> |

## BAB 1

### DESKRIPSI PERSOALAN

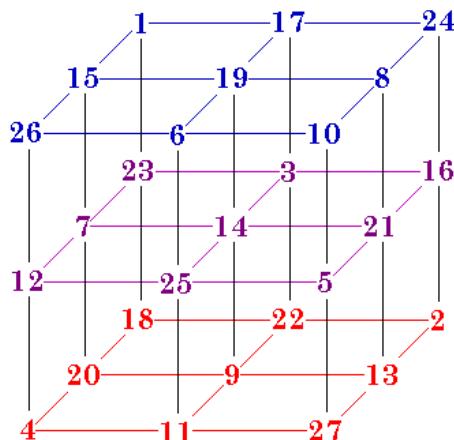
**Magic square** merupakan kotak berukuran  $n \times n$  yang memiliki hasil penjumlahan baris, kolom, dan diagonal yang sama, yaitu  $M(n)$  yang dikenal dengan magic number. Secara umum, magic square tersusun oleh angka unik dari 1 hingga  $n^2$ , tapi terdapat variasi yang membebaskan isinya yang disebut multi magic square. Namun pada persoalan ini kita hanya akan menggunakan aturan dasar.



Gambar 1. Magic Square  $M = 15$

Sumber: [Magic Square - The Algorithms](#)

Magic square memiliki dimensi-2 yang terdiri dari baris dan kolom. Namun Magic square adalah permasalahan dasar yang menjadi lebih kompleks ketika dimensinya diubah menjadi dimensi-3, yaitu magic cube.



Gambar 2. Order-3 Magic Cube

Sumber: [John Hendricks](#)

**Magic cube** merupakan suatu kubus berukuran  $n \times n \times n$  yang tersusun dari angka 1 hingga  $n^3$ . Mirip dengan magic square, semua  $n^2$  baris,  $n^2$  kolom,  $n^2$  pilar, dan 4 diagonal ruang, masing-masing harus memiliki hasil penjumlahan yang sama yaitu magic number(magic constant) yang dilambangkan dengan  $M_3(n)$ .

**Diagonal magic cube** (*perfect magic cube*) adalah salah satu variasi dari magic cube. Perbedaan dari magic cube biasa adalah pada diagonal bidang. Pada diagonal magic cube, penjumlahan pada diagonal juga harus sama dengan nilai magi number.

Untuk memperjelas, berikut adalah syarat-syarat dan ketentuan dari suatu diagonal magic cube secara lengkap:

Diagonal Magic Cube adalah kubus berukuran  $n \times n \times n$  yang disusun dari bilangan 1 hingga  $n^3$  tanpa pengulangan, dengan persyaratan:

- Terdapat satu angka yang merupakan magic number dari kubus tersebut (Magic number tidak harus termasuk dalam rentang 1 hingga  $n^3$ , magic number juga bukan termasuk ke dalam angka yang harus dimasukkan ke dalam kubus)
- Jumlah angka-angka untuk setiap baris sama dengan magic number
- Jumlah angka-angka untuk setiap kolom sama dengan magic number
- Jumlah angka-angka untuk setiap tiang sama dengan magic number
- Jumlah angka-angka untuk seluruh diagonal ruang pada kubus sama dengan magic number
- Jumlah angka-angka untuk seluruh diagonal pada suatu potongan bidang dari kubus sama dengan magic number

Dalam tugas ini, mahasiswa akan menyusun Diagonal Magic Cube berukuran  $5 \times 5 \times 5$ , dengan keadaan awal berupa susunan acak angka 1 hingga  $5^3$ . Setiap iterasi pada algoritma pencarian lokal mengizinkan penukaran posisi dua angka di dalam kubus, yang tidak harus bersebelahan. Khusus untuk genetic algorithm, diperbolehkan menukar lebih dari dua angka dalam satu iterasi, meskipun penukaran dua angka tetap diperbolehkan.

## BAB 2

### PEMBAHASAN

#### 2.1. Objective Function

Perhatikan bahwa magic number bagi suatu magic cube akan konstan. Menurut referensi (John Hendricks), magic number  $M$  bagi suatu magic cube berukuran  $N \times N \times N$  adalah

$$M_3(N) = \frac{N(N^3+1)}{2}$$

Maka, bagi kubus  $5 \times 5 \times 5$ , magic number nya adalah 315. Angka ini akan digunakan pada fungsi objektif permasalahan ini.

Fungsi objektif yang dirancang dalam penyelesaian masalah *magic cube* bertujuan untuk mengevaluasi seberapa baik susunan bilangan dalam kubus memenuhi kriteria magic cube. Dalam konteks ini, fungsi objektif memberikan nilai yang menunjukkan jumlah ketidaksesuaian antara total nilai pada tiap baris, kolom, tiang, serta semua diagonal dengan nilai *magic constant*. Semakin rendah nilai fungsi objektif, semakin dekat susunan kubus dengan konfigurasi ideal dari *magic cube*.

$$F(n) = 109 - (R + C + P + DR + DB)$$

Dengan definisi untuk setiap variabelnya:

| Variabel | Definisi   |
|----------|--|
| $F(n)$   | Fungsi objektif  |
| $R$      | Banyaknya baris dengan penjumlahan semua angkanya berjumlah tidak sama dengan 315          |
| $C$      | Banyaknya kolom dengan penjumlahan semua angkanya berjumlah tidak sama dengan 315          |
| $P$      | Banyaknya pilar dengan penjumlahan semua angkanya berjumlah tidak sama dengan 315          |
| $DR$     | Banyaknya diagonal baris dengan penjumlahan semua angkanya berjumlah tidak sama dengan 315 |
| $DB$     | Banyaknya diagonal ruang dengan penjumlahan semua angkanya berjumlah tidak sama dengan 315 |

Pada kubus  $5 \times 5 \times 5$ , terdapat  $5^2 = 25$  baris,  $5^2 = 25$  kolom,  $5^2 = 25$  pilar, 30 total diagonal pada potongan dan empat diagonal ruang masing-masing memiliki jumlah yang sama dengan sebuah angka  $M(n)$ .

Secara total, jumlah maksimum baris, kolom, dan diagonal yang mungkin mencapai nilai 315 adalah:

$$25 + 25 + 25 + 30 + 4 = 109$$

Maka, nilai dari suatu state pun akan berada dalam rentang  $[0, 109]$ , dimana state dengan nilai 109 merupakan solusi *magic cube* yang ingin kita temukan.

Pembangkitan neighbor(tetangga) bagi suatu state akan dilakukan dengan cara menukar posisi 2 buah kubus di dalam magic cube. Karena ada 125 buah kubus, maka dapat dilakukan perhitungan sebagai berikut

$$C_r^n = \frac{n!}{r!(n-r)!}$$

Dengan nilai  $n = 125$  dan  $r = 2$  maka didapatkan perhitungan sebagai berikut

$$C_2^{125} = \frac{125!}{2!(123)!} = \frac{125 \times 124}{2} = 7750$$

Dengan demikian terdapat tepat 7750 buah neighbor yang dapat dipilih. Pemilihan neighbor sebagai successor akan berbeda tergantung algoritma local search yang digunakan.

## 2.2. Implementasi Algoritma Local Search

Untuk mengimplementasikan local search dalam menyelesaikan persoalan Diagonal Magic Cube, kami menggunakan bahasa pemrograman Python, dengan bantuan library Numpy untuk mempercepat pemrosesan data kubus yang sangat banyak.

### 2.2.1. Diagonal Magic Cube

Dalam implementasi algoritma-algoritma local search, kami membuat struktur data Magic Cube sebagai berikut.

| <b>Class</b>        |   |
|---------------------|---|
| class MagicCube     | Merepresentasikan sebuah magic cube         |
| <b>Attribute</b>    |   |
| size (int)          | Ukuran dari kubus                           |
| data (array of int) | Representasi Matriks 3 dimensi dalam sebuah |

|                            |  |
|----------------------------|--|
|                            | array satu dimensi   |
| magic_sum (int)            | Nilai magic number untuk magic cube berukuran size   |
| <b>Fungsi dan Prosedur</b> |  |
| get_state_value() → int    | Mendapatkan state value menggunakan fungsi objektif yang sudah dibuat                            |
| copy() → MagicCube         | Membuat sebuah copy dari magic cube  |
| randomize()                | Mengacak isi dari data magic cube  |
| is_perfect() → bool        | Mengecek apakah magic cube merupakan sebuah perfect magic cube atau diagonal magic cube          |
| swap_index(i1, i2)         | Menukar elemen pada index i1 dengan elemen pada index i2   |
| swap_index_copy(i1, i2)    | Membuat copy dari magic cube dan melakukan pertukaran elemen pada index i1 dengan elemen pada i2 |

Berikut merupakan potongan Source Code untuk Class tersebut. Perlu diperhatikan bahwa kami hanya memberikan source code untuk bagian yang dianggap penting saja.

```
class MagicCube:
    """
    A Magic Cube represented by a 1D array

    :var size: The dimensions of the Magic Cube, default is 5
    :var data: The 1D array representing the Magic Cube
    :var magic_sum: The magic number/constant for the Magic Cube
    """

    def __init__(self, size=5, data=None):
        """
        Generates a Magic Cube in the form of a 1D array with
        elements from 1 to 125.

        :param size: Magic Cube Dimensions, default = 5
        """
        self.size: int = size # dimensions, default 5x5x5
        self.data: np.ndarray = data if data is not None else
            np.random.permutation(np.arange(1, size**3 + 1))
        self.magic_sum: int = size * (size**3 + 1) // 2 # default
        is 315
```

```

def copy(self) -> 'MagicCube':
    """
        Creates a deep copy of the current MagicCube instance.

    :return: A new MagicCube instance that is a deep copy of
    this instance.
    """
    return copy.deepcopy(self)

def randomize(self) -> None:
    """
        Randomize the Magic Cube by shuffling the 1D array.
    """
    np.random.shuffle(self.data)

def swap_index(self, i1, i2):
    """
        Swap two elements in the Magic Cube at the specified indices
    """
    self.data[i1], self.data[i2] = self.data[i2], self.data[i1]

def swap_index_copy(self, i1, i2) -> 'MagicCube':
    """
        Returns a new Magic Cube with two elements swapped at the
    specified indices.
    """
    new_cube = self.copy()
    new_cube.swap_index(i1, i2)
    return new_cube

def get_state_value(self):
    """
        Returns the value of the current state of the Magic Cube.
    """
    return ObjectiveFunction.get_object_value(self)

def is_perfect(self) -> bool:
    """
        Check if the Magic Cube is a perfect magic cube.

        A perfect magic cube has the same magic sum for:
        - All rows
        - All columns
        - All pillars
        - All space diagonals
        - All side diagonals

    :return: True if the cube is perfect, False otherwise.
    """
    # Check all rows
    for z in range(self.size):
        for y in range(self.size):
            if np.sum(self.get_row(y, z)) != self.magic_sum:
                return False

    # Check all columns
    for z in range(self.size):

```

```

        for x in range(self.size):
            if np.sum(self.get_col(x, z)) != self.magic_sum:
                return False

        # Check all pillars
        for y in range(self.size):
            for x in range(self.size):
                if np.sum(self.get_pillar(x, y)) != self.magic_sum:
                    return False

        # Check all space diagonals
        for diag in self.get_space_diags():
            if np.sum(diag) != self.magic_sum:
                return False

        # Check all side diagonals on each axis
        for diag_x in self.get_side_diags_x():
            if np.sum(diag_x) != self.magic_sum:
                return False

        for diag_y in self.get_side_diags_y():
            if np.sum(diag_y) != self.magic_sum:
                return False

        for diag_z in self.get_side_diags_z():
            if np.sum(diag_z) != self.magic_sum:
                return False

    # If all checks pass, it's a perfect magic cube
    return True

```

### 2.2.2. Objective Function

Untuk mendapatkan state value dari masing-masing state berikut adalah implementasi dari fungsi objektif untuk magic cube.

| Class                              |   |
|------------------------------------|---|
| class ObjectiveFunction            | Static Class untuk menghitung objective function dari suatu magic cube  |
| Fungsi dan Prosedur                |   |
| get_object_value(magic_cube) → int | Mendapatkan state value menggunakan fungsi objektif yang sudah dibuat   |
| __check_315(magic_cube) → int      | Menghitung ( $R + C + P + DR + DB$ ) yaitu total baris/kolom/pilar/diagonal ruang/diagonal baris yang memiliki jumlah bernilai magic number yaitu 315 |

Berikut merupakan potongan Source Code untuk Class tersebut. Perlu diperhatikan bahwa kami hanya memberikan source code untuk bagian yang dianggap penting saja.

```
class ObjectiveFunction:
    """
        Objective function for local search in a magic cube.

        Static Class --> no Constructor

        Use by calling ObjectiveFunction.get_object_value: Returns
        objective value from internal function
    """

    @staticmethod
    def get_object_value(magic_cube: MagicCube) -> int:
        """
            Returns objective value from internal function.

            :return: Objective function value of a magic cube
        """

        # 109 is the maximum possible state value for a 5x5x5 cube
        return 109 - ObjectiveFunction.__check_315(magic_cube)
```

### 2.2.3. Hill-Climbing Steepest Ascent

Berikut adalah implementasi dari algoritma Hill-Climbing Steepest Ascent

| Class                                     |   |
|---|---|
| class HillClimbSteepest                   | Class sebagai implementasi dari Hill-Climbing Steepest Ascent   |
| Attribute                                 |   |
| states (array of magic cube)              | Menyimpan state-state untuk setiap iterasi mulai dari initial state hingga final state yang dihasilkan. Index 0 merupakan initial state, Index len-1 merupakan final state. |
| Fungsi dan Prosedur                       |   |
| hill_climb_steepest_ascent()              | Fungsi algoritma Hill Climbing Steepest Ascent  |
| __get_highest_value_neighbour() : private | Fungsi private untuk mendapatkan neighbor terbaik dari semua successor yang mungkin dari current state  |

Berikut merupakan potongan Source Code untuk Class tersebut.

```
class HillClimbSteepest:
    """
        A local search algorithm: Hill-Climbing Steepest Ascent.
    """
    def __init__(self, initial_cube: MagicCube):
        self.states: list[MagicCube] = [initial_cube]      #
        self.states[-1] is the current state

        def hill_climb_steepest_ascent(self) -> tuple[list[MagicCube], int]:
            """
                Returns the best state and the amount of iterations.
            """
            current = self.states[-1].copy()                  # copy of Magic
            cube class initial
            current_value = current.get_state_value()         # initial state
            value
            i = 0                                              # initiation the
            number of iterations

            # Loop of hill-climbing steepest ascent
            while True:
                # find the best state and its value
                neighbor, neighbour_value =
                self.__get_highest_value_neighbour()

                if neighbour_value <= current_value:
                    # if the neighbor state value is LESS than or EQUAL
                    to the current state value,
                    # stop the local search
                    return self.states, i

                self.states.append(neighbor)
                # current = neighbor --> using self.states[-1]
                current_value = neighbour_value
                i += 1
                print("iteration", i, "- current value", current_value)

            # -- INTERNAL FUNCTIONS --

            def __get_highest_value_neighbour(self):
                """
                    Returns the best cube state and its state value
                """

                # Initialize arrays for successors and their state values
                successors: list[MagicCube] = []
                successors_state: list[int] = []
                data_len = self.states[-1].size ** 3

                # Generate all possible swaps and calculate their state
                values
                for x1 in range(data_len):
                    for x2 in range(x1 + 1, data_len):
                        # get successor
                        cube_swap = self.states[-1].swap_index_copy(x1, x2)
```

```

state_value = cube_swap.get_state_value()

# add to successors list
successors.append(cube_swap)
successors_state.append(state_value)

# Convert to NumPy arrays for fast indexing
np_successors_state = np.array(successors_state)

# Find the index of the maximum state value
max_index = np.argmax(np_successors_state)

# Return the best successor and its state value
return successors[max_index], successors_state[max_index]

```

#### 2.2.4. Hill-Climbing with Sideways Move

Berikut adalah implementasi dari algoritma Hill-Climbing with Sideways Move

| Class                                     |   |
|---|---|
| class HillClimbSideways                   | Class sebagai implementasi dari Hill-Climbing with Sideways Move  |
| Attribute                                 |   |
| states (array of magic cube)              | Menyimpan state-state untuk setiap iterasi mulai dari initial state hingga final state yang dihasilkan. Index 0 merupakan initial state, Index len-1 merupakan final state. |
| max_sides                                 | Jumlah maksimum iterasi sideways.   |
| Fungsi dan Prosedur                       |   |
| hill_climb_sideways_move()                | Fungsi algoritma Hill Climbing with Sideways Move.  |
| __get_highest_value_neighbour() : private | Fungsi private untuk mendapatkan neighbor terbaik dari semua successor yang mungkin dari current state  |

Berikut merupakan potongan Source Code untuk Class tersebut. Perlu diperhatikan bahwa kami hanya memberikan source code untuk bagian yang dianggap penting saja.

```
class HillClimbSideways:
```

```

"""
A local search algorithm: Hill-Climbing Sideways Move.
"""

def __init__(self, max_side, initial_cube: MagicCube):
    self.states: list[MagicCube] = [initial_cube] # self.states[-1] is the current state
    self.max_sides: int = int(max_side)

    def hill_climb_sideways_move(self) -> tuple[list[MagicCube], int]:
        """
        Returns the best state and the amount of iterations.
        """
        current = self.states[-1].copy() # copy of Magic cube class initial
        current_value = current.get_state_value() # initial state value
        i = 0 # initiation the number of iterations
        i_sides = 0 # initiation the number of iterations with sideways move

        # Loop of hill-climbing sideways move
        while True:
            # find the best state and its value
            neighbour, neighbour_value =
                self.__get_highest_value_neighbour()

            if (neighbour_value < current_value) or (i_sides == self.max_sides):
                # if the neighbour objective function value is LESS than the current objective function value
                # stop the local search
                return self.states, i

            self.states.append(neighbour)
            if neighbour_value == current_value:
                i_sides += 1
            else:
                i_sides = 0
            current_value = neighbour_value
            i += 1
            print(f"iteration {i}, sideways iteration {i_sides} - current value {current_value}")

        # -- INTERNAL FUNCTION --
        def __get_highest_value_neighbour(self):
            """
            Returns the best state and its state value.
            """

            # Initialize arrays for successors and their state values
            successors: list[MagicCube] = []
            successors_state: list[int] = []
            data_len = self.states[-1].size ** 3

            # Generate all possible swaps and calculate their state values
            for x1 in range(data_len):
                for x2 in range(x1 + 1, data_len):
                    # get successor
                    cube_swap = self.states[-1].swap_index_copy(x1, x2)
                    state_value = cube_swap.get_state_value()

                    # add to successors list
                    successors.append(cube_swap)

            return successors, successors_state

```

```

        successors_state.append(state_value)

    # Convert to NumPy arrays for fast indexing
    np_successors_state = np.array(successors_state)

    # Find the index of the maximum state value
    max_index = np.argmax(np_successors_state)

    max_indices = np.where(np_successors_state ==
    np_successors_state.max())[0]           # Get max successor
    value indices
    successors_max = np.array(successors)[max_indices]
    # Successors that have maximum state value
    successors_state_max = np_successors_state[max_indices]
    # Successor values that have maximum state value
    i_same = []

        # Get the indices of successors that are present in the
    current state list
        for idx in range(len(self.states)):
            for idx_max, successor_max in enumerate(successors_max):
                if (successor_max.data == self.states[idx].data):
                    i_same.append(idx_max)

        if len(i_same) > 0:
            # Delete list of successors (with maximum state value)
            # that are contained in the current state list
            state_same_drop = np.delete(successors_max, i_same,
            axis=0)
            state_value_same_drop = np.delete(successors_state_max,
            i_same, axis=0)

            # If there are some successors equal to the current
            state list
            # return the next state that has maximum state value
            return state_same_drop[0], state_value_same_drop[0]

        # Return the best successor and its state value
        return successors[max_index], successors_state[max_index]

```

### 2.2.5. Random Restart Hill-Climbing

Random Restart Hill-Climbing adalah varian dari algoritma hill climbing yang dirancang untuk mengatasi kendala local maximum, yang juga dihadapi oleh metode Steepest Ascent Hill-Climbing dan Hill-Climbing with Sideway Move. Dalam algoritma ini, jika proses mencapai *local maximum*, pencarian diulang dari kondisi awal yang diacak. Langkah ini diulang beberapa kali, dengan harapan bahwa salah satu percobaan akan menemukan solusi global atau setidaknya solusi yang lebih baik dibandingkan hasil pencarian sebelumnya.

