

# **Laporan Tugas Besar 1 IF3170**

## **Inteligensi Artifisial**

Pencarian Solusi Diagonal Magic Cube dengan Local Search



### **Kelompok 10 :**

1. Rici Trisna Putra 13522026
2. Francesco Michael Kusuma 13522038
3. Keanu Amadius Gonza Wrahatno 13522082
4. Dimas Bagoes Hendrianto 13522112

**PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
2024**

## **DAFTAR ISI**

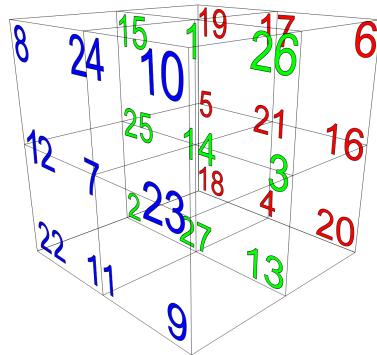
DAFTAR ISI	1
BAB 1	2
BAB 2	4
2.1 Pemilihan Objective Function	4
2.1.1 Magic Number	4
2.1.2 Objective Function	4
2.2 Penjelasan Implementasi	5
2.2.1 Kelas Cube	5
2.2.2 Kelas GeneticAlgorithm	9
2.2.3 Kelas HillClimbingSearch	12
2.2.4 Kelas HillClimbingSearchSteepestAscent	14
2.2.5 Kelas HillClimbingSearchSidewaysMove	17
2.2.6 Kelas RandomRestartHillClimbing	20
2.2.7 Kelas SimulatedAnnealing	21
2.2.7 Kelas StochasticHillClimbing	24
2.3 Hasil eksperimen dan analisis	25
BAB 3	26
3.1 Kesimpulan	26
3.2 Saran	26
BAB 4	28
REFERENSI	29

## BAB 1

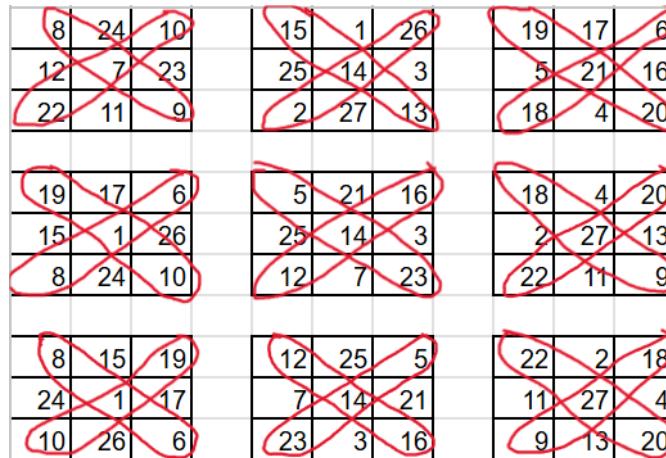
## **DESKRIPSI PERSOALAN**

Diagonal cube dengan order 5 baru ditemukan pada 13 November 2003 oleh Walter Trump dan Christian Boyer. Diagonal magic cube tidak mungkin dibuat pada orde  $1 < m < 5$ . Diagonal magic cube merupakan kubus yang tersusun dari angka 1 hingga  $n^3$  tanpa pengulangan dengan  $n$  adalah panjang sisi pada kubus tersebut. Angka-angka pada tersusun sedemikian rupa dengan aturan 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, kolom, tiang, diagonal ruang, dan diagonal pada suatu potongan bidang sama dengan magic number

Berikut ilustrasi dari potongan bidang yang ada pada suatu kubus berukuran 3:



Terdapat 9 potongan bidang, yaitu:



Diagonal yang dimaksud adalah yang dilingkari warna merah saja

Tugas kali ini kami akan menyelesaikan permasalahan Diagonal Magic Cube berukuran 5x5x5 menggunakan algoritma local search. Initial state dari suatu kubus adalah susunan angka 1 hingga  $5^3$  secara acak. Algoritma yang digunakan untuk menyelesaikan permasalahan ini adalah

- Steepest Ascent Hill-climbing
- Hill-climbing with Sideways Move
- Random Restart Hill-climbing
- Stochastic Hill-climbing
- Simulated Annealing
- Genetic Algorithm

Langkah yang boleh dilakukan adalah menukar posisi dari 2 angka pada kubus tersebut (2 angka yang ditukar tidak harus bersebelahan). Successor didapatkan dengan melakukan sekali penukaran.