Random Restart meningkatkan kemungkinan menemukan solusi optimal dengan menghindari jebakan *local maximum*, meskipun membutuhkan waktu lebih lama karena pencarian dilakukan dari berbagai kondisi awal. Ini merupakan pendekatan yang sederhana namun efektif untuk memperluas area

pencarian dalam ruang solusi. Semakin banyak pengulangan yang dilakukan, semakin luas cakupan ruang solusi yang terjangkau dan semakin lama waktu yang dibutuhkan untuk menemukan solusi. Selain itu, lebih banyak pengulangan juga meningkatkan peluang untuk menemukan solusi optimal global.

Pada permasalahan Diagonal Magic Cube (Perfect Magic Cube) berukuran 5x5x5, Random Restart Hill-Climbing dapat dimanfaatkan untuk menemukan susunan angka yang memenuhi syarat magic number. Algoritma ini bekerja dengan memulai dari konfigurasi acak, menjalankan proses hill climbing, dan jika solusi yang ditemukan tertahan di local maximum, algoritma diulang dengan konfigurasi acak yang baru.

Implementasi yang dilakukan pada algoritma ini berisi kelas RandomRestartHillClimbing dengan beberapa fungsi seperti evaluate, swap\_elems, revert\_swap, dan run.

| <b>Class</b>                    |   |
|---------------------------------|---|
| class RandomRestartHillClimbing | Class sebagai implementasi dari Hill-Climbing with Random Restart   |
| <b>Attribute</b>                |   |
| self(cube_size (int)            | Ukuran kubus dalam bentuk integer.  |
| self.max_restarts (int)         | Jumlah maksimum restart acak yang diizinkan. Menentukan berapa kali algoritma akan mencoba solusi baru dengan restart acak untuk menemukan solusi terbaik           |
| self.max_iterations (int)       | Jumlah maksimum iterasi per restart. Menentukan berapa kali maksimal algoritma akan mencari neighbor terbaik pada setiap restart.                                   |
| self.best_score (float)         | Skor terbaik yang ditemukan oleh algoritma, diinisialisasi dengan float('inf'). Menyimpan nilai fungsi objektif tertinggi (optimal) yang ditemukan selama eksekusi. |
| self.initial_cube (MagicCube)   | Objek MagicCube yang berisi kondisi awal kubus.   |

|  |  |
|--|--|
| self.best_iteration_scores (list of float) | Daftar yang menyimpan nilai fungsi objektif pada setiap iterasi selama eksekusi algoritma. Berisi serangkaian skor float untuk memplot grafik hasil pencarian.                       |
| best_scores_per_restart (list of float)    | Daftar yang menyimpan nilai terbaik yang dicapai pada setiap restart.  |
| <b>Fungsi dan Prosedur</b>                 |  |
| evaluate                                   | Mengevaluasi objek cube dengan fungsi objektif objective_function. Mengembalikan skor fitness (semakin tinggi semakin baik).   |
| swap_elements                              | Melakukan pertukaran dua elemen acak di dalam kubus untuk mengeksplorasi solusi baru. Mengembalikan indeks dari elemen-elemen yang ditukar.  |
| revert_swap                                | Mengembalikan kondisi awal dari pertukaran elemen yang dilakukan di swap_elements. Membantu mengembalikan kubus ke keadaan semula sebelum mencoba pasangan swap lainnya.             |
| run  | Menjalankan algoritma RandomRestartHillClimbing. Melakukan restart hingga batas max_restarts. Dalam setiap restart, melakukan iterasi max_iterations untuk mencari neighbor terbaik. |

Berikut merupakan potongan dari Class RandomRestartHillClimbing

```
class RandomRestartHillClimbing:
    def run(self):
        self.best_cube = copy.deepcopy(self.initial_cube)
        self.best_score = self.evaluate(self.initial_cube)
        best_states_per_restart = [] # Track the best state of each
        restart
        best_scores_per_restart = [] # Track the best score of each
        restart
        total_iterations = 0

        for restart in range(self.max_restarts):
            cube = copy.deepcopy(self.initial_cube) if restart == 0
            else MagicCube(size=self.cube_size)
            current_score = self.evaluate(cube)
            best_local_state = copy.deepcopy(cube)
```

```

        best_local_score = current_score
        iteration_scores = []

        for iteration in range(self.max_iterations):
            best_neighbor_score = current_score
            best_swap = None

            # Evaluate all neighbors by swapping pairs of
            elements
            for i in range(len(cube.data)):
                for j in range(i + 1, len(cube.data)):
                    # Perform swap
                    cube.data[i], cube.data[j] = cube.data[j],
                    cube.data[i]
                    neighbor_score = self.evaluate(cube)

                    # Track the best neighbor
                    if neighbor_score > best_neighbor_score:
                        best_neighbor_score = neighbor_score
                        best_swap = (i, j)

                    # Revert swap to try the next pair
                    cube.data[i], cube.data[j] = cube.data[j],
                    cube.data[i]

            # If no better neighbor is found, break out of the
            loop
            if best_swap is None:
                print(f"No improvement found at Restart
{restart}, Iteration {iteration}. Best local score:
{best_local_score}")
                break

            # Perform the best swap
            i, j = best_swap
            cube.data[i], cube.data[j] = cube.data[j],
            cube.data[i]
            current_score = best_neighbor_score
            total_iterations += 1

            # Update best state for the current restart
            if current_score > best_local_score:
                best_local_state = copy.deepcopy(cube)
                best_local_score = current_score

            iteration_scores.append(current_score)

            # Save the best state and score for this restart
            best_states_per_restart.append(best_local_state)
            best_scores_per_restart.append(best_local_score)
            print(f"Restart {restart} completed. Best score for this
restart: {best_local_score}")

        return best_states_per_restart, total_iterations
    
```

### 2.2.6. Stochastic Hill Climbing

Untuk mendapatkan state value dari masing-masing state berikut adalah implementasi dari fungsi objektif untuk magic cube.

|       |
|-------|
| Class |
|-------|

|                              |   |
|------------------------------|---|
| class StochasticHillClimb    | Class sebagai implementasi dari Stochastic Hill-Climbing  |
| <b>Attributes</b>            |   |
| states (array of magic cube) | Menyimpan state-state untuk setiap iterasi mulai dari initial state hingga final state yang dihasilkan. Index 0 merupakan initial state, Index len-1 merupakan final state. |
| iteration_value              | Menyimpan nilai state value pada setiap iterasi   |
| <b>Fungsi dan Prosedur</b>   |   |
| stochastic_hill_climb(nmax)  | Melakukan local search dengan algoritma Stochastic Hill Climbing dengan jumlah iterasi maksimum adalah nmax   |
| __generate_neighbors()       | Mendapatkan semua neighbor yang mungkin dari current state  |

Berikut merupakan potongan Source Code untuk Class tersebut. Perlu diperhatikan bahwa kami hanya memberikan source code untuk bagian yang dianggap penting saja.

```
class StochasticHillClimb:
    """
    A local search algorithm: Stochastic Hill-Climbing without
    temperature control.
    """
    def __init__(self, initial_cube: MagicCube):
        self.states: list[MagicCube] = [initial_cube]      #
        self.states[-1] is the current state
        self.iteration_value: list[int] = []

    def stochastic_hill_climb(self, nmax: int) -> tuple[list[any],
int, list[int]]:
        """
        Executes the Stochastic Hill Climbing algorithm for a
        maximum of nmax iterations.
        Returns the best state and the number of iterations
        performed.
        """
        current = self.states[-1].copy()                  # Copy of the
initial cube
        current_value = current.get_state_value()         # Initial state
value
        self.iteration_value.append(current_value)
        i = 0                                              # Initiation of
the number of iterations
```

```

# Loop of stochastic hill-climbing
while i < nmax:
    # Generate neighbors
    neighbors = self.__generate_neighbors()
    if not neighbors or current_value == 109:
        break

    # Randomly select a neighbor
    neighbor = random.choice(neighbors)
    neighbor_value = neighbor.get_state_value()
    self.iteration_value.append(neighbor_value)

    # Always accept the neighbor if it's better
    if neighbor_value > current_value:
        self.states.append(neighbor)
        current_value = neighbor_value

    i += 1
    print("iteration", i, "- current value", current_value)

return self.states, i, self.iteration_value

# -- INTERNAL FUNCTIONS --

def __generate_neighbors(self) -> list[MagicCube]:
    """
    Generates all possible neighbors (successor states).
    """
    successors: list[MagicCube] = []
    data_len = self.states[-1].size ** 3

    for x1 in range(data_len):
        for x2 in range(x1 + 1, data_len):
            cube_swap = self.states[-1].swap_index_copy(x1, x2)
            successors.append(cube_swap)

    return successors

```

### 2.2.7. Simulated Annealing

Untuk mendapatkan state value dari masing-masing state berikut adalah implementasi dari fungsi objektif untuk magic cube.

| Class                        |   |
|------------------------------|---|
| class SimulatedAnnealing     | Class sebagai implementasi dari Simulated Annealing.  |
| Attributes                   |   |
| states (array of magic cube) | Menyimpan state-state untuk setiap iterasi mulai dari initial state hingga final state yang |

|                                    |   |
|------------------------------------|---|
|                                    | dihasilkan. Index 0 merupakan initial state, Index len-1 merupakan final state. |
| data_per_iteration(array of float) | Menyimpan hasil $e^{\Delta E/T}$ per iterasi                                    |
| MAX_TIME                           | Menyimpan jumlah iterasi maksimum   |
| INITIAL_TEMPERATURE                | Menyimpan nilai temperatur awal   |
| <b>Fungsi dan Prosedur</b>         |   |
| simulated_annealing()              | Melakukan local search dengan algoritma Simulated Annealing                     |
| __get_random_neighbour()           | Mendapatkan sebuah neighbor secara random                                       |
| __accept_by_probability()          | Mengecek apakah suatu bilangan acak lebih kecil dari nilai probabilitas         |

Berikut merupakan potongan Source Code untuk Class tersebut. Perlu diperhatikan bahwa kami hanya memberikan source code untuk bagian yang dianggap penting saja.

```
class SimulatedAnnealing:
    """
        Implementation of simulated annealing algorithm

        :var INITIAL_TEMPERATURE: constant of initial temperature of
        algorithm
        :var cube: The initial of a magic cube
        :var objective: The objective value of current state
        :var time: The time variable for iteration
    """

    def __init__(self, initial_cube, cube_size):
        """
            Initialization for the algorithm

            :param initial_cube: The initial magic cube object before
            algorithm starts
            :param cube_size: The size of initial magic cube
        """

        # CONSTANTS (Algorithm Settings)
        self.INITIAL_TEMPERATURE = 2
        self.MAX_TIME = 250000
        self.time = 1
        self(cube = initial_cube
        self.objective =
        ObjectiveFunction.get_object_value(initial_cube)
        self.size = cube_size
```

```

        self.states: list[MagicCube] = [initial_cube]
        self.data_per_iteration: list[float] = []
        self.stuck_frequency: int = 0

    def simulated_annealing(self):
        """
        Execute the Algorithm
        """
        if not self.cube.is_perfect():
            while self.time <= self.MAX_TIME:
                neighbor, neighbor_objective =
                self.__get_random_neighbor()
                if neighbor.is_perfect():
                    self.states.append(neighbor)
                    break
                if neighbor_objective > self.objective:
                    self.states.append(neighbor)
                    self.cube = neighbor
                    self.objective = neighbor_objective
                else:
                    delta_E = neighbor_objective - self.objective
                    if self.__accept_by_probability(delta_E,
                        self.time):
                        if delta_E < 0:
                            self.states.append(neighbor)
                            self.stuck_frequency += 1
                            self.cube = neighbor
                            self.objective = neighbor_objective
                            temperature: float =
                            self.INITIAL_TEMPERATURE / math.log(self.time + 1)
                            probability: float = math.exp(delta_E /
                            temperature)
                            self.data_per_iteration.append(probability)
                            self.time += 1
            return

    def get_states(self) -> list[MagicCube]:
        return self.states

    def get_probability_per_iteration(self) -> list[float]:
        return self.data_per_iteration

    def get_stuck_frequency(self) -> int:
        return self.stuck_frequency

    # -- INTERNAL FUNCTION --

    def __get_random_neighbor(self) -> [MagicCube, int]:
        """
        Returns a random neighbor and it's value
        """
        i = random.randint(0, self.size ** 3 - 1)
        j = random.randint(i, self.size ** 3 - 1)
        neighbor: MagicCube = self.cube.swap_index_copy(i, j)
        neighbor_objective: int =
        ObjectiveFunction.get_object_value(neighbor)
        return [neighbor, neighbor_objective]

```

```

def __accept_by_probability(self, delta_e, time) -> bool:
    temperature: float = self.INITIAL_TEMPERATURE /
math.log(time + 1)
    probability: float = math.exp(delta_e / temperature)
    rnd: float = random.random()
    return rnd < probability

```

### 2.2.8. Genetic Algorithm

Untuk mendapatkan state value dari masing-masing state berikut adalah implementasi dari fungsi objektif untuk magic cube.

| Class                        |   |
|------------------------------|---|
| class GeneticAlgorithm       | Class sebagai implementasi dari Genetic Algorithm.  |
| Attributes                   |   |
| states (array of magic cube) | Menyimpan state-state untuk setiap iterasi mulai dari initial state hingga final state yang dihasilkan. Index 0 merupakan initial state, Index len-1 merupakan final state. |
| population                   | Menyimpan populasi berupa magic cube dan state valuenya   |
| MAX_ITERATION                | Jumlah iterasi maksimum   |
| MUTATION_CHANCE              | Kemungkinan terjadinya mutasi   |
| Fungsi dan Prosedur          |   |
| genetic_algorithm            | Melakukan local search dengan algoritma Genetic Algorithm   |
| __create_population          | Membuat populasi  |
| __sort_population()          | Menyortir populasi  |
| __get_parents()              | Mendapatkan parent dari populasi saat ini   |
| __reproduce(a,b)             | Menghasilkan anak dari dua buah parent magic cube   |
| __mutate(magic_cube)         | Melakukan mutasi pada magic cube  |

Berikut merupakan potongan Source Code untuk Class tersebut. Perlu diperhatikan bahwa kami hanya memberikan source code untuk bagian yang dianggap penting saja.

```

class GeneticAlgorithm:

    def genetic_algorithm(self):
        self.population = self.__create_population()
        for p in self.population:
            if p[1] == self.TARGET_VALUE:
                self.states.append(p[0])
                return
        best = 0
        while self.iteration <= self.MAX_ITERATION:
            self.__sort_population()
            self.range = self.__create_range()
            children: list[[MagicCube, int]] = []
            for i in range(self.POPULATION_COUNT):
                parent_one, parent_two = self.__get_parents()
                child_one, child_two = self.__reproduce(parent_one,
parent_two)
                self.__mutate(child_one)
                self.__mutate(child_two)
                children.append([child_one,
ObjectiveFunction.get_object_value(child_one)])
                children.append([child_two,
ObjectiveFunction.get_object_value(child_two)])
                for c in children:
                    if c[1] == self.TARGET_VALUE:
                        self.states.append(c[0])
                        return
            best = 0
            best_instance = None
            total = 0
            for c in children:
                total += c[1]
                if c[1] > best:
                    best_instance = c[0]
                    best = c[1]
            self.average = total / self.POPULATION_COUNT
            self.states.append(best_instance)
            self.population = children
            # print(best)
            self.iteration += 1
        print(best)
        return

    def __create_population(self) -> list[[MagicCube, int]]:
        population_temp = [MagicCube() for _ in
range(self.POPULATION_COUNT)]
        ret = []
        for m in population_temp:
            ret.append([m, ObjectiveFunction.get_object_value(m)])
        return ret

    def __sort_population(self):

```

```

        return sorted(self.population, key=lambda x: x[1],
reverse=True)

    def __create_range(self) -> list[[float, float]]:
        total = 0
        i = 0
        ret = []
        for p in self.population:
            total += p[1]
        for p in self.population:
            ret.append([i, i + (100 * p[1] / total)])
            i += (100 * p[1] / total)
        return ret

    def __get_parents(self) -> tuple[MagicCube, MagicCube]:
        rnd = random.random() * 100
        parent_one, parent_two = None, None
        for i in range(len(self.range)):
            r = self.range[i]
            if r[0] <= rnd <= r[1]:
                parent_one = self.population[i][0]

        rnd = random.random() * 100
        for i in range(len(self.range)):
            r = self.range[i]
            if r[0] <= rnd <= r[1]:
                parent_two = self.population[i][0]

        return parent_one, parent_two

    def __reproduce(self, a: MagicCube, b: MagicCube) ->
tuple[MagicCube, MagicCube]:
        crossover_point = random.randint(1, self.cube_size**3-1)
        data_one = np.concatenate((a.data[:crossover_point],
b.data[crossover_point:]))

        data_two = np.concatenate((b.data[:crossover_point],
a.data[crossover_point:]))

        child_one = MagicCube(self.cube_size, data_one)
        child_two = MagicCube(self.cube_size, data_two)
        return child_one, child_two

    def __mutate(self, m: MagicCube):
        if random.random() < self.MUTATION_CHANCE:
            i = random.randint(0, self.cube_size**3-1)
            j = random.randint(i, self.cube_size**3-1)
            m.swap_index(i, j)
        return

```

## BAB 3

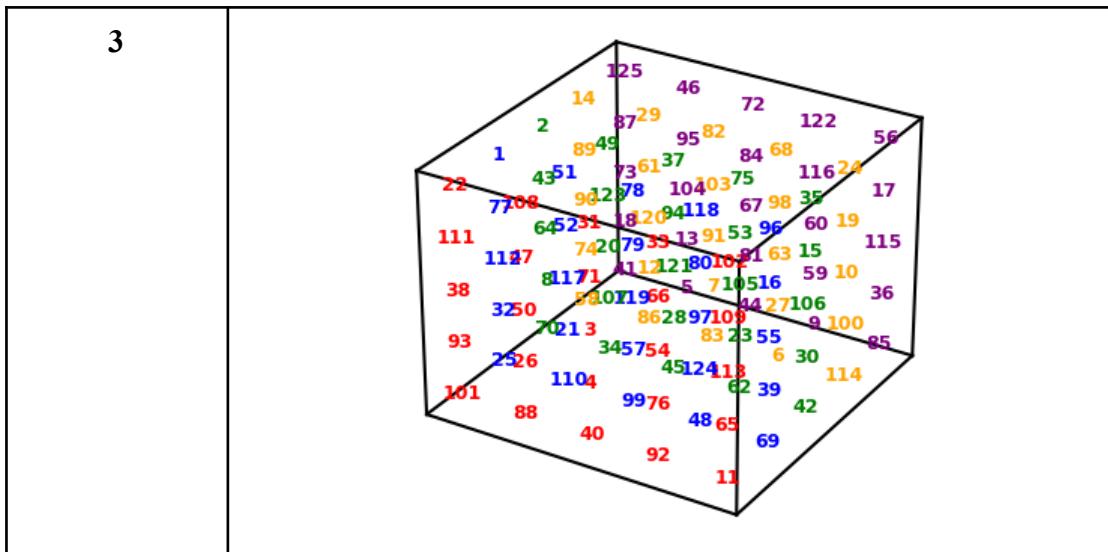
# HASIL DAN ANALISIS

### 3.1. Hasil Eksperimen

#### 3.1.1. Hill-Climbing Steepest Ascent

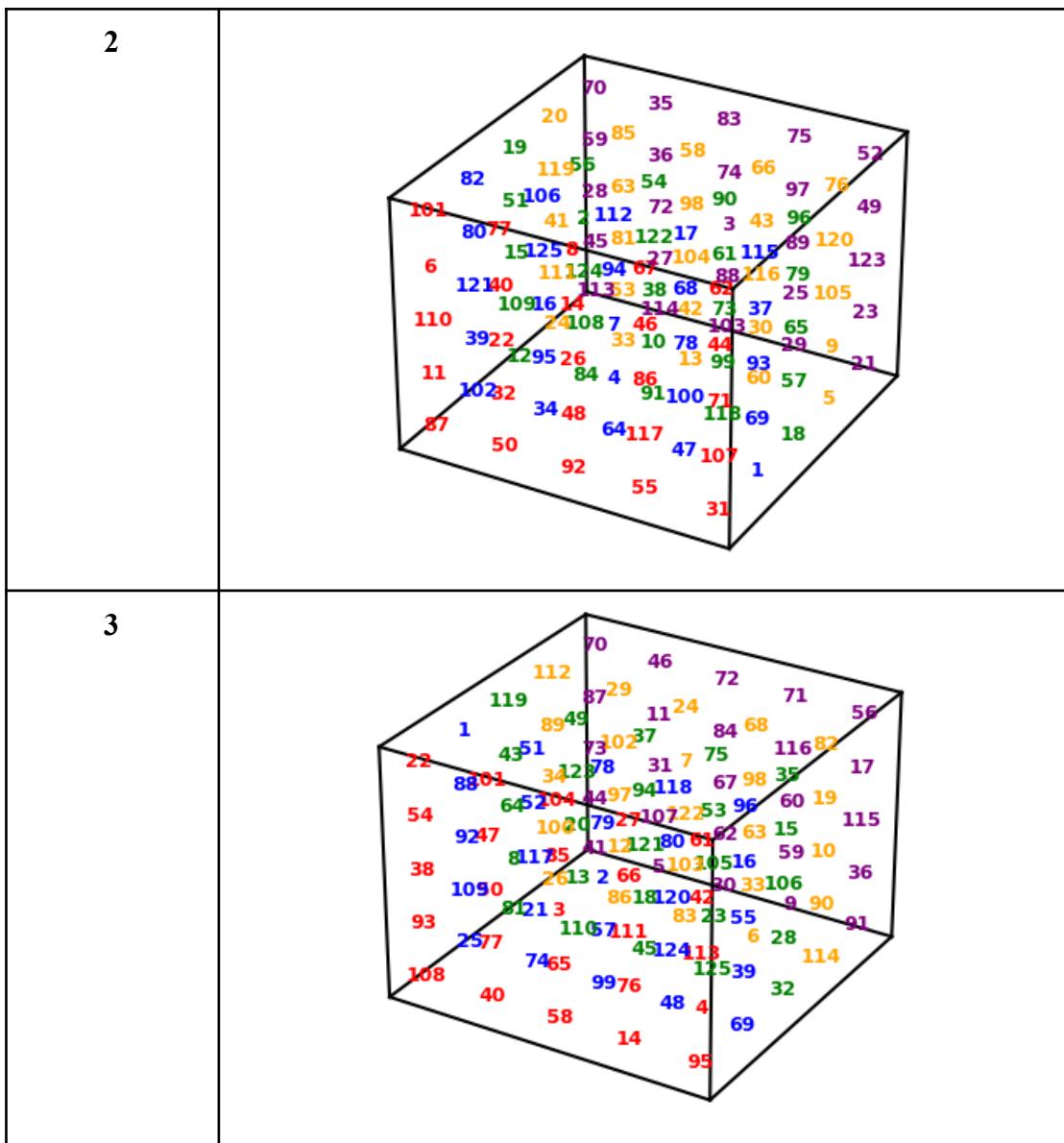
Pada percobaan Hill Climbing Steepest Ascent tidak ada parameter yang ditambahkan atau diubah. Pada proses pencarian ini dilakukan tiga kali percobaan dengan hasil state awal sebagai berikut.

| Percobaan | Representasi State Awal 3D |
|-----------|----------------------------|
| 1         |                            |
| 2         |                            |



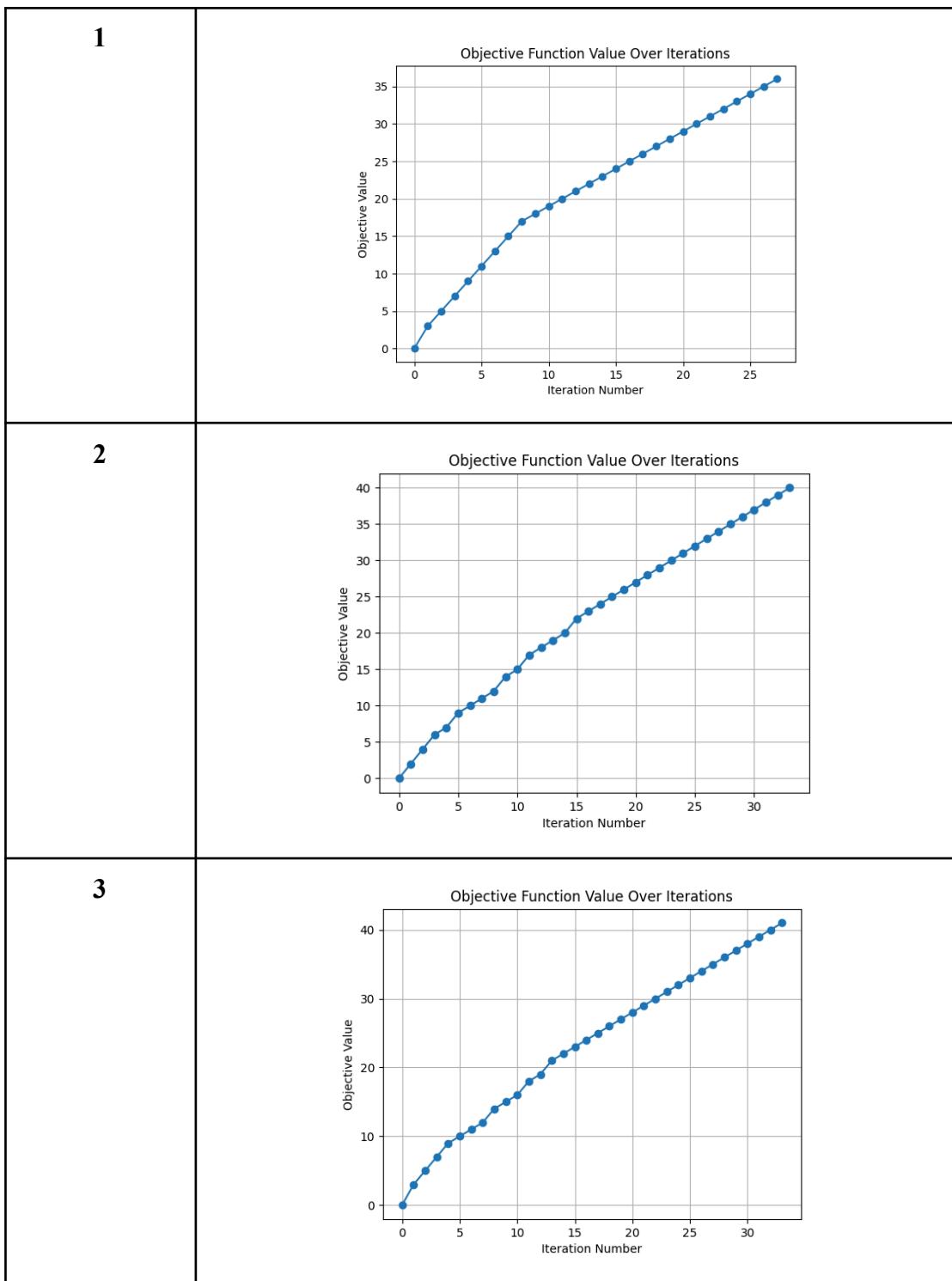
Setelah dilakukan *running*, dihasilkan state akhir kubus dan banyak iterasi yang telah diperoleh sebagai berikut.

| Percobaan | Representasi State Akhir 3D |
|-----------|-----------------------------|
| 1         |                             |



Kemudian, diperoleh grafik nilai *state value* setiap iterasi pada masing-masing percobaan sebagai berikut.

| Percobaan | Grafik Nilai State Value pada Setiap Iterasi |
|-----------|--|
|-----------|--|



Selain itu, diperoleh durasi, banyak iterasi yang dilakukan, dan nilai state akhir yang diperoleh sebagai berikut.

| Parameter  | Percobaan 1 | Percobaan 2 | Percobaan 3 |
|------------|-------------|-------------|-------------|
| Durasi (s) | 201         | 241         | 248         |

|                   |    |    |    |
|-------------------|----|----|----|
| Banyak Iterasi    | 27 | 33 | 33 |
| Final State Value | 36 | 40 | 41 |

Berdasarkan hasil percobaan, terlihat bahwa semakin lama durasi atau semakin banyak iterasi, hasil final state value akan semakin baik. Pada percobaan 1, nilai final state yang dihasilkan adalah 36 dan diperoleh dengan durasi 201 detik dan iterasi sebanyak 27 kali. Pada percobaan 2, nilai final state yang dihasilkan adalah 40 yang diperoleh dengan durasi pencarian mencapai 241 detik dan iterasi sebanyak 33 kali. Sedangkan, pada percobaan 3, nilai final state yang dihasilkan adalah 41 yang diperoleh dengan durasi pencarian mencapai 248 detik dan iterasi sebanyak 33 kali.

Percobaan 1 memiliki durasi dan iterasi yang paling sedikit daripada percobaan lain, tetapi nilai final state yang dihasilkan juga lebih kecil daripada percobaan yang lain. Perbedaan antara percobaan 2 dan 3 terdapat pada durasi dan nilai final state-nya. Pada jumlah iterasi yang sama, percobaan 3 menghasilkan nilai final state lebih baik daripada percobaan 2, walaupun memiliki durasi yang lebih lama. Hal ini menunjukkan bahwa pencarian solusi optimum lokal menggunakan algoritma ini bergantung pada state awal. Selain itu, penggunaan algoritma ini juga menunjukkan bahwa semakin banyak iterasi/durasi, nilai final state yang dihasilkan akan semakin baik. Hal ini dapat terjadi karena algoritma ini berhenti ketika state neighbour yang didapat memiliki nilai state yang kurang dari atau sama dengan nilai state sebelumnya.

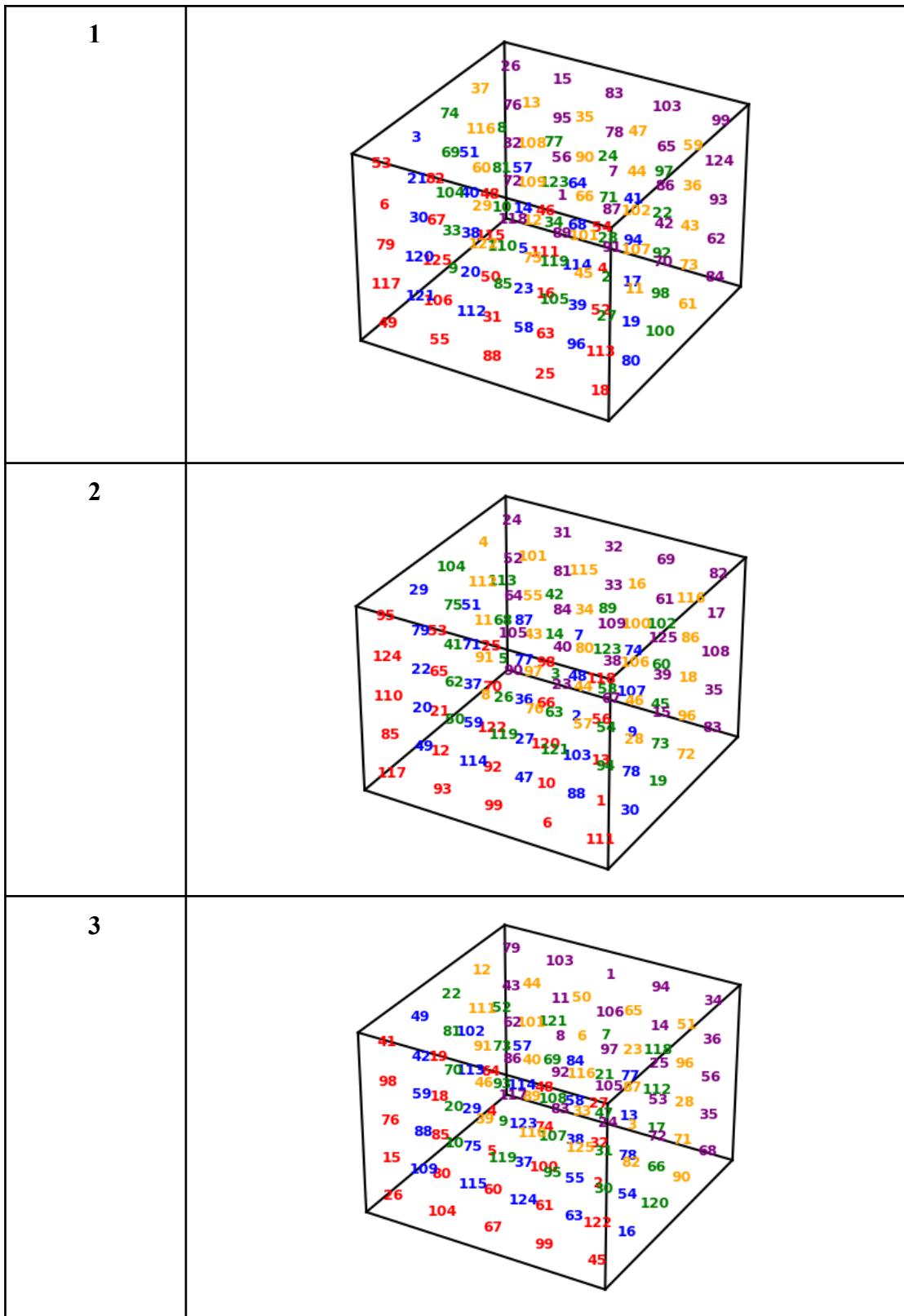
### 3.1.2. Hill-Climbing with Sideways Move

Pada percobaan Hill Climbing with Sideways Move terdapat parameter yang dapat diubah, yaitu `max_sides`, yang mengatur iterasi *sideways* maksimum ketika state value-nya sama berturut-turut pada setiap iterasi. Berikut merupakan konfigurasi dari masing-masing percobaan yang dilakukan.

| Parameter              | Percobaan 1 | Percobaan 2 | Percobaan 3 |
|------------------------|-------------|-------------|-------------|
| <code>max_sides</code> | 10          | 25          | 50          |

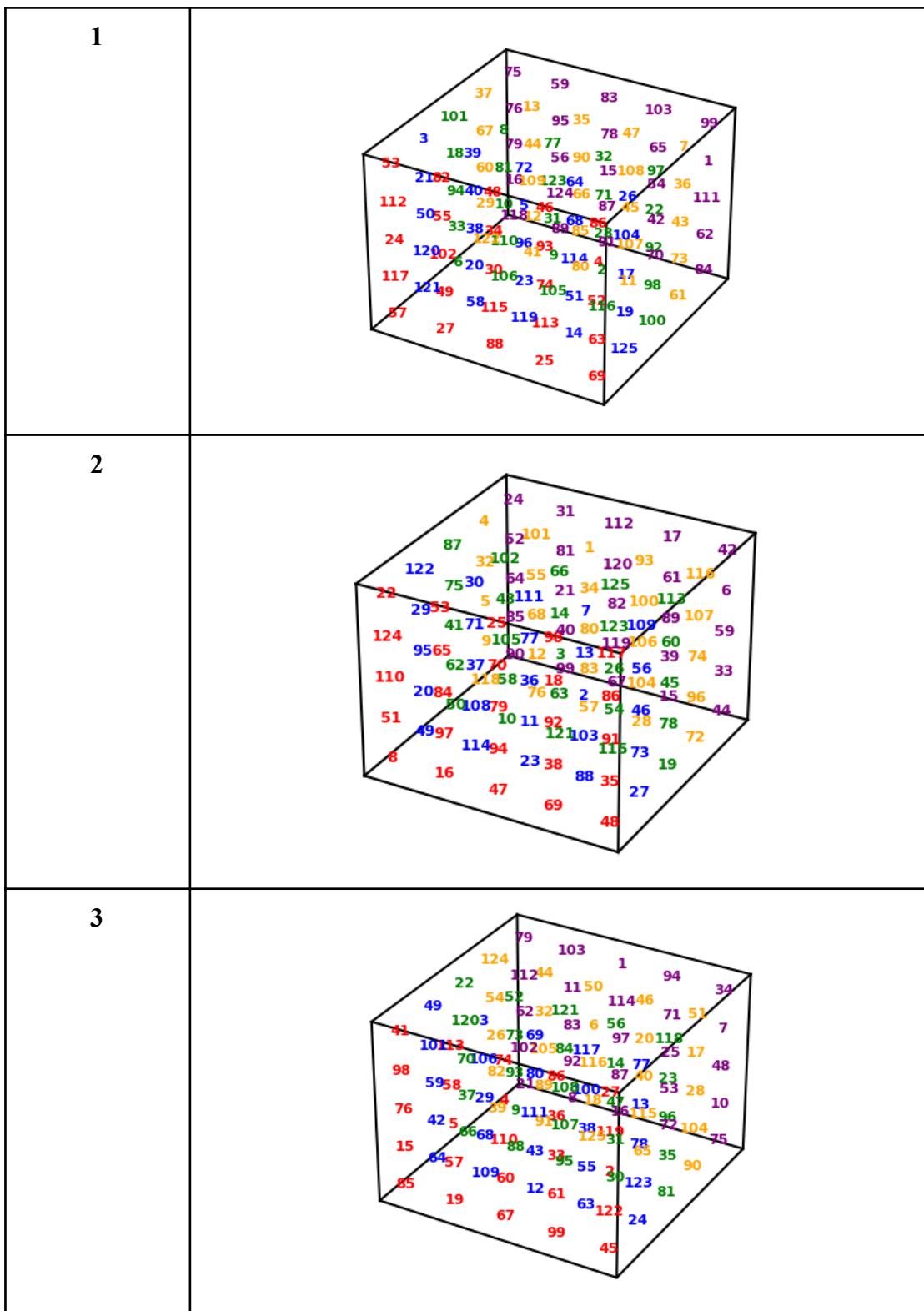
Setelah itu, diperoleh *state* awal dari masing-masing percobaan sebagai berikut.

| Percobaan | Representasi State Awal 3D |
|-----------|----------------------------|
|-----------|----------------------------|