## BAB 2

# PEMBAHASAN

### 2.1 Pemilihan Objective Function

#### 2.1.1 *Magic Number*

Untuk rentang angka dari *magic cube order 5* yaitu 1 hingga 125 memiliki median yaitu 63, yang kemudian menjadi angka penting dalam penyelesaian persoalan *magic cube order 5* ini. Untuk memenuhi syarat atau aturan dari *magic cube* yaitu 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*, dan jumlah angka-angka untuk seluruh diagonal pada suatu potongan bidang dari kubus sama dengan *magic number*, maka karena setiap elemen tersebut terdiri dari 5 buah angka maka angka yang dipilih menjadi *magic number* adalah 315 yang apabila dirumuskan adalah sebagai berikut :

$$S = \frac{n(n^3+1)}{2} = \frac{5(5^3+1)}{2} = 315$$

n = order magic cube

#### 2.1.2 *Objective Function*

Pada algoritma *local search*, hal yang terpenting adalah *state* dan *valuensya*. *Action* yang dilakukan pada algoritma ini adalah bergerak ke *neighbor* dari *current state*. Untuk bergerak ke *neighbor* diperlukan adanya *successor* yang nantinya akan dipilih satu. Pemilihan ini tergantung dari jenis algoritmanya dengan menentukan *value* pada *successornya*. *Value* ini didapat dari rumus berikut:

*Value* dihitung dengan menentukan jumlah baris, kolom, tiang, diagonal ruang, dan seluruh diagonal pada tiap potongan bidang yang jumlah angkanya sama dengan *magic number*.

- Jumlah baris pada diagonal *magic cube* 5x5 adalah 25
- Jumlah kolom pada diagonal *magic cube* 5x5 adalah 25
- Jumlah tiang pada diagonal *magic cube* 5x5 adalah 25
- Jumlah diagonal pada tiap potongan bidang pada diagonal *magic cube* 5x5 adalah 30
- Jumlah diagonal ruang pada *magic cube* 5x5 adalah 4

Maka *value* untuk mencapai global maksimum adalah 109

$$f_{total}(S) = f_{baris}(S) + f_{kolom}(S) + f_{tiang}(S) + f_{diagonal-ruang}(S) + f_{diagonal-sisi}(S)$$

Contoh: apabila suatu state hanya ada 1 baris dan 1 kolom yang jumlah angkanya sama dengan magic number, maka *value* dari *state* tersebut adalah 2.

Jadi rumus untuk mencari *valuanya* adalah jumlah elemen yang jumlah semua angkanya sama dengan *magic number*. Catatan: elemen adalah baris dan kolom dan pilar dan diagonal ruang dan semua diagonal pada tiap potongan bidang.