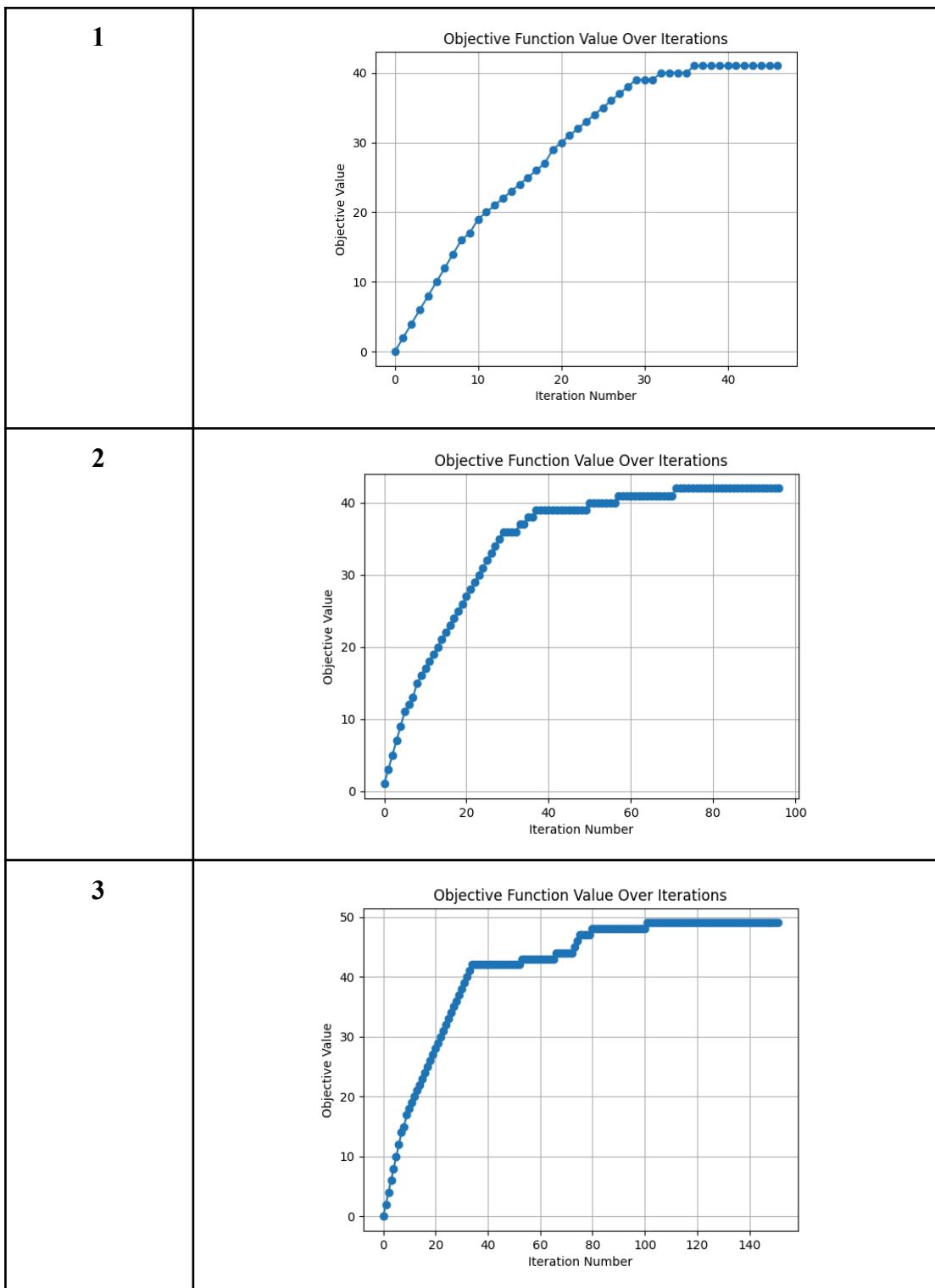
Setelah dilakukan *running*, dihasilkan state akhir kubus dan banyak iterasi yang telah diperoleh sebagai berikut.

| Percobaan | Representasi State Akhir 3D |
|-----------|-----------------------------|
|-----------|-----------------------------|



Kemudian, diperoleh grafik nilai *state value* setiap iterasi pada masing-masing percobaan sebagai berikut.

| Percobaan | Grafik Nilai State Value pada Setiap Iterasi |
|-----------|--|
|-----------|--|



Selain itu, diperoleh durasi, banyak iterasi yang dilakukan (termasuk iterasi sideways), dan nilai state akhir yang diperoleh sebagai berikut.

| Parameter  | Percobaan 1 | Percobaan 2 | Percobaan 3 |
|------------|-------------|-------------|-------------|
| Durasi (s) | 704         | 1458        | 2157        |

|                   |    |    |     |
|-------------------|----|----|-----|
| Banyak Iterasi    | 46 | 96 | 151 |
| Final State Value | 41 | 42 | 49  |

Berdasarkan hasil percobaan, terlihat bahwa pencarian solusi magic cube menggunakan hill climbing with sideways move memiliki nilai final state yang hampir sama. Pada percobaan 1, nilai final state yang dihasilkan adalah 41 dengan durasi pencarian mencapai 704 detik dan iterasi sebanyak 46 kali, serta iterasi sideways maksimum sebanyak 10 kali. Pada percobaan 2, nilai final state yang dihasilkan adalah 42 dengan durasi pencarian mencapai 1458 detik dan iterasi sebanyak 96 kali, serta iterasi sideways maksimum sebanyak 25 kali. Sedangkan, pada percobaan 3, nilai final state yang dihasilkan adalah 49 yang diperoleh dengan durasi pencarian mencapai 2157 detik dan iterasi sebanyak 151 kali, serta iterasi sideways maksimum sebanyak 50 kali. Ketiga percobaan tersebut menghasilkan nilai final state yang saling mendekati, tetapi nilainya masih jauh dari perfect cube (109). Hal ini terjadi karena pencarian solusi dengan algoritma ini dapat berpotensi terjebak di lokal optimum sehingga nilai final state yang didapatkan cenderung tidak meningkat di akhir iterasi.

Percobaan 1 memiliki durasi dan iterasi paling sedikit daripada percobaan lain. Sedangkan, percobaan 3 memiliki durasi dan iterasi yang lebih tinggi daripada percobaan lain. Nilai state value yang dihasilkan juga memiliki nilai yang paling kecil pada percobaan 1 dan nilai terbesar pada percobaan 3. Hal ini menunjukkan bahwa semakin banyak iterasi sideways move yang diperbolehkan, maka akan semakin besar kemungkinan untuk mendapatkan nilai final state yang paling mendekati maksimum (109). Berdasarkan hasil percobaan, semakin banyak iterasi sideways yang diizinkan, nilai final state yang dihasilkan juga akan semakin besar. Hal ini menandakan bahwa nilai final state berbanding lurus dengan iterasi sideways. Namun, semakin banyak iterasi sideways yang diizinkan, durasi untuk mencapai final state akan semakin lama.

### 3.1.3. Random Restart Hill-Climbing

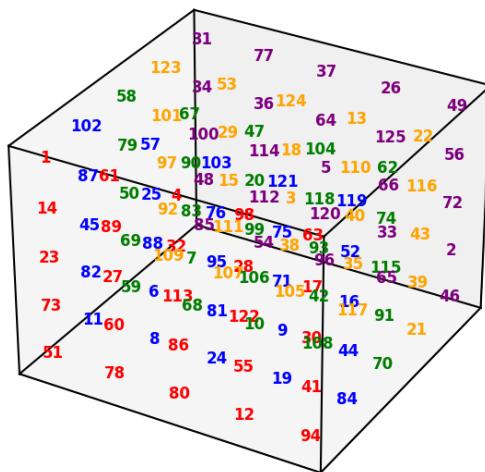
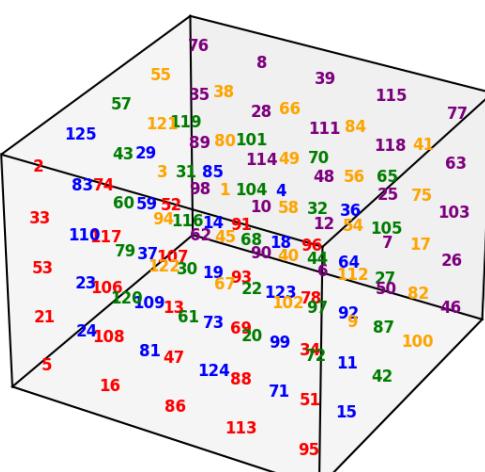
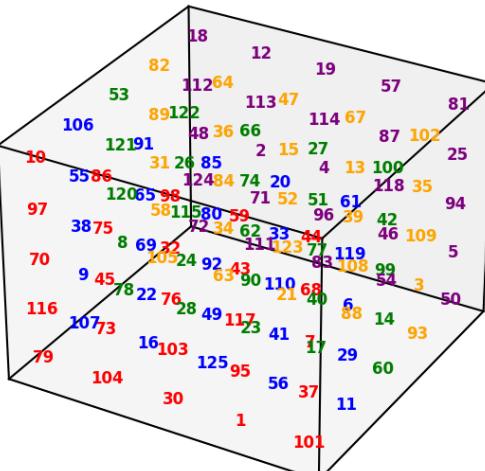
Pada bagian ini dilakukan percobaan beberapa konfigurasi untuk algoritma Hill Climbing with Random Restart. Algoritma ini memiliki 2 parameter utama yang dapat diubah yaitu `max_iterations` dan `max_restart`, kedua parameter ini masing-masing mengatur jumlah iterasi maksimum per restart dan jumlah restart maksimum. Berikut adalah konfigurasi dari beberapa percobaan yang dilakukan.

| Parameter                   | Percobaan 1 | Percobaan 2 | Percobaan 3 |
|-----------------------------|-------------|-------------|-------------|
| <code>max_iterations</code> | 100         | 100         | 20          |
| <code>max_restarts</code>   | 10          | 30          | 10          |

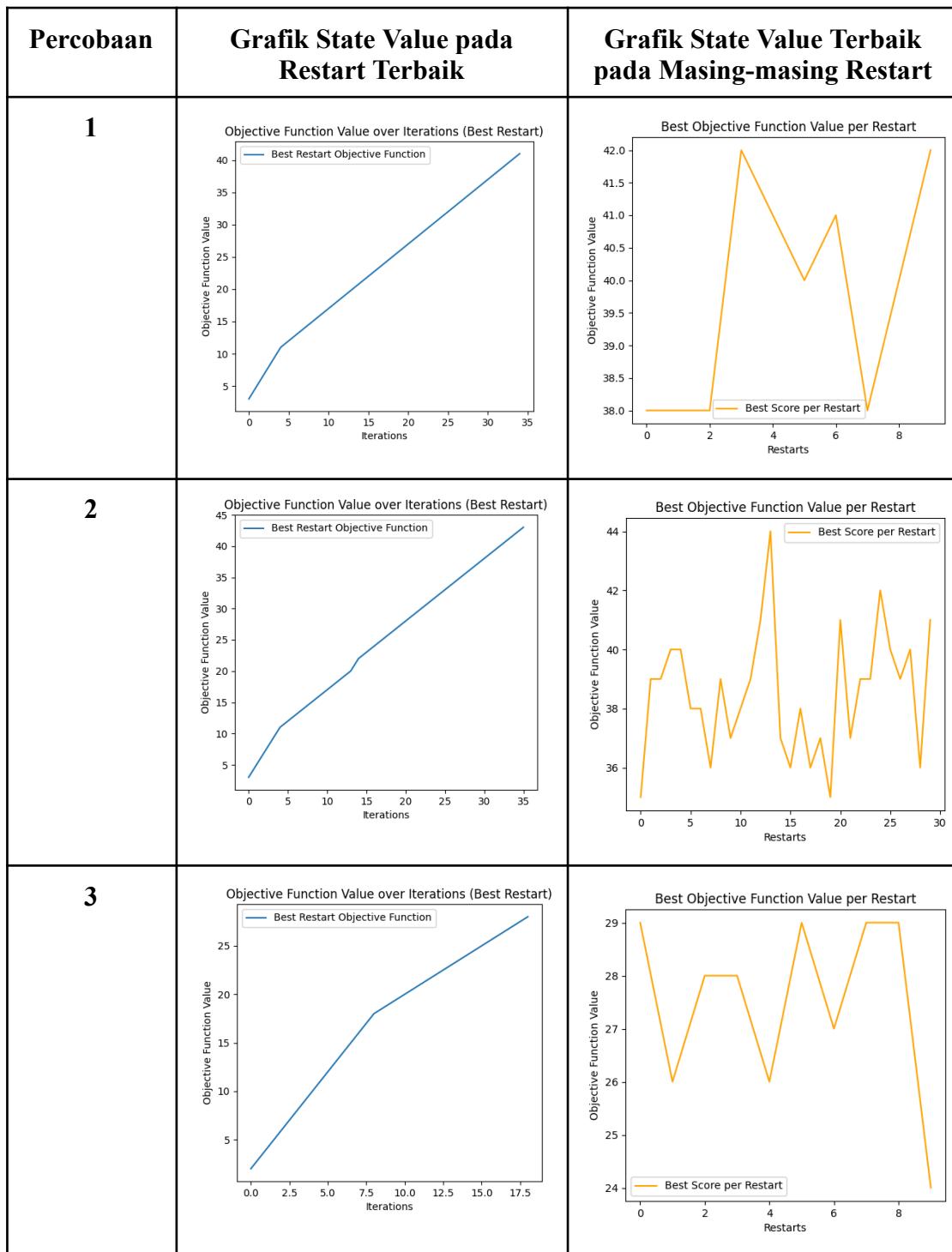
Kemudian dari masing-masing percobaan memiliki kondisi awal sebagai berikut

| Percobaan | Representasi State Awal 3D |
|-----------|----------------------------|
| 1         |                            |
| 2         |                            |
| 3         |                            |

Setelah dilakukan *running* program maka didapatkan kondisi akhir sebagai berikut

| Percobaan | Representasi State Akhir 3D  |
|-----------|--|
| 1         |    |
| 2         |   |
| 3         |  |

Kemudian grafik dari *state value* pada masing-masing percobaan sebagai berikut



Selain itu didapatkan pula hasil dari masing-masing percobaan sebagai berikut.

| Parameter  | Percobaan 1 | Percobaan 2 | Percobaan 3 |
|------------|-------------|-------------|-------------|
| Durasi (s) | 1941        | 4855        | 913         |

|                            |      |      |    |
|----------------------------|------|------|----|
| Rerata iterasi per restart | 33.1 | 31.7 | 20 |
| Final State Value          | 42   | 44   | 29 |

Berdasarkan hasil percobaan, konfigurasi parameter yang berbeda menunjukkan dampak yang signifikan terhadap durasi dan kualitas solusi yang dihasilkan oleh algoritma Hill Climbing dengan random restart. Pada hasil 1, dengan max\_restart sebesar 10 dan max\_iterations sebesar 100, algoritma mencapai nilai objektif akhir sebesar 42 dalam waktu 1941 detik. Rata-rata iterasi per restart adalah 33.1, menunjukkan bahwa algoritma membutuhkan rata-rata 33 iterasi pada setiap restart untuk mencapai solusi lokal terbaik. Hasil ini menunjukkan bahwa konfigurasi ini memberikan solusi yang baik dalam durasi yang moderat.

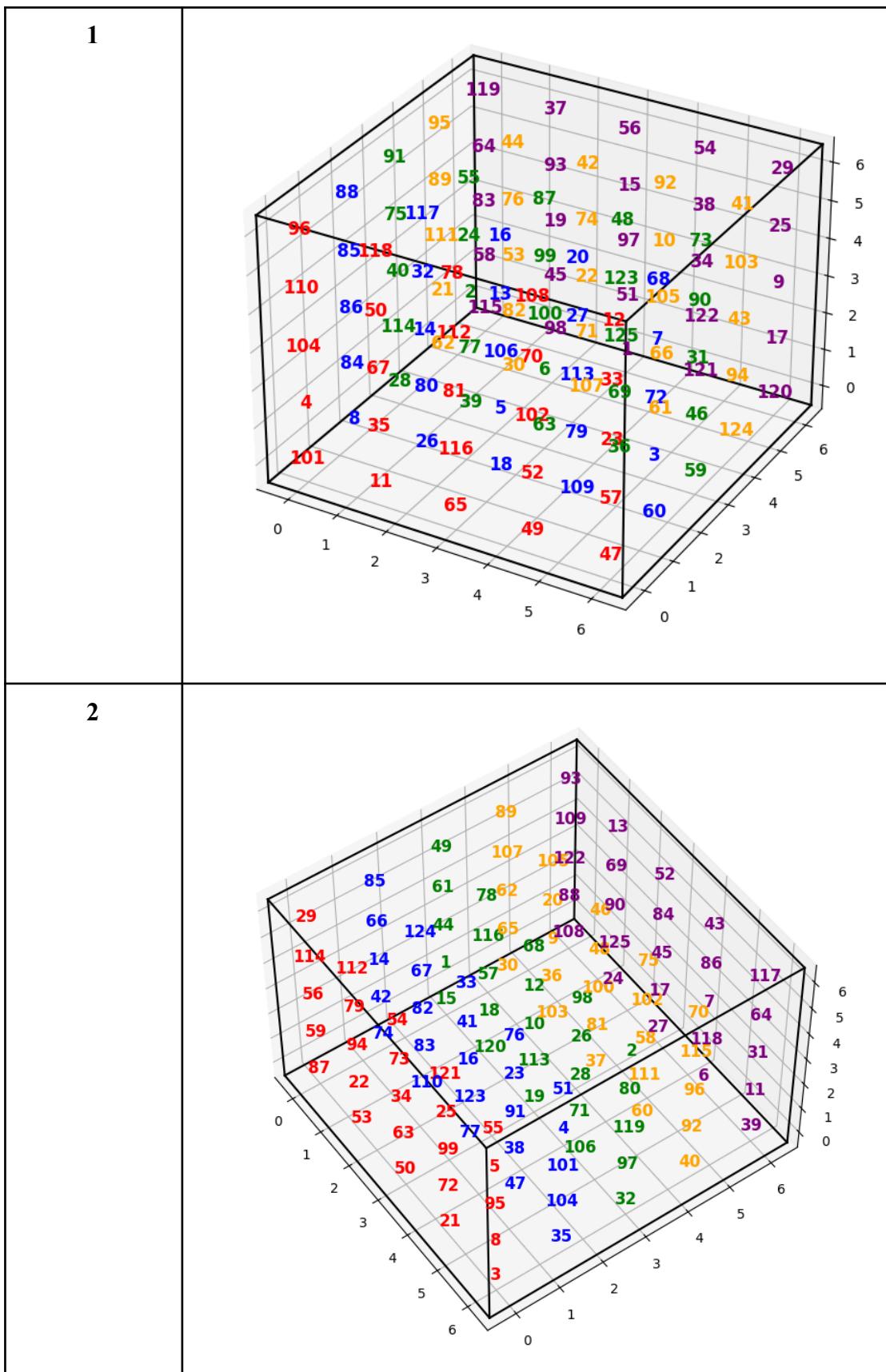
Pada hasil 2, dengan max\_restart yang ditingkatkan menjadi 30 dan max\_iterations tetap 100, durasi eksekusi meningkat secara signifikan menjadi 4855 detik, hampir tiga kali lipat dibandingkan hasil 1. Meski durasi eksekusi jauh lebih lama, algoritma mencapai nilai objektif akhir sebesar 44, yang lebih tinggi dibandingkan hasil 1 tetapi tidak signifikan. Peningkatan nilai objektif ini menunjukkan bahwa peningkatan jumlah restart memberi algoritma lebih banyak kesempatan untuk menemukan solusi yang lebih baik. Namun, peningkatan kualitas solusi ini dicapai dengan mengorbankan durasi yang jauh lebih lama.

Pada hasil 3, max\_iterations dikurangi menjadi 20, sementara max\_restart tetap 10. Hasilnya, durasi eksekusi turun drastis menjadi 913 detik, yang merupakan durasi tercepat di antara ketiga konfigurasi. Namun, nilai objektif akhir turun menjadi 29, yang menunjukkan bahwa solusi yang dihasilkan kurang optimal dibandingkan hasil 1 dan hasil 2. Ini menunjukkan bahwa pengurangan jumlah iterasi dapat membuat algoritma berhenti terlalu cepat sebelum menemukan solusi yang lebih baik, mengakibatkan algoritma cenderung terjebak pada solusi lokal yang kurang optimal.

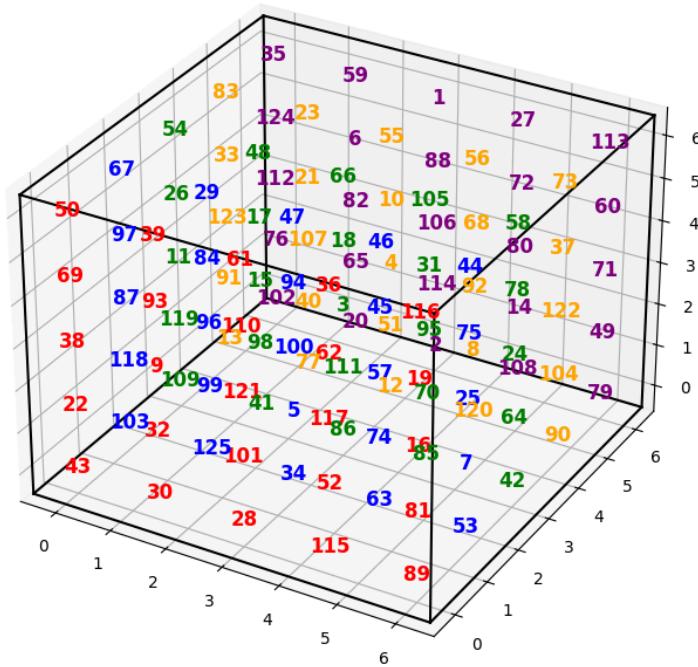
### 3.1.4. Stochastic Hill-Climbing

Pada bagian ini dilakukan percobaan beberapa konfigurasi untuk algoritma Stochastic Hill Climbing. Untuk jumlah iterasi maksimum yang diset adalah 10.000.

| Percobaan | Representasi State Awal 3D |
|-----------|----------------------------|
|-----------|----------------------------|

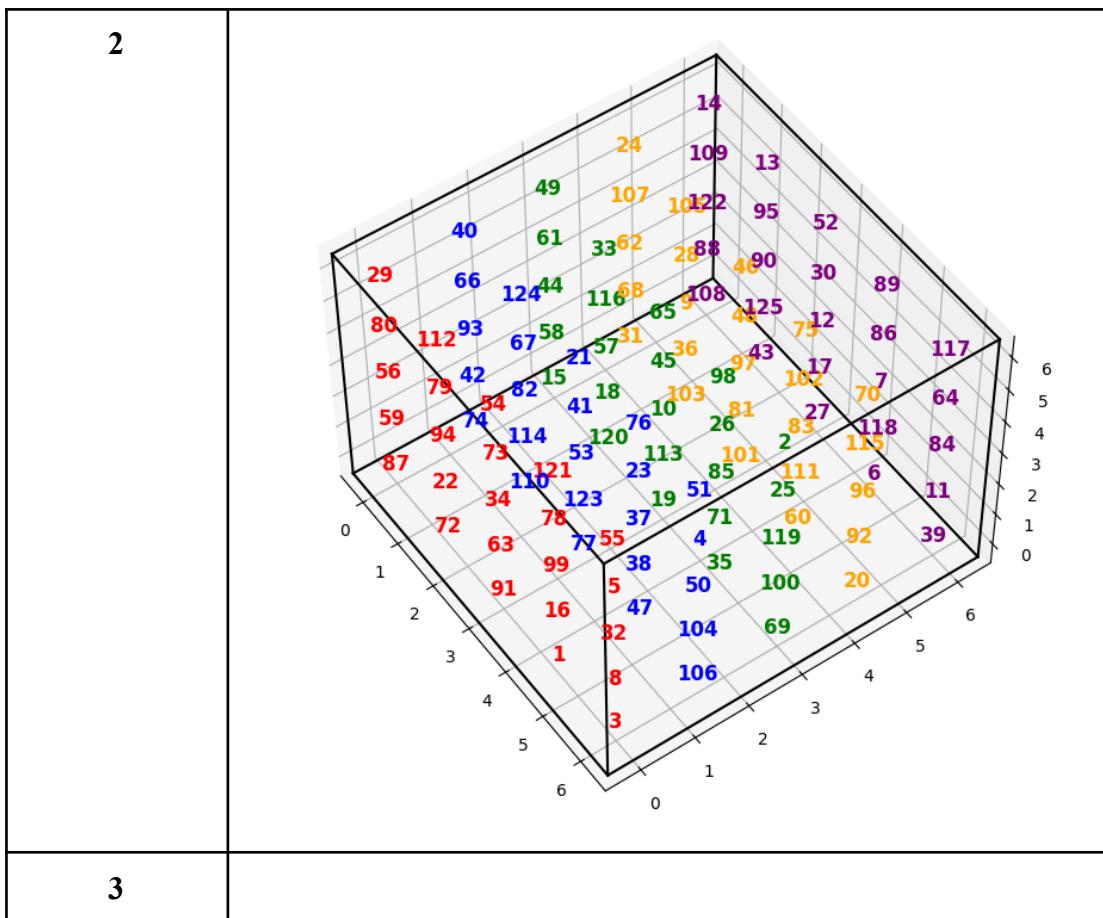


3

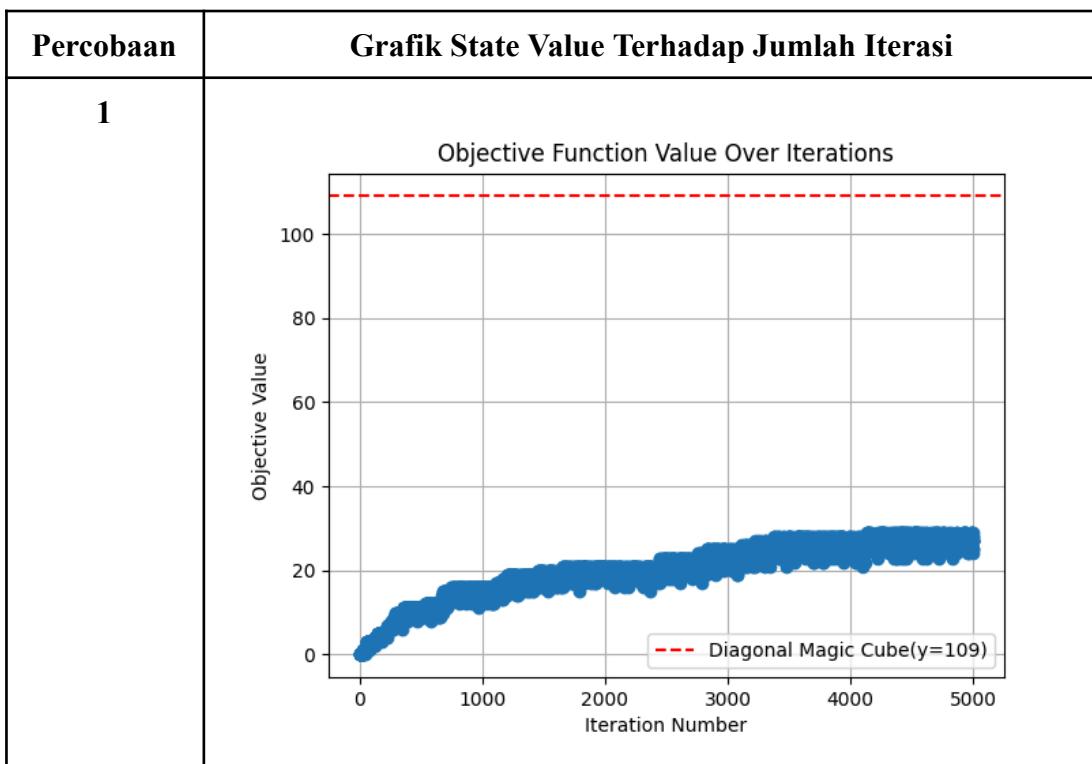


Setelah dilakukan *running* program maka didapatkan kondisi akhir sebagai berikut

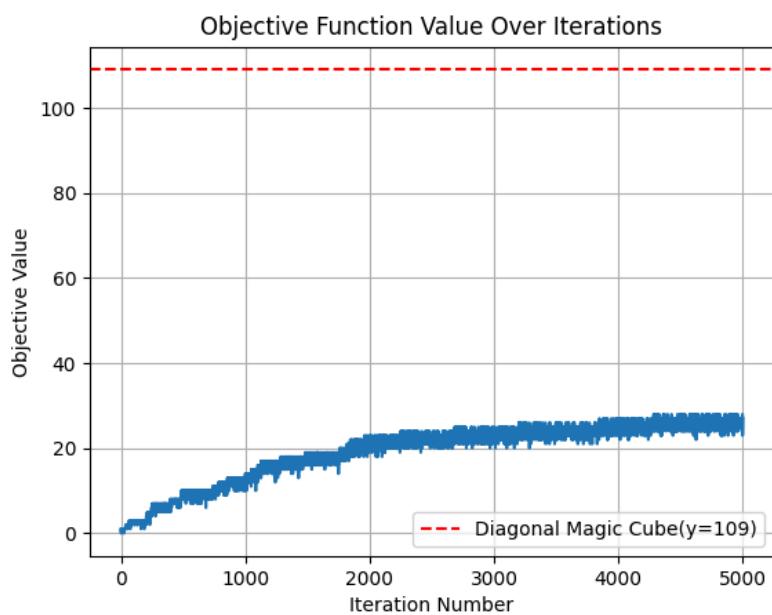
| Percobaan | Representasi State Akhir 3D   |
|-----------|---|
| 1         | <p>A 3D plot showing the final state representation after running the program. The axes range from 0 to 6. The plot contains colored numbers representing states. Compared to the initial state, many values have changed, indicating the progression of the algorithm.</p> |



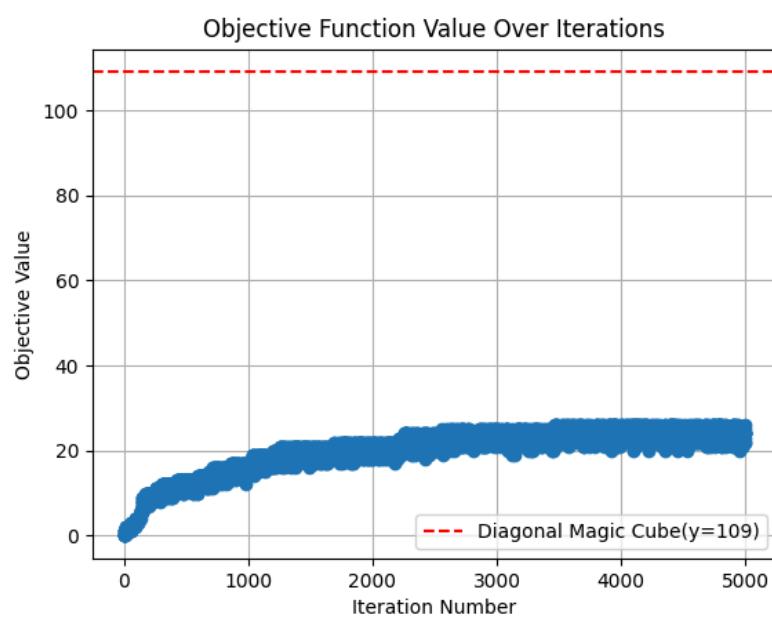
Kemudian grafik dari *state value* pada masing-masing percobaan sebagai berikut



2



3



Berikut adalah state awal dan akhir dari masing-masing percobaan

| Perco<br>baan | Initial State   | Final State   |
|---------------|---|---|
| 1             | [101 4 104 110 96 11 35 67 50 118<br>65 116 81 112 78 49 52 102<br>70 108 47 57 23 33 12 8 84 86<br>85 88 26 80 14 32 117 18] | [101 89 87 110 96 11 35 67 50 118<br>107 115 81 112 78 49 41 99<br>70 108 47 10 23 2 12 8 84 6 85<br>52 83 80 14 21 117 18] |

|          |   |   |
|----------|---|---|
|          | [ 5 106 13 16 109 79 113 27 20 60<br>3 72 7 68 28 114 40 75<br>91 39 77 2 24 55 63 6 100 99<br>87 36 69 125 123 48 59 46<br>31 90 73 62 21 111 89 95 30 82<br>53 76 44 107 71 22 74 42<br>61 66 105 10 92 124 94 43 103 41<br>115 58 83 64 119 98 45 19<br>93 37 1 51 97 15 56 121 122 34<br>38 54 120 17 9 25 29]  | [ 5 106 13 16 109 79 113 27 36 30<br>3 72 95 94 28 114 7 75<br>91 63 73 33 24 120 39 86 100 102<br>104 20 92 125 123 48 59 71<br>31 90 77 62 22 111 4 40 60 82<br>53 76 44 65 46 32 74 61<br>29 43 105 57 69 124 122 66 103 88<br>116 58 26 64 119 98 45 42<br>93 37 1 51 97 15 56 121 68 34<br>38 54 55 17 9 25 19]  |
| <b>2</b> | [ 87 59 56 114 29 53 22 94 79 112<br>50 63 34 73 54 21 72 99<br>25 121 3 8 95 5 55 74 42 14 66<br>85 110 83 82 67 124 77<br>123 16 41 33 47 38 91 23 76 35<br>104 101 4 51 15 1 44 61<br>49 120 18 57 116 78 19 113 10 12<br>68 106 71 28 26 98 32 97<br>119 80 2 30 65 62 107 89 103 36<br>9 20 105 37 81 100 48 46<br>60 111 58 102 75 40 92 96 115 70<br>108 88 122 109 93 24 125 90<br>69 13 27 17 45 84 52 6 118 7<br>86 43 39 11 31 64 117] | [ 87 59 56 80 29 72 22 94 79 112<br>91 63 34 73 54 1 16 99<br>78 121 3 8 32 5 55 74 42 93 66<br>40 110 114 82 67 124 77<br>123 53 41 21 47 38 37 23 76 106<br>104 50 4 51 15 58 44 61<br>49 120 18 57 116 33 19 113 10 45<br>65 35 71 85 26 98 69 100<br>119 25 2 31 68 62 107 24 103 36<br>9 28 105 101 81 97 48 46<br>60 111 83 102 75 20 92 96 115 70<br>108 88 122 109 14 43 125 90<br>95 13 27 17 12 30 52 6 118 7<br>86 89 39 11 84 64 117] |
| <b>3</b> | [ 43 22 38 69 50 30 32 9 93 39<br>28 101 121 110 61 115 52 117<br>62 36 89 81 16 19 116 103 118 87<br>97 67 125 99 96 84 29 34<br>5 100 94 47 63 74 57 45 46 53<br>7 25 75 44 109 119 11 26<br>54 41 98 15 17 48 86 111 3 18<br>66 85 70 95 31 105 42 64<br>24 78 58 13 91 123 33 83 77 40<br>107 21 23 12 51 4 10 55<br>120 8 92 68 56 90 104 122 37 73<br>102 76 112 124 35 20 65 82<br>6 59 2 114 106 88 1 108 14 80<br>72 27 79 49 71 60 113] | [ 83 48 52 35 50 29 32 9 93 39<br>28 101 121 110 82 86 34 117<br>62 36 89 100 16 15 17 103 95 87<br>97 92 125 99 96 84 30 38<br>55 81 94 47 63 74 57 45 46 53<br>7 25 75 44 109 5 11 26<br>54 41 98 19 116 22 115 111 3 51<br>66 112 70 118 31 105 85 64<br>20 78 68 13 91 123 33 43 65 40<br>107 80 23 12 72 4 58 119<br>120 8 67 10 56 90 104 14 18 73<br>102 76 42 124 69 24 77 61<br>6 59 2 114 106 88 1 108 122 21<br>37 27 79 49 71 60 113] |

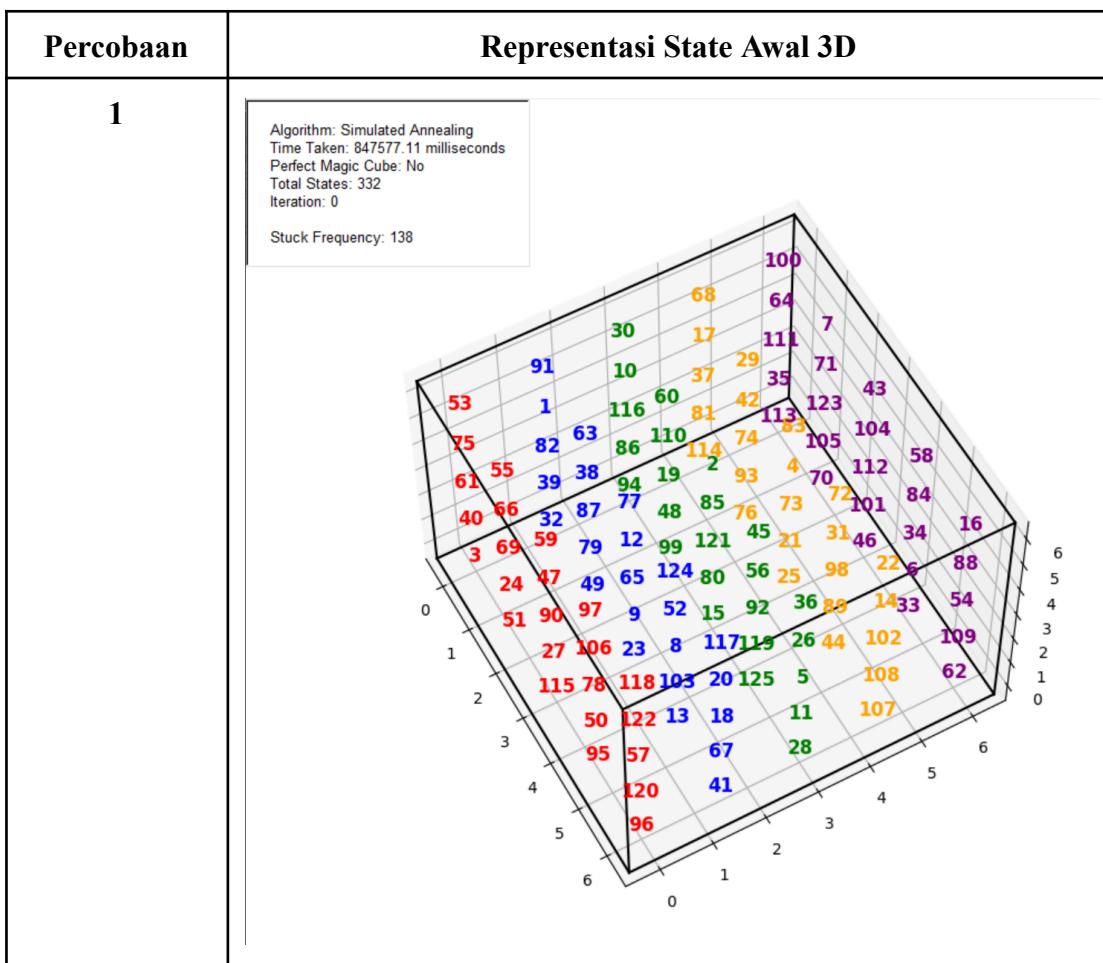
| Parameter      | Percobaan 1 | Percobaan 2 | Percobaan 3 |
|----------------|-------------|-------------|-------------|
| Durasi (s)     | 352.71015   | 311.48486   | 3649.12868  |
| Jumlah Iterasi | 5000        | 5000        | 5000        |

|                     |    |    |    |
|---------------------|----|----|----|
| Total States        | 29 | 28 | 27 |
| Initial State Value | 0  | 0  | 0  |
| Final State Value   | 29 | 28 | 26 |

Berdasarkan hasil eksperimen tersebut dapat dilihat bahwa peningkatan jumlah iterasi sebanding dengan durasi pencarian. Namun dapat dilihat bahwa solusi terjebak pada local optima yang masih jauh dari global optima. Hal ini dikarenakan pemilihan random neighbor yang masih kurang baik karena masih dapat terjebak pada local optima.