## 2.2 Penjelasan Implementasi

### 2.2.1 Kelas Cube

```
import numpy as np
import random

class Cube:
    def __init__(self, n):
        self.n = n
        self.magic_number = (self.n ** 3 + 1) / 2 * n
        self.data = self.generate_random_config(n)

    def generate_random_config(self, n):
        unique_numbers = random.sample(range(1, (n ** 3) + 1), n ** 3)
        data = np.array(unique_numbers).reshape((n, n, n))
        return data

    def swap(self, pos1, pos2):
        temp = self.data[pos1[0], pos1[1], pos1[2]]
        self.data[pos1[0], pos1[1], pos1[2]] = self.data[pos2[0], pos2[1], pos2[2]]
        self.data[pos2[0], pos2[1], pos2[2]] = temp

    def calculate_value(self):
        return (self.calculate_column_value() +
               self.calculate_row_value() +
               self.calculate_height_value() +
               self.calculate_side_diagonal_value() +
               self.calculate_space_diagonal_value())

    def calculate_column_value(self):
        total = 0
        for i in range(self.n):
            for y in range(self.n):
                if np.sum(self.data[i, y, :]) == self.magic_number:
                    total += 1
        return total

    def calculate_row_value(self):
        total = 0
        for i in range(self.n):
```

```
        for z in range(self.n):
            if np.sum(self.data[i, :, z]) == self.magic_number:
                total += 1
        return total

    def calculate_height_value(self):
        total = 0
        for y in range(self.n):
            for z in range(self.n):
                if np.sum(self.data[:, y, z]) == self.magic_number:
                    total += 1
        return total

    def calculate_side_diagonal_value(self):
        total = 0
        for z in range(self.n):
            if np.sum(self.data[z, np.arange(self.n), np.arange(self.n)]) == self.magic_number:
                total += 1
            if np.sum(self.data[z, np.arange(self.n), np.arange(self.n - 1, -1, -1)]) == self.magic_number:
                total += 1

        for y in range(self.n):
            if np.sum(self.data[np.arange(self.n), y, np.arange(self.n)]) == self.magic_number:
                total += 1
            if np.sum(self.data[np.arange(self.n), y, np.arange(self.n - 1, -1, -1)]) == self.magic_number:
                total += 1

        for i in range(self.n):
            if np.sum(self.data[np.arange(self.n), np.arange(self.n), i]) == self.magic_number:
                total += 1
            if np.sum(self.data[np.arange(self.n - 1, -1, -1), np.arange(self.n), i]) == self.magic_number:
                total += 1

        return total

    def calculate_space_diagonal_value(self):
        total = 0
        diagonals = [
            np.sum(self.data[np.arange(self.n), np.arange(self.n), np.arange(self.n)]),
            np.sum(self.data[np.arange(self.n), np.arange(self.n - 1, -1, -1), np.arange(self.n)]),
            np.sum(self.data[np.arange(self.n), np.arange(self.n), np.arange(self.n - 1, -1, -1)]),
            np.sum(self.data[np.arange(self.n), np.arange(self.n - 1, -1, -1), np.arange(self.n - 1, -1, -1)])
        ]
        total += sum(1 for x in diagonals if x == self.magic_number)
        return total

    def generate_random_point(self):
        return [random.randint(0, self.n - 1) for _ in range(3)]

    def print_value(self):
        counter = 0
        for i in range(self.n):
            for j in range(self.n):
```

```
for k in range(self.n):
    print(self.data[i, j, k], end=" ")
    counter += 1
    if counter % 25 == 0:
        print()
print()

def plot_number_cube(self, ax, cube_data, title):
    spacing = 1
    colors = ['red', 'blue', 'green', 'purple', 'orange']

    for i in range(self.n):
        for j in range(self.n):
            for k in range(self.n):
                x_center = i * spacing + spacing / 2
                y_center = j * spacing + spacing / 2
                z_center = k * spacing + spacing / 2

                color = colors[k % len(colors)]

                ax.text(x_center, y_center, z_center, str(cube_data[i, j, k]),
                        ha='center', va='center', color=color, fontsize=10)

    ax.set_xlim([0, self.n])
    ax.set_ylim([0, self.n])
    ax.set_zlim([0, self.n])

    ax.set_xlabel('X Axis')
    ax.set_ylabel('Y Axis')
    ax.set_zlabel('Z Axis')
    ax.set_title(title)
```

1. def \_\_init\_\_

Fungsi yang berguna untuk pembentukan kubus yang memiliki properti n sebagai ukuran atau dimensi kubus, properti magic number sebagai jumlah tiap baris, kolom, diagonal sisi, dan diagonal ruang sebagai pembanding dalam penentuan nilai heuristik atau *value* dari kubus itu sendiri, dan properti data sebagai tempat penyimpanan angka dalam kubus tersebut

2. def generate\_random\_config

Fungsi untuk menciptakan sebuah kubus dengan ukuran sisi n secara *random* mengikuti aturan kubus unik dimana setiap kubus kecilnya berisikan angka yang berlainan antara satu dengan yang lainnya. Penciptaan kubus diawali dengan pembuatan sebuah *array* sepanjang n pangkat 3 dengan angka yang berbeda, kemudian *array* tersebut akan *dreshape* menjadi *array* 3 dimensi dan kemudian *direturn*.