### 3.1.5. Simulated Annealing

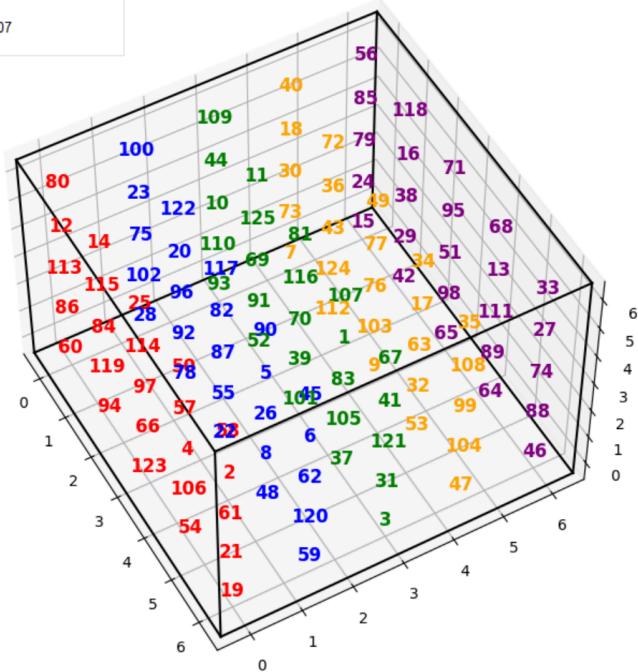
Pada percobaan Simulated Annealing tidak ada parameter yang ditambahkan atau diubah. Pada proses pencarian ini dilakukan tiga kali percobaan dengan hasil state awal sebagai berikut.



2

Algorithm: Simulated Annealing  
 Time Taken: 121645.87 milliseconds  
 Perfect Magic Cube: No  
 Total States: 273  
 Iteration: 0

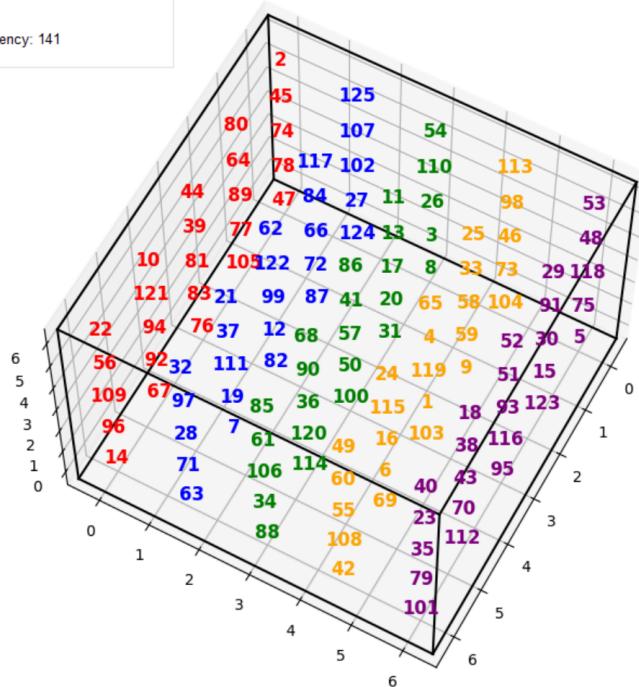
Stuck Frequency: 107



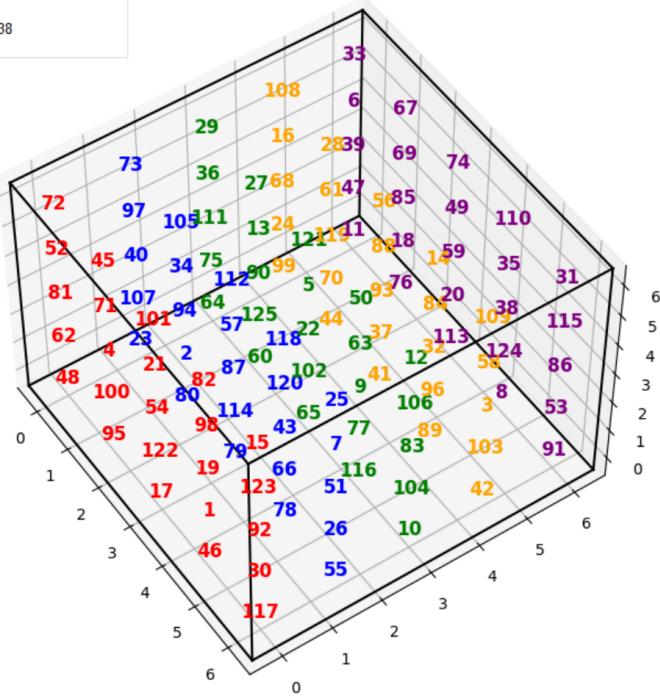
3

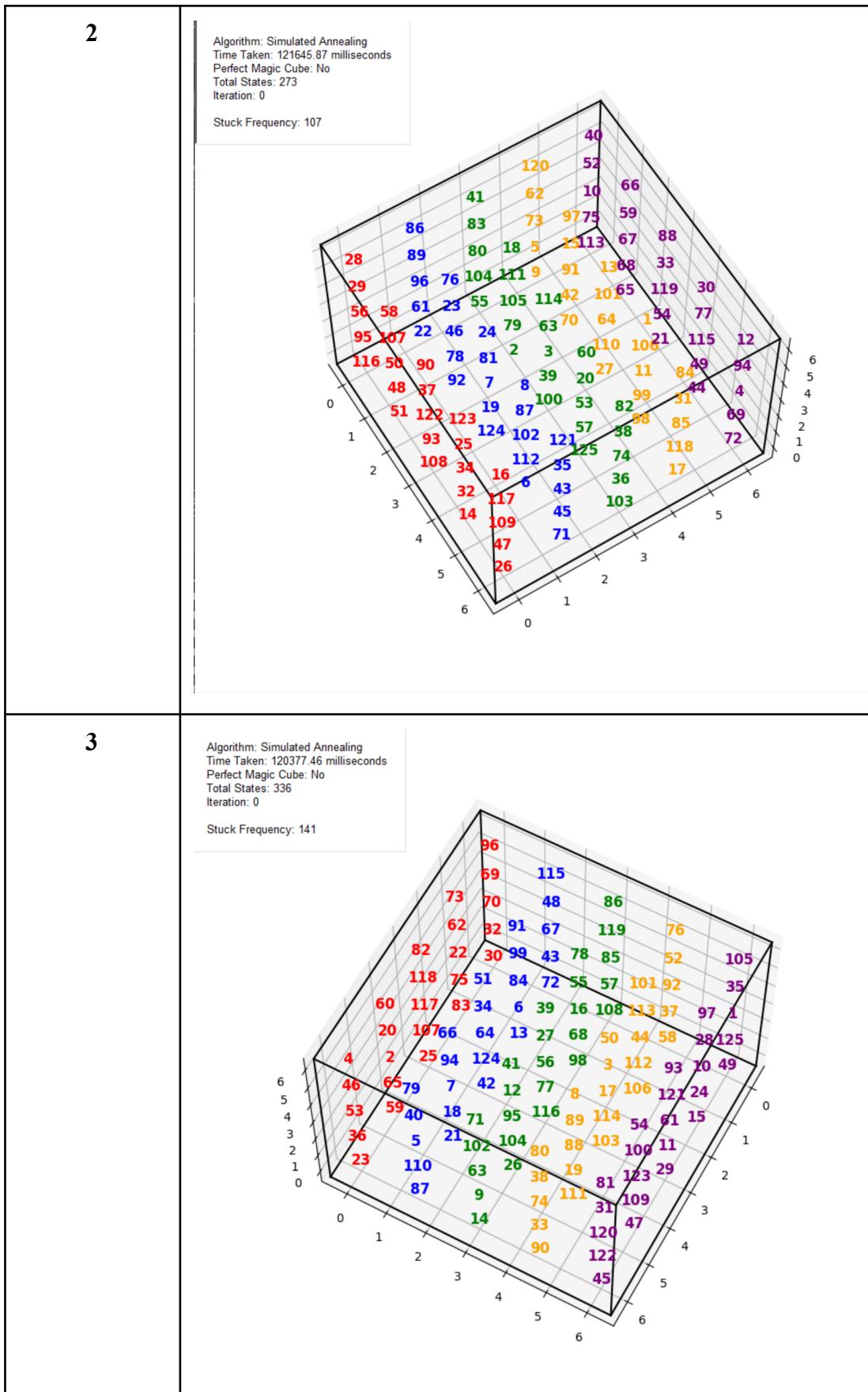
Algorithm: Simulated Annealing  
 Time Taken: 120377.46 milliseconds  
 Perfect Magic Cube: No  
 Total States: 336  
 Iteration: 0

Stuck Frequency: 141

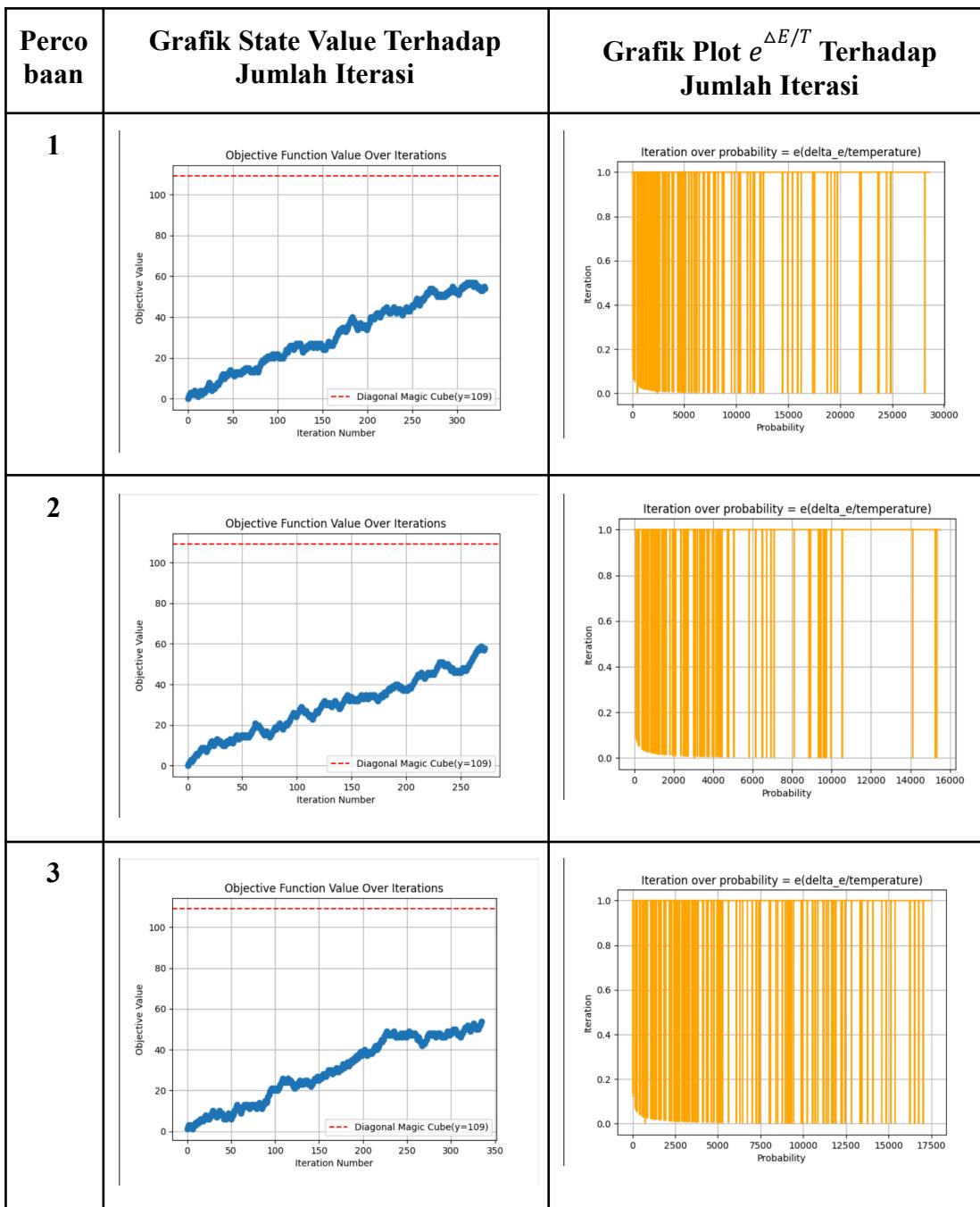


Setelah dilakukan *running* program maka didapatkan kondisi akhir sebagai berikut

| Percobaan | Representasi State Akhir 3D   |
|-----------|---|
| 1         | <p>Algorithm: Simulated Annealing<br/>Time Taken: 847577.11 milliseconds<br/>Perfect Magic Cube: No<br/>Total States: 332<br/>Iteration: 0<br/>Stuck Frequency: 138</p>  |



Kemudian grafik dari *state value* pada masing-masing percobaan sebagai berikut



Berikut adalah state awal dan akhir dari masing-masing percobaan

| Perco<br>baan | Initial State   | Final State  |
|---------------|---|--|
| 1             | 3 40 61 75 53 51 24 69 66 55 115<br>27 90 47 59 95 50 78<br>106 97 96 120 57 122 118 32 39 82<br>1 91 49 79 87 38 63 23 | 48 62 81 52 72 95 100 4 71 45 17<br>122 54 21 101 46 1 19<br>98 82 117 30 92 123 15 23 107 40<br>97 73 80 2 94 34 105 79 |

|          |   |  |
|----------|---|--|
|          | 9 65 12 77 13 103 8 52 124 41<br>67 18 20 117 94 86 116 10<br>30 99 48 19 110 60 15 80 121 85<br>2 125 119 92 56 45 28 11<br>5 26 36 114 81 37 17 68 76 93<br>74 42 29 25 21 73 4 83  | 114 87 57 112 78 66 43 120 118 55<br>26 51 7 25 64 75 111 36<br>29 60 125 90 13 27 65 102 22 5<br>121 116 77 9 63 50 10 104<br>83 106 12 99 24 68 16 108 44 70<br>119 61 28 41 37 93 88 56   |
| <b>2</b> | 60 86 113 12 80 94 119 84 115 14<br>123 66 97 114 25 54 106 4<br>57 50 19 21 61 2 58 28 102 75<br>23 100 78 92 96 20 122 22<br>55 87 82 117 48 8 26 5 90 59<br>120 62 6 45 93 110 10 44<br>109 52 91 69 125 11 101 39 70 116<br>81 37 105 83 1 107 3 31<br>121 41 67 7 73 30 18 40 112 124<br>43 36 72 9 103 76 77 49 | 116 95 56 29 28 51 48 50 107 58<br>108 93 122 37 90 14 32 34<br>25 123 26 47 109 117 16 22 61 96<br>89 86 92 78 46 23 76 124<br>19 7 81 24 6 112 102 87 8 71<br>45 43 35 121 55 104 80 83<br>41 2 79 105 111 18 100 39 3 63<br>114 125 57 53 20 60 103 36<br>74 38 82 9 5 73 62 120 70 42<br>91 15 97 27 110 64 101 13 |
| <b>3</b> | 47 78 74 45 2 105 77 89 64 80 76<br>83 81 39 44 67 92 94<br>121 10 14 96 109 56 22 124 27 102<br>107 125 87 72 66 84 117 82<br>12 99 122 62 7 19 111 37 21 63<br>71 28 97 32 8 3 26 110<br>54 31 20 17 13 11 100 50 57 41<br>86 114 120 36 90 68 88 34<br>106 61 85 104 73 46 98 113 9 59<br>58 33 25 103 1 119 4 65  | 30 32 70 69 96 83 75 22 62 73 25<br>107 117 118 82 59 65 2<br>20 60 23 36 53 46 4 72 43 67<br>48 115 13 6 84 99 91 42<br>124 64 34 51 21 18 7 94 66 87<br>110 5 40 79 108 57 85 119<br>86 98 68 16 55 78 116 77 56 27<br>39 26 104 95 12 41 14 9<br>63 102 71 58 37 92 52 76 106 112<br>44 113 101 103 114 17 3 50     |

| Parameter           | Percobaan 1 | Percobaan 2 | Percobaan 3 |
|---------------------|-------------|-------------|-------------|
| Durasi (s)          | 847.577     | 121.645     | 120.377     |
| Jumlah Iterasi      | 500000      | 250000      | 250000      |
| Total States        | 332         | 272         | 336         |
| Initial State Value | 0           | 0           | 0           |
| Final State Value   | 55          | 58          | 56          |

Berdasarkan hasil eksplorasi, bisa disimpulkan bahwa semakin banyaknya iterasi yang dilakukan maka semakin lama pula durasi pencarian. Pada implementasi penulis, algoritma Simulated annealing tidak berhasil menemukan solusi. Hal ini dikarenakan jumlah iterasi yang dibatasi akibat memakan waktu yang sangat banyak,

juga didukung oleh bahasa python yang kurang optimal dalam kecepatan. Tetapi, bisa dilihat bahwa solusi yang dihasilkan memiliki value yang cukup tinggi yaitu mencapai 58. Namun, semakin banyak iterasi tidak menjamin value akhir yang semakin besar. Hal ini dikarenakan pada iterasi dengan sebanyak itu probabilitas pemilihan stuck akan semakin mengecil, maka semakin sulit pula untuk keluar dari local maxima.

Seperti yang terlihat pada grafik, secara rata rata value akan selalu naik berdasarkan waktu. Ketiga grafik, terutama grafik ketiga, menunjukkan bahwa value masih mungkin turun. Tetapi, menuju akhir iterasi frekuensi penurunan semakin kecil dibandingkan pada awal iterasi. Grafik pada sebelah kanan menunjukkan perubahan probabilitas terhadap waktu. Nilai T yang digunakan penulis pada implementasi adalah logaritma, yang memastikan penurunan T yang cukup lamban. Hal ini terbukti pada grafik dimana penurunan probabilitas tidak terjadi secara drastis. Terdapat pula kasus dimana probabilitas menjadi 1 akibat  $\Delta E = 0$ .