3. def swap

Fungsi yang berguna untuk menukar posisi dua kubus di dalam sebuah kubus besar berdasarkan dua buah parameter masukan yaitu pos1 dan pos2 yang masing-masing mewakili posisi kubus kecil satu dan kubus kecil dua

4. def calculate\_value

Fungsi untuk menghitung nilai heuristik dari sebuah kubus dengan memanggil fungsi hitung lainnya yaitu calculate\_column\_value(), calculate\_row\_value(), calculate\_height\_value(), calculate\_side\_diagonal\_value(), dan calculate\_space\_diagonal\_value() dan mengembalikan nilai heuristik

5. def calculate\_column\_value

Fungsi untuk menghitung berapa banyak kolom dalam kubus yang total nilainya sama dengan *magic number* dan banyak kolom tadi akan dilemparkan sebagai hasil dari fungsi

6. def calculate\_row\_value

Fungsi untuk menghitung berapa banyak baris dalam kubus yang total nilainya sama dengan *magic number* dan banyak baris tadi akan dilemparkan sebagai hasil dari fungsi

7. def calculate\_height\_value

Fungsi untuk menghitung berapa banyak tiang dalam kubus yang total nilainya sama dengan *magic number* dan banyak tiang tadi akan dilemparkan sebagai hasil dari fungsi

8. def calculate\_side\_diagonal\_value

Fungsi untuk menghitung berapa banyak diagonal sisi dalam kubus yang total nilainya sama dengan *magic number* dan banyak diagonal sisi tadi akan dilemparkan sebagai hasil dari fungsi

9. def calculate\_space\_diagonal\_value

Fungsi untuk menghitung berapa banyak diagonal ruang dalam kubus yang total nilainya sama dengan *magic number* dan banyak diagonal ruang tadi akan dilemparkan sebagai hasil dari fungsi

10. def generate\_random\_point

Fungsi yang membuat sepasang posisi dalam kubus untuk ditukar secara acak yang keduanya akan menjadi *return* dari fungsi ini

11. def print\_value

Fungsi untuk menampilkan kubus yang dibuat untuk memeriksa hasil

## 12. plot\_number\_cube

Fungsi untuk menvisualisasikan suatu kubus secara 3 dimensi dengan bantuan kakas matplotlib