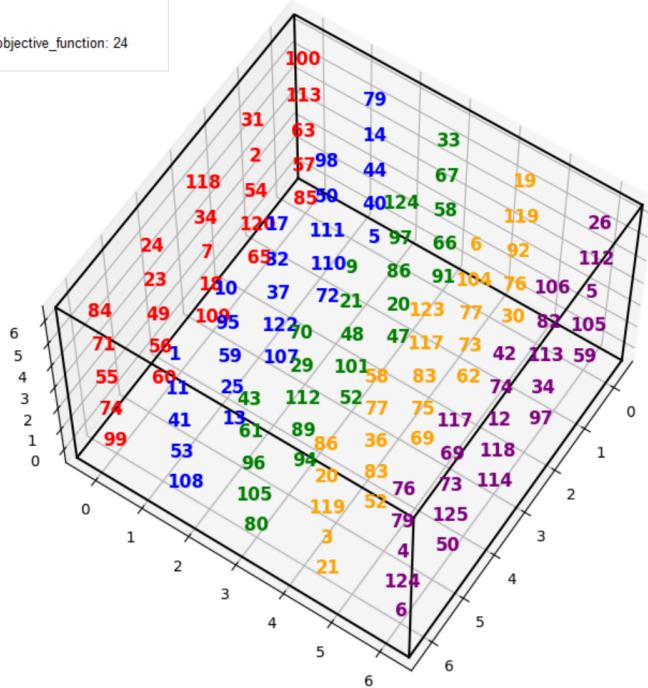
### 3.1.6. Genetic Algorithm

Pada algoritma ini, terdapat 6 buah percobaan. 3 diantaranya berdasarkan parameter banyak populasi, dan 3 diantaranya berdasarkan parameter banyak iterasi. Hasil dari percobaan adalah sebagai berikut.

| Percobaan | Representasi State Awal 3D  |
|-----------|---|
| 1         | <p>Algorithm: Genetic Algorithm<br/> Time Taken: 59310.62 milliseconds<br/> Perfect Magic Cube: No<br/> Total States: 200<br/> Iteration: 0<br/> Maximum_objective_function: 25</p> |

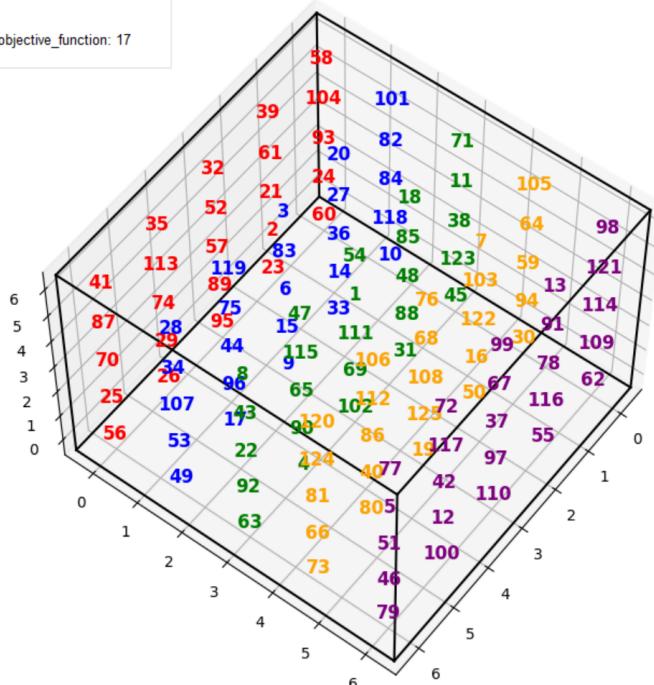
2

Algorithm: Genetic Algorithm  
 Time Taken: 37381.92 milliseconds  
 Perfect Magic Cube: No  
 Total States: 200  
 Iteration: 0  
 Maximum\_objective\_function: 24



3

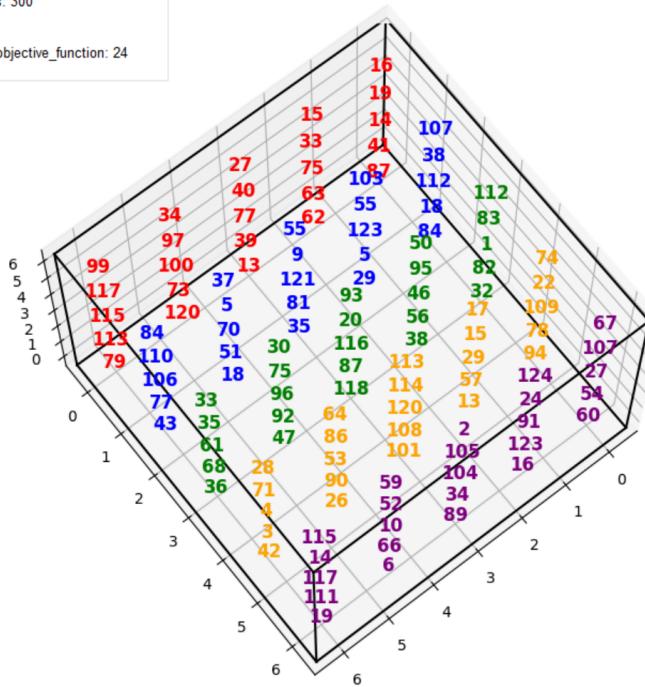
Algorithm: Genetic Algorithm  
 Time Taken: 14861.08 milliseconds  
 Perfect Magic Cube: No  
 Total States: 200  
 Iteration: 0  
 Maximum\_objective\_function: 17



4

Algorithm: Genetic Algorithm  
 Time Taken: 43389.89 milliseconds  
 Perfect Magic Cube: No  
 Total States: 300  
 Iteration: 0

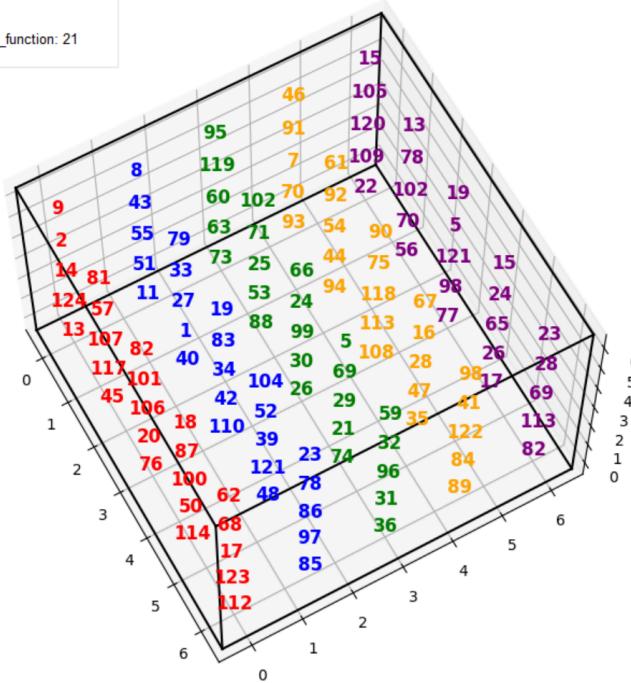
Maximum\_objective\_function: 24



5

Algorithm: Genetic Algorithm  
 Time Taken: 27089.60 milliseconds  
 Perfect Magic Cube: No  
 Total States: 200  
 Iteration: 0

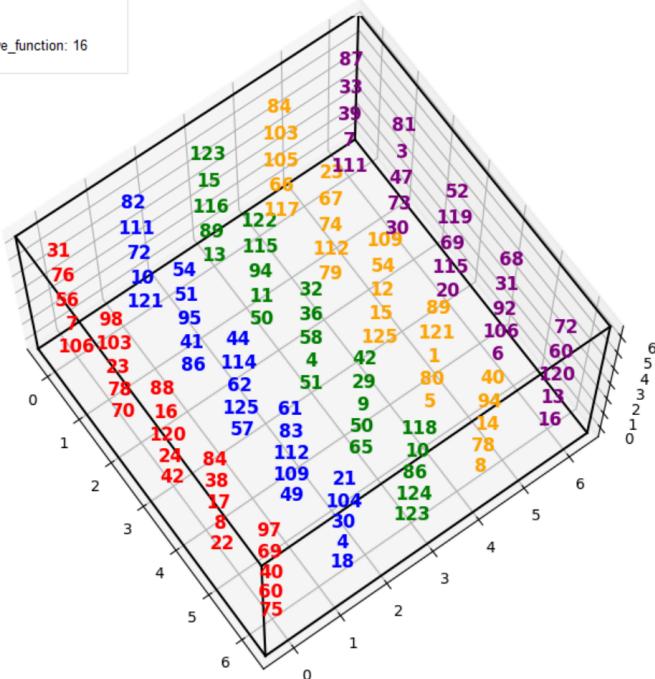
Maximum\_objective\_function: 21



6

Algorithm: Genetic Algorithm  
Time Taken: 15040.31 milliseconds  
Perfect Magic Cube: No  
Total States: 100  
Iteration: 0

Maximum\_objective\_function: 16



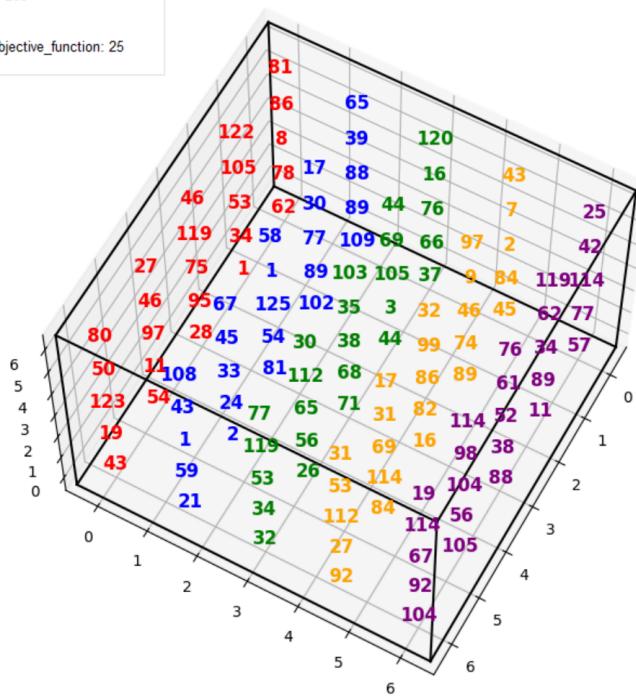
Setelah dilakukan *running* program maka didapatkan kondisi akhir sebagai berikut

| Percobaan | Representasi State Akhir 3D |
|-----------|-----------------------------|
|-----------|-----------------------------|

1

Algorithm: Genetic Algorithm  
Time Taken: 59310.62 milliseconds  
Perfect Magic Cube: No  
Total States: 200  
Iteration: 0

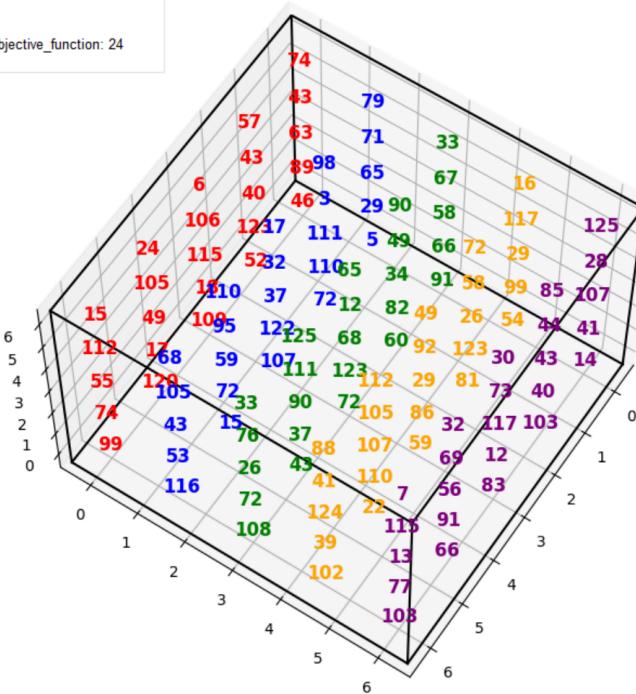
Maximum\_objective\_function: 25



2

Algorithm: Genetic Algorithm  
Time Taken: 37381.92 milliseconds  
Perfect Magic Cube: No  
Total States: 200  
Iteration: 0

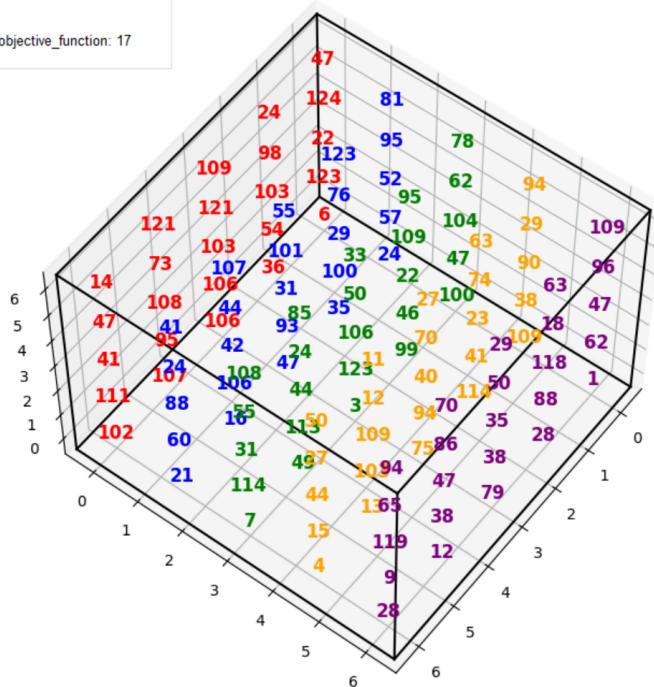
Maximum\_objective\_function: 24



3

Algorithm: Genetic Algorithm  
 Time Taken: 14861.08 milliseconds  
 Perfect Magic Cube: No  
 Total States: 200  
 Iteration: 0

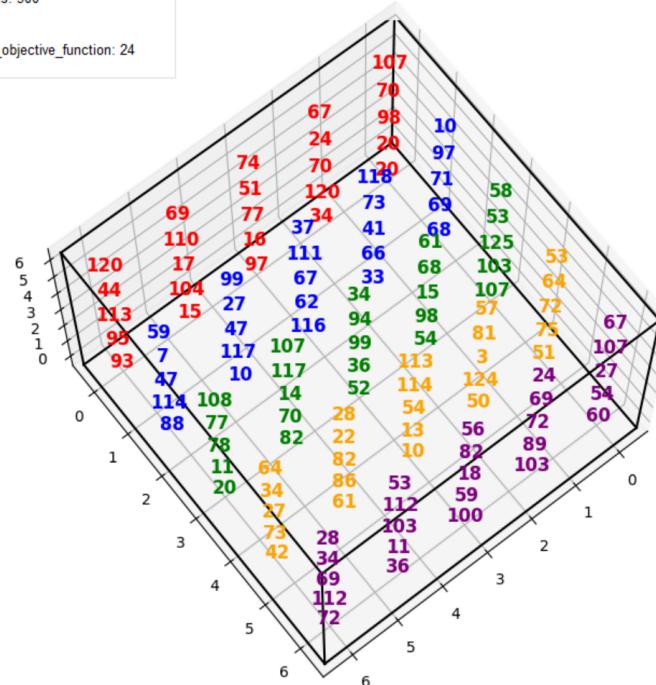
Maximum\_objective\_function: 17



4

Algorithm: Genetic Algorithm  
 Time Taken: 43389.89 milliseconds  
 Perfect Magic Cube: No  
 Total States: 300  
 Iteration: 0

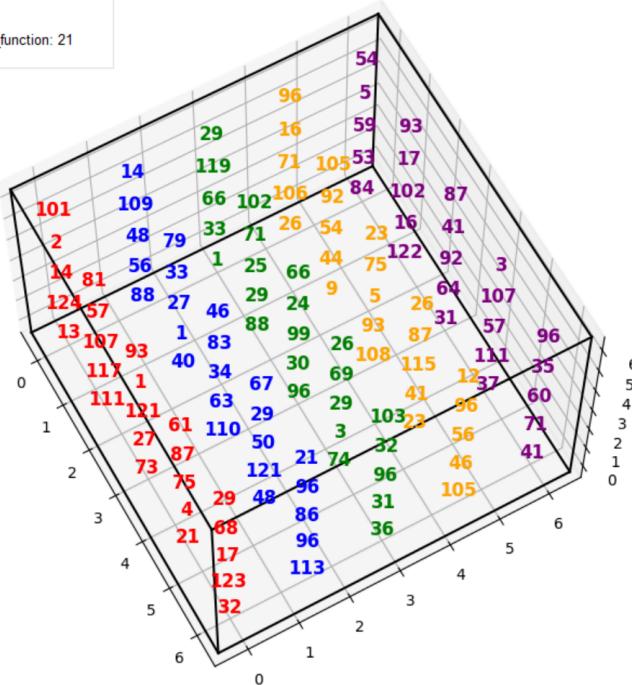
Maximum\_objective\_function: 24



5

Algorithm: Genetic Algorithm  
Time Taken: 27089.60 milliseconds  
Perfect Magic Cube: No  
Total States: 200  
Iteration: 0

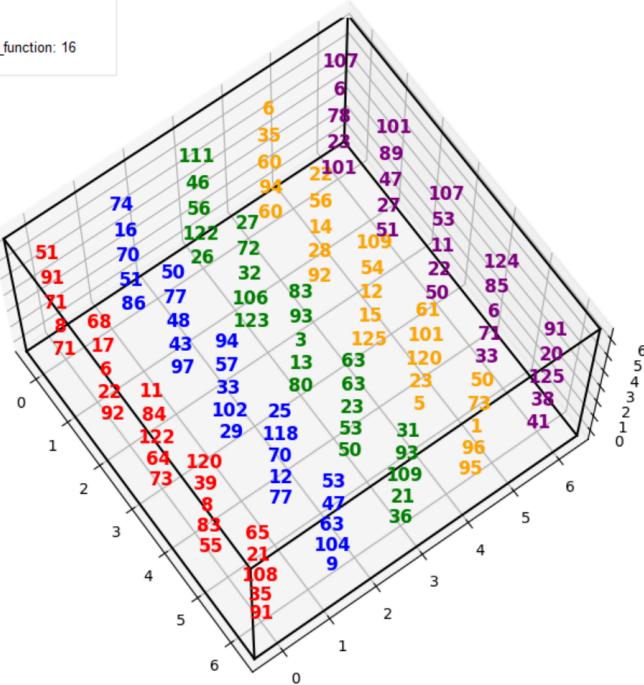
Maximum\_objective\_function: 21



6

Algorithm: Genetic Algorithm  
Time Taken: 15040.31 milliseconds  
Perfect Magic Cube: No  
Total States: 100  
Iteration: 0

Maximum\_objective\_function: 16



Berikut adalah state awal dan akhir dari masing-masing percobaan

| <b>Perco<br/>baan</b> | <b>Initial State</b>   | <b>Final State</b>  |
|-----------------------|--|---|
| <b>1</b>              | 53 73 49 50 90 58 20 98 14 39 99<br>47 71 72 35 7 109 62<br>76 42 105 45 123 54 21 11 43 124<br>34 107 37 106 6 108 100 88<br>87 40 91 75 4 13 70 36 23 59<br>66 100 97 52 53 87 78 81<br>41 35 89 74 32 31 68 95 5 67<br>107 15 27 83 47 51 43 44<br>24 88 12 20 30 99 91 112 101 123<br>121 73 122 34 92 46 80 62  | 62 78 8 86 81 1 34 53 105 122 28<br>95 75 119 46 54 11 97<br>46 27 43 19 123 50 80 109 89 88<br>39 65 102 89 77 30 17 81<br>54 125 1 58 2 24 33 45 67 21<br>59 1 43 108 37 66 76 16<br>120 44 3 105 69 44 71 68 38 35<br>103 26 56 65 112 30 32 34<br>53 119 77 45 84 2 7 43 89 74<br>46 9 97 16 82 86 99 32  |
| <b>2</b>              | 85 57 63 113 100 65 120 54 2 31<br>109 18 7 34 118 60 56 49<br>23 24 99 74 55 71 84 5 40 44<br>14 79 72 110 111 50 98 107<br>122 37 32 17 13 25 59 95 10 108<br>53 41 11 1 91 66 58 67<br>33 47 20 86 97 124 52 101 48 21<br>9 94 89 112 29 70 80 105<br>96 61 43 30 76 92 119 19 62 73<br>77 104 6 69 75 83 117 123   | 46 89 63 43 74 52 123 40 43 57<br>109 18 115 106 6 120 17 49<br>105 24 99 74 55 112 15 5 29 65<br>71 79 72 110 111 3 98 107<br>122 37 32 17 15 72 59 95 110 116<br>53 43 105 68 91 66 58 67<br>33 60 82 34 49 90 72 123 68 12<br>65 43 37 90 111 125 108 72<br>26 76 33 54 99 29 117 16 81 123<br>26 58 72 59 86 29 92 49<br>22 110 107 105 112 102 39 124 41<br>88 14 41 107 28 125 103 40 43<br>44 85 83 12 117 73 30 66 91 56<br>69 32 103 77 13 115 7 |
| <b>3</b>              | 60 24 93 104 58 23 2 21 61 39 95<br>89 57 52 32 26 29 74<br>113 35 56 25 70 87 41 10 118 84<br>82 101 33 14 36 27 20 9<br>15 6 83 3 17 96 44 75 119 49<br>53 107 34 28 45 123 38 11<br>71 31 88 48 85 18 102 69 111 1<br>54 4 90 65 115 47 63 92<br>22 43 8 30 94 59 64 105 50 16<br>122 103 7 19 125 108 68 76<br>80 40 86 112 106 73 66 81 124 120<br>62 109 114 121 98 55 116 78<br>91 13 110 97 37 67 99 100 12 42<br>117 72 79 46 51 5 77 | 6 123 22 124 47 36 54 103 98 24<br>106 106 103 121 109 107 95 108<br>73 121 102 111 41 47 14 24 57 52<br>95 81 35 100 29 76 123 47<br>93 31 101 55 16 106 42 44 107 21<br>60 88 24 41 100 47 104 62<br>78 99 46 22 109 95 3 123 106 50<br>33 49 113 44 24 85 7 114<br>31 55 108 109 38 90 29 94 114 41<br>23 74 63 75 94 40 70 27<br>13 103 109 12 11 4 15 44 27 50<br>1 62 47 96 109 28 88 118<br>18 63 79 38 35 50 29 12 38 47<br>86 70 28 9 119 65 94  |