### 2.2.2 Kelas GeneticAlgorithm

```
import numpy as np
import matplotlib.pyplot as plt
import random
from algorithm.Cube import Cube
import time

class GeneticAlgorithm:
    def __init__(self, population_size, nmax, n = 5, elite_count=1, crossover_rate=0.8, mutation_rate=0.1):
        self.population_size = population_size
        self.nmax = nmax
        self.n = n
        self.elite_count = elite_count
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate
        self.population = [Cube(n) for _ in range(population_size)]
        self.max_values = []
        self.mean_values = []
        self.iteration = 0
        self.best_per_iteration = []
        self.two_best_initial = []
        self.two_best_last = []
        self.best_value = 0

    def solve(self):
        sorted_population = sorted(self.population, key=lambda x: x.calculate_value(), reverse=True)
        self.two_best_initial.append(sorted_population[0].data.copy())
        self.two_best_initial.append(sorted_population[1].data.copy())

        start_time = time.time()
        for gen in range(self.nmax):
            self.iteration += 1
            parent_population = []
            IsEnd, population_strip = self.generate_random_selection_probability()

            if IsEnd:
                break

            sorted_population = sorted(self.population, key=lambda x: x.calculate_value(), reverse=True)
            self.best_per_iteration.append(sorted_population[0].data.copy())
            new_population = sorted_population[:self.elite_count]
```

```
        while len(new_population) < self.population_size:
            parent1, parent2 = self.select_parents(population_strip)

            if random.random() < self.crossover_rate:
                offspring1, offspring2 = self.crossover(parent1, parent2)
            else:
                offspring1, offspring2 = parent1.data.copy(), parent2.data.copy()

            if random.random() < self.mutation_rate:
                offspring1 = self.mutation(offspring1)
            if random.random() < self.mutation_rate:
                offspring2 = self.mutation(offspring2)

            new_population.append(Cube(self.n))
            new_population[-1].data = offspring1.reshape(5, 5, 5)

            if len(new_population) < self.population_size:
                new_population.append(Cube(self.n))
                new_population[-1].data = offspring2.reshape(5, 5, 5)

        self.population = new_population

        print("\nRESULT")
        end_time = time.time()
        duration = end_time - start_time
        print(f"Genetic Algorithm Duration: {duration:.4f} seconds")

        self.two_best_last.append(sorted_population[0].data.copy())
        self.two_best_last.append(sorted_population[1].data.copy())

        print("Best value: ", self.best_value)
        print("Number of population: ", self.population_size)
        print("Number of iterations: ", self.iteration)

        self.plot_value()

    def generate_random_selection_probability(self):
        population_value = [cube.calculate_value() for cube in self.population]

        self.max_values.append(max(population_value))
        self.mean_values.append(np.mean(population_value))

        total = sum(population_value)
        IsEnd = any(value == 109 for value in population_value)

        self.best_value = max(population_value)

        if total == 0:
            population_strip = [100 * (i + 1) / self.population_size for i in range(self.population_size)]
        else:
            population_strip = np.cumsum([val / total * 100 for val in population_value]).tolist()
```

```
    return IsEnd, population_strip

def select_parents(self, population_strip):
    parent1 = self.roulette_selection(population_strip)
    parent2 = self.roulette_selection(population_strip)
    return parent1, parent2

def roulette_selection(self, population_strip):
    rand = random.uniform(0, 100)
    for idx, strip in enumerate(population_strip):
        if rand <= strip:
            return self.population[idx]
    return self.population[-1]

def crossover(self, parent1, parent2):
    temp1 = np.array(parent1.data).reshape(125)
    temp2 = np.array(parent2.data).reshape(125)

    start, end = sorted(random.sample(range(125), 2))

    offspring1 = [None] * 125
    offspring2 = [None] * 125

    offspring1[start:end] = temp1[start:end]
    offspring2[start:end] = temp2[start:end]

def fill_offspring(offspring, parent_data, start, end):
    current_pos = end
    for val in parent_data:
        if val not in offspring:
            if current_pos >= 125:
                current_pos = 0
            while start <= current_pos < end:
                current_pos += 1
            offspring[current_pos] = val
            current_pos += 1
    return offspring

offspring1 = fill_offspring(offspring1, temp2, start, end)
offspring2 = fill_offspring(offspring2, temp1, start, end)

offspring1 = np.array(offspring1).reshape(5, 5, 5)
offspring2 = np.array(offspring2).reshape(5, 5, 5)

return offspring1, offspring2

def mutation(self, individual):
    mutated = individual.flatten()

    for idx in range(len(mutated)):
        if random.random() < 0.05:
            swap_idx = random.randint(0, len(mutated) - 1)
```

```
        mutated[idx], mutated[swap_idx] = mutated[swap_idx], mutated[idx]

    return np.array(mutated).reshape(5, 5)

def ordered_crossover(self, parent1, parent2):
    child = [None] * len(parent1)
    start, end = sorted(random.sample(range(len(parent1)), 2))
    child[start:end] = parent1[start:end]

    current_pos = end
    for elem in parent2:
        if elem not in child:
            if current_pos >= len(parent1):
                current_pos = 0
            child[current_pos] = elem
            current_pos += 1

    return np.array(child)

def plot_value(self):
    fig1 = plt.figure(figsize=(12, 6))

    grid = fig1.add_gridspec(2,2,height_ratios = [1,1])
    ax1 = fig1.add_subplot(grid[0,0], projection='3d')
    self.population[0].plot_number_cube(ax1, self.two_best_initial[0], "Initial Cube 1")
    ax2 = fig1.add_subplot(grid[0,1], projection='3d')
    self.population[0].plot_number_cube(ax2, self.two_best_initial[1], "Initial Cube 2")
    ax3 = fig1.add_subplot(grid[1,0], projection='3d')
    self.population[0].plot_number_cube(ax3, self.two_best_last[0], "Final Cube 1")
    ax4 = fig1.add_subplot(grid[1,1], projection='3d')
    self.population[0].plot_number_cube(ax4, self.two_best_last[1], "Final Cube 2")

    ax1.view_init(elev=30, azim=30)
    ax2.view_init(elev=30, azim=30)
    ax3.view_init(elev=30, azim=30)
    ax4.view_init(elev=30, azim=30)

    plt.tight_layout()

    plt.figure(figsize=(12, 6))

    plt.plot(self.mean_values, linestyle='-', color='r', label='Mean Value', linewidth = 0.5)
    plt.plot(self.max_values, linestyle='-', color='b', label='Max Value')

    plt.title("Max and Mean Cube Values Over Generations")
    plt.xlabel("Generation")
    plt.ylabel("Cube Value")
    plt.grid()

    plt.legend()

    plt.tight_layout()
```

```
plt.show()
```