|          |   |  |
|----------|---|--|
| <b>4</b> | 47 78 74 45 2 105 77 89 64 80 76<br>83 81 39 44 67 92 94<br>121 10 14 96 109 56 22 124 27 102<br>107 125 87 72 66 84 117 82<br>12 99 122 62 7 19 111 37 21 63<br>71 28 97 32 8 3 26 110<br>54 31 20 17 13 11 100 50 57 41<br>86 114 120 36 90 68 88 34<br>106 61 85 104 73 46 98 113 9 59<br>58 33 25 103 1 119 4 65  | 30 32 70 69 96 83 75 22 62 73 25<br>107 117 118 82 59 65 2<br>20 60 23 36 53 46 4 72 43 67<br>48 115 13 6 84 99 91 42<br>124 64 34 51 21 18 7 94 66 87<br>110 5 40 79 108 57 85 119<br>86 98 68 16 55 78 116 77 56 27<br>39 26 104 95 12 41 14 9<br>63 102 71 58 37 92 52 76 106 112<br>44 113 101 103 114 17 3 50   |
| <b>5</b> | 13 124 14 2 9 45 117 107 57 81<br>76 20 106 101 82 114 50 100<br>87 18 112 123 17 68 62 11 51 55<br>43 8 40 1 27 33 79 110<br>42 34 83 19 48 121 39 52 104 85<br>97 86 78 23 73 63 60 119<br>95 88 53 25 71 102 26 30 99 24<br>66 74 21 29 69 5 36 31<br>96 32 59 93 70 7 91 46 94 44<br>54 92 61 108 113 118 75 90<br>35 47 28 16 67 89 84 122 41 98<br>22 109 120 105 15 56 70 102<br>78 13 77 98 121 5 19 17 26 65<br>24 15 82 113 69 28 23  | 13 124 14 2 101 111 117 107 57 81<br>73 27 121 1 93 21 4 75<br>87 61 32 123 17 68 29 88 56 48<br>109 14 40 1 27 33 79 110<br>63 34 83 46 48 121 50 29 67 113<br>96 86 96 21 1 33 66 119<br>29 88 29 25 71 102 96 30 99 24<br>66 74 3 29 69 26 36 31<br>96 32 103 26 106 71 16 96 9 44<br>54 92 105 108 93 5 75 23<br>23 41 115 87 26 105 46 56 96 12<br>84 53 59 5 54 122 16 102<br>17 93 31 64 92 41 87 37 111 57<br>107 3 41 71 60 35 96 |
| <b>6</b> | 106 7 56 76 31 70 78 23 103 98<br>42 24 120 16 88 22 8 17<br>38 84 75 60 40 69 97 121 10 72<br>111 82 86 41 95 51 54 57<br>125 62 114 44 49 109 112 83 61 18<br>4 30 104 21 13 89 116 15<br>123 50 11 94 115 122 51 4 58 36<br>32 65 50 9 29 42 123 124<br>86 10 118 117 66 105 103 84 79 112<br>74 67 23 125 15 12 54 109<br>5 80 1 121 89 8 78 14 94 40<br>111 7 39 33 87 30 73 47<br>3 81 20 115 69 119 52 6 106 92<br>31 68 16 13 120 60 72 | 71 8 71 91 51 92 22 6 17 68 73<br>64 122 84 11 55 83 8<br>39 120 91 35 108 21 65 86 51 70<br>16 74 97 43 48 77 50 29<br>102 33 57 94 77 12 70 118 25 9<br>104 63 47 53 26 122 56 46<br>111 123 106 32 72 27 80 13 3 93<br>83 50 53 23 63 63 36 21<br>109 93 31 60 94 60 35 6 92 28<br>14 56 22 125 15 12 54 109<br>5 23 120 101 61 95 96 1 73 50<br>101 23 78 6 107 51 27 47<br>89 101 50 22 11 53 107 33 71 6<br>85 124 41 38 125 20 91   |

| Parameter  | Percobaan 1 | Percobaan 2 | Percobaan 3 | Percobaan 4 | Percobaan 5 | Percobaan 6 |
|------------|-------------|-------------|-------------|-------------|-------------|-------------|
| Durasi (s) | 59.310      | 37.381      | 14.861      | 43.389      | 27.089      | 15.040      |

|                     |     |     |     |     |     |     |
|---------------------|-----|-----|-----|-----|-----|-----|
| Jumlah Iterasi      | 200 | 200 | 200 | 300 | 200 | 100 |
| Banyak Populasi     | 300 | 200 | 100 | 150 | 150 | 150 |
| Initial State Value | 0   | 0   | 0   | 0   | 0   | 0   |
| Final State Value   | 25  | 24  | 17  | 24  | 21  | 16  |

Terdapat beberapa hal yang bisa disimpulkan dari tabel diatas. Pertama, algoritma ini gagal untuk menemukan solusi perfect magic square. Dengan menggunakan banyak populasi awal algoritma sebagai variabel kontrol, dapat dilihat bahwa semakin besar populasi maka semakin akurat pula hasil pencarian. Namun, hal ini juga berakibat pada bertambahnya waktu pencarian.

Hal yang sama berlaku pula dengan banyaknya iterasi sebagai variabel kontrol. Semakin banyak iterasi yang dilakukan algoritma, semakin besar pula nilai state akhir yang diberikan. Sama juga dengan waktu pencarian yang semakin lama untuk iterasi yang semakin tinggi. Akibatnya, bisa disimpulkan bahwa banyaknya populasi serta iterasi berbanding lurus dengan waktu pencarian serta nilai fungsi objektif state terbaik.

### 3.2. Analisis

Dari hasil eksperimen implementasi dari algoritma-algoritma local search, dapat dilihat bahwa setiap algoritma menghasilkan sebuah final state yang berhenti sebelum mencapai global optima. Hal ini dikarenakan algoritma-algoritma local search cenderung terjebak pada local optima. Berikut adalah state value terbaik yang didapatkan oleh masing-masing algoritma yang kami implementasikan.

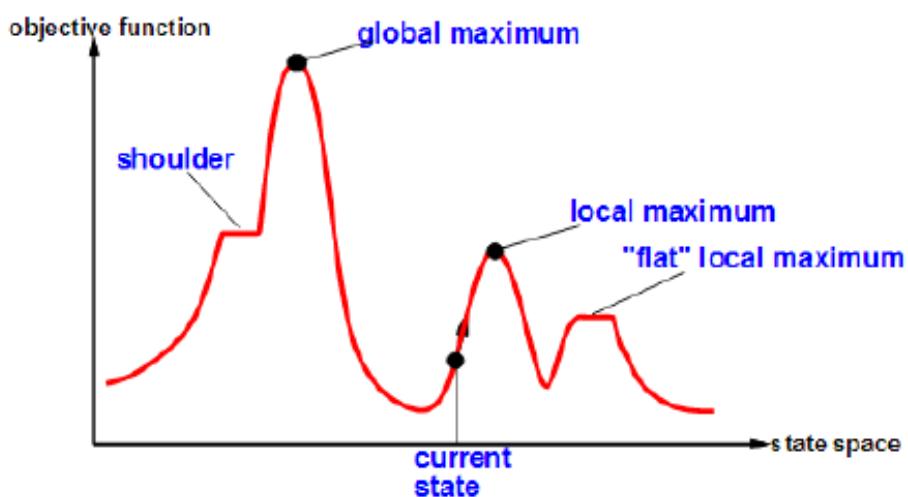
| Algoritma                        | State Value Terbaik | Durasi State Value Terbaik (s) |
|----------------------------------|---------------------|--------------------------------|
| Hill-Climbing Steepest Ascent    | 41                  | 248                            |
| Hill-Climbing with Sideways Move | 49                  | 2157                           |

|                              |    |      |
|------------------------------|----|------|
| Random Restart Hill-Climbing | 44 | 4855 |
| Stochastic Hill Climbing     | 29 | 352  |
| Simulated Annealing          | 58 | 121  |
| Genetic Algorithm            | 25 | 59   |

Karena fungsi objektif memiliki nilai state value maksimum atau nilai dimana state merupakan global optimal bernilai 109, maka algoritma yang paling mendekati global optimum adalah algoritma yang memiliki state value terdekat dengan 109. Sehingga urutan algoritma yang mendekati global optimum dari yang terdekat adalah:

1. Simulated Annealing - 58
2. Sideways Move Hill Climbing - 49
3. Random Restart Hill Climbing - 44
4. Steepest Ascent Hill Climbing - 41
5. Stochastic Hill Climbing - 29
6. Genetic Algorithm - 25

Hal tersebut dapat dijelaskan karena masing-masing algoritma memiliki cara yang berbeda untuk mencapai global optima.



source: state space landscape (Lazebnik, 2018)

Untuk Steepest Ascent Hill Climbing, algoritma berhenti pada local maximum karena dapat terjebak pada “peak” dan berhenti karena tidak lagi menemukan neighbor yang memiliki nilai state value yang lebih baik.

Untuk Hill Climbing with Sideways Move, algoritma memiliki dasar yang sama seperti variasi Steepest Ascent, namun berbeda dimana Hill Climbing with Sideways Move dapat mengeksplorasi “flat”, yaitu state lain yang memiliki nilai stated value yang sama. Hal ini dilakukan dengan harapan bahwa “flat” tersebut adalah “shoulder” sehingga dapat mencapai global optima.

Untuk Random Restart Hill Climbing memiliki hasil yang lebih baik dibandingkan Hill Climbing Steepest Ascent, hal ini disebabkan dengan adanya tahapan *restart* memungkinkan untuk memulai dari kondisi baru dan memperluas area eksplorasi algoritma dengan harapan salah satu solusi, merupakan global maxima sehingga algoritma dapat keluar dari *local maximum*.

Untuk Stochastic Hill Climbing memiliki nilai yang paling buruk dibandingkan algoritma Hill Climbing lain, hal ini disebabkan algoritma ini sangat bergantung pada keacakan neighbor hal ini tentu akan sulit untuk ditebak. Algoritma ini menggunakan random successor sebagai neighbor dengan harapan salah satu yang terpilih adalah yang terbaik untuk mendapatkan global optima.

Untuk Simulated Annealing, algoritma memiliki nilai final state yang paling baik diantara algoritma-algoritma lainnya. Hal ini disebabkan sifatnya yang secara perlahan mendekati solusi global optima dengan menggabungkan sifat random walk dan stochastic HC.

Untuk Genetic Algorithm, algoritma memiliki nilai final state yang paling buruk diantara algoritma-algoritma lainnya. Hal ini disebabkan sifatnya menggabungkan sifat random walk dan stochastic HC, sehingga menyebabkan algoritma ini bergantung pada keacakan. Kemudian banyaknya parameter yang mungkin diatur, kombinasi parameter yang kurang tepat bisa membuat algoritma ini menjadi tidak optimal.

Konsistensi hasil yang didapatkan dari tiap-tiap algoritma tentu berbeda-beda. Hal ini dikarenakan terdapat local search yang menggunakan sistem acak. Algoritma-algoritma yang menggunakan atau meliput penggunaan randomness dalam penentuan neighbor adalah Random Restart Hill Climbing, Stochastic Hill Climbing, Simulated Annealing dan Genetic algorithm. Randomness ini sangat mengurangi konsistensi dari hasil algoritma, namun konsistensi juga dapat ditinggikan dengan banyaknya iterasi yang dilakukan, contoh pada Simulated Annealing yang jumlah iterasinya sangat besar.

Selain itu terdapat algoritma Steepest Ascent Hill Climbing dan Hill Climbing with Sideways Move yang tidak menggunakan konsep acak. Hal ini membuat algoritma konsisten dalam mendapatkan hasil state value yang sama untuk pengulangan pencarian.

Jika dikaji lebih dalam melihat hasil dari eksperimen, pada algoritma Hill Climbing Steepest Ascent memiliki hasil [36-41] yang bervariasi tetapi tidak terlalu besar. Pada algoritma Hill Climbing with Sideways Move memiliki variasi yang lebih besar dibandingkan Hill Climbing Steepest Ascent. Pada algoritma Random Restart Hill Climbing memiliki hasil yang cenderung konsisten meski max restart diubah, tetapi tidak konsisten jika

terjadi perubahan nilai max iteration. Pada algoritma Stochastic Hill Climbing memiliki konsistensi yang sangat baik [26-29] tetapi dikarenakan algoritma ini sangat bergantung pada keacakan untuk *generate neighbor* maka hal ini bisa saja tidak terjadi terus menerus. Pada algoritma Simulated Annealing memiliki konsistensi yang sama baiknya dengan Stochastic [55-58]. Kemudian terakhir pada Genetic algorithm memiliki konsistensi yang rendah pula tetapi lebih jika dibandingkan dengan Random Restart HC.

Berdasarkan hasil eksperimen, urutan algoritma dari algoritma yang paling konsisten menuju tidak konsisten adalah:

Simulated Annealing = Stochastic HC > HC with Sideways Move  $\geq$  Steepest Ascent HC > Genetic Algorithm > Random Restart HC

Namun, jika dilihat secara teoritis, menurut kami seharusnya urutan algoritma dari algoritma yang paling konsisten menuju tidak konsisten adalah:

Steepest Ascent HC = HC with Sideways Move > Simulated Annealing  $\geq$  Random Restart HC  $\geq$  Stochastic > Genetic Algorithm

Perbedaan hasil eksperimen dan teoritis, disebabkan eksperimen yang kami gunakan masih secara kasar, tanpa menggunakan variabel kontrol. Hal ini tentu berpengaruh pada hasil variasi dari masing-masing algoritma. Hal ini juga terjadi karena jumlah sampel/eksperimen yang hanya sedikit sehingga dapat terjadi anomali pada data hasil eksperimen.

Pada algoritma Genetic Algorithm, banyak iterasi dan jumlah populasi akan mempengaruhi hasil akhir pencarian. Hal ini ditunjukan dari hasil eksperimen dimana semakin kecil populasi maka semakin kecil state value yang dicapai dan sebaliknya semakin besar populasi maka semakin besar state valuenya. Hal ini juga sama dengan jumlah iterasi, dimana semakin sedikit iterasi, maka semakin kecil state value yang dicapai dan sebaliknya semakin banyak iterasi maka semakin besar state valuenya. Sehingga kami dapat menyimpulkan bahwa banyak iterasi dan jumlah populasi sebanding dengan hasil state value yang didapatkan.

## **BAB 4**

### **KESIMPULAN DAN SARAN**

#### **4.1. Kesimpulan**

Dalam tugas ini setelah dilakukan percobaan terhadap beberapa algoritma untuk menyelesaikan masalah *magic cube*. Kami menggunakan beberapa algoritma local search seperti variasi hill climbing, simulated annealing, dan genetic algorithm. Hasil implementasi menunjukkan algoritma menghasilkan solusi yang paling dekat dengan global optima adalah algoritma *simulated annealing*.

Dari hasil eksperimen kami juga dapat menentukan algoritma yang sifatnya konsisten dalam menemukan solusi serta tidak konsisten karena faktor acak(randomness) yang digunakan saat pemilihan neighbor atau saat perpindahan state.

Dengan demikian, kami berhasil menyelesaikan persoalan pencarian solusi diagonal magic cube dengan local search, yaitu algoritma Steepest Ascent Hill-Climbing, Hill-Climbing with Sideways Move, Hill-Climbing Random Restart, Stochastic Hill-Climbing, Genetic Algorithm, dan Simulated Annealing yang berhasil diimplementasikan dalam bahasa Python.

#### **4.2. Saran**

Saran untuk kelompok ini diantaranya :

1. Memulai penggerjaan dari lebih awal
2. Meningkatkan komunikasi antar anggota
3. Melakukan pembagian tugas yang lebih jelas dan rata
4. Mendalami pengetahuan mengenai bahasa pemrograman lain selain python, seperti C++, agar implementasi dapat berjalan lebih cepat.
5. Mendalami bahasa pemrograman Python sehingga dapat menggunakan konsep concurrency.

## PEMBAGIAN TUGAS

| NIM      | Nama                       | Pembagian Tugas   |
|----------|----------------------------|---|
| 12821046 | Fardhan Indrayesa          | <p>Implementasi</p> <ul style="list-style-type: none"> <li>- Objective Function</li> <li>- Hill Climb Steepest Ascent</li> <li>- Hill Climb with Sideways Move</li> </ul> <p>Laporan</p> <ul style="list-style-type: none"> <li>- Pembahasan (HC with Sideways Move)</li> <li>- Hasil dan Analisis (HC Steepest Ascent, HC with Sideways Move)</li> </ul>   |
| 13522037 | Farhan Nafis Rayhan        | <p>Implementasi</p> <ul style="list-style-type: none"> <li>- Simulated Annealing</li> <li>- Genetic Algorithm</li> </ul> <p>Laporan</p> <ul style="list-style-type: none"> <li>- Hasil dan Analisis (Simulated Annealing, Genetic Algorithm)</li> <li>- Kesimpulan</li> </ul>   |
| 13522091 | Raden Francisco Trianto B. | <p>Implementasi</p> <ul style="list-style-type: none"> <li>- Magic Cube</li> <li>- Objective Function</li> <li>- Stochastic Hill Climbing</li> <li>- GUI dan Visualisasi</li> <li>- Bonus Video Player</li> <li>- Readme</li> </ul> <p>Laporan</p> <ul style="list-style-type: none"> <li>- Deskripsi Persoalan</li> <li>- Pembahasan (Magic Cube, Objective Function, HC Steepest Ascent, Stochastic HC, Simulated Annealing, Genetic</li> </ul> |

|          |                 |   |
|----------|-----------------|---|
|          |                 | <p>Algorithm)</p> <ul style="list-style-type: none"> <li>- Hasil dan Analisis (Stochastic HC &amp; Analisis Keseluruhan)</li> <li>- Kesimpulan dan Saran</li> </ul>   |
| 18321011 | Wikan Priambudi | <p>Implementasi</p> <ul style="list-style-type: none"> <li>- Objective Function</li> <li>- Hill Climb Random Restart</li> </ul> <p>Laporan</p> <ul style="list-style-type: none"> <li>- Deskripsi Persoalan</li> <li>- Pembahasan (HC Random Restart)</li> <li>- Hasil dan Analisis (HC Random Restart &amp; Analisis Keseluruhan)</li> <li>- Kesimpulan &amp; Saran</li> </ul> |

## REFERENSI

Hendricks, J. (2009). "Magic Squares". RecMath. Diakses pada <http://recmath.org/Magic%20Squares/hendricks.htm>

Weisstein, Eric W. "Magic Cube." From MathWorld--A Wolfram Web Resource. <https://mathworld.wolfram.com/MagicCube.html>.

Magischvierkant. *Three Dimensional Magic Squares*. <https://www.magischvierkant.com/three-dimensional-eng/magic-features/>.

GeeksforGeeks. Introduction to Hill Climbing | Artificial Intelligence. <https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/>

Norvig, P., & Russell, S. J. (2010). *Instructor's Manual: Exercise Solutions for Artificial Intelligence: A Modern Approach* (3rd ed., International Version). Pearson Education.

A Local Search Approach To The Student Sectioning Problem Using Hill Climbing - Scientific Figure on ResearchGate. Available from: [https://www.researchgate.net/figure/state-space-landscape-Lazebnik-2018\\_fig3\\_337347748](https://www.researchgate.net/figure/state-space-landscape-Lazebnik-2018_fig3_337347748)

GeeksforGeeks. *Genetic Algorithms*. <https://www.geeksforgeeks.org/genetic-algorithms/>

OEIS. *Magic constant of smallest order-n perfect magic cube*. <https://oeis.org/A109130>