1. def \_\_init\_\_

Kelas utama yang mengimplementasikan algoritma genetika dengan parameter:

- population\_size: Ukuran populasi
- nmax: Maksimum jumlah generasi
- n: Dimensi kubus
- elite\_count: Jumlah individu terbaik yang dipertahankan
- crossover\_rate: Tingkat crossover ( $0.8 = 80\%$  kemungkinan)
- mutation\_rate: Tingkat mutasi ( $0.1 = 10\%$  kemungkinan)

2. def solve(self):

Ini adalah fungsi utama yang menjalankan algoritma genetika:

Melakukan iterasi sebanyak nmax generasi

Pada setiap generasi:

- Menghitung probabilitas seleksi
- Menyimpan individu elite
- Melakukan crossover dan mutasi untuk menghasilkan populasi baru
- Mengganti populasi lama dengan populasi baru

3. def generate\_random\_selection\_probability(self):

Fungsi ini menghitung nilai fitness setiap individu dalam populasi, menghasilkan probabilitas seleksi berdasarkan nilai fitness, mengembalikan IsEnd (true jika solusi optimal ditemukan) dan strip probabilitas kumulatif

4. def select\_parents(self, population\_strip):

Fungsi untuk memilih dua parent menggunakan metode roulette wheel selection

5. def roulette\_selection(self, population\_strip):

Implementasi roulette wheel selection, menggunakan probabilitas kumulatif untuk memilih individu. Semakin tinggi fitness, semakin besar kemungkinan terpilih

6. def crossover(self, parent1, parent2):

Melakukan crossover antara dua parent dengan :

- Mengubah bentuk data menjadi array 1D
- Memilih titik crossover secara acak
- Menukar bagian array setelah titik crossover
- Menghasilkan dua offspring

7. def mutation(self, individual):

Melakukan mutasi pada individu dengan :

- Untuk setiap gen, ada 5% kemungkinan mutasi
- Mutasi dilakukan dengan menukar posisi dengan gen lain secara acak
- Mengembalikan individu yang telah dimutasi

8. def ordered\_crossover(self, parent1, parent2):

Implementasi ordered crossover :

- Memilih segmen dari parent1
- Mengisi sisa posisi dengan elemen dari parent2 yang belum ada
- Menjaga urutan elemen dari parent2

9. def plot\_value(self):

Fungsi untuk memvisualisasi plotting statistik nilai maksimal populasi dan nilai rata-rata populasi dalam bentuk grafik dan juga visualisasi kubus 3d pada konfigurasi awal dan konfigurasi akhir dengan bantuan kakas matplotlib

### 2.2.3 Kelas HillClimbingSearch

```
from algorithm.Cube import *
import matplotlib.pyplot as plt
import copy

class HillClimbingSearch:
    def __init__(self, n):
        self.n = n
        self(cube) = Cube(n)
        self.values = []
        self.current_value = 0
        self.iteration = 0
        self.initial_state = []
        self.final_state = []

    def solve(self):
        self.current_value = self(cube).calculate_value()

        initial_config = copy.deepcopy(self(cube).data)
        list_swap_points = []

        while True:
            self.values.append(self.current_value)
            self.iteration += 1
            better_found = False

            for i in range(self.n**3):
                for j in range(i + 1, self.n**3):
                    pos1 = self.linearpos_to_3dpos(i)
```

```
pos2 = self.linearpos_to_3dpos(j)

self(cube).swap(pos1, pos2)
new_value = self(cube).calculate_value()

if new_value > self.current_value:
    self.current_value = new_value
    better_found = True
    break

self(cube).swap(pos1, pos2)

if better_found:
    list_swap_points.append([self.from_3dpos_to_linearpos(pos1), self.from_3dpos_to_linearpos(pos2)])
    break

if not better_found:
    break

self.initial_state = initial_config
self.final_state = self(cube).data

return list_swap_points, initial_config

def linearpos_to_3dpos(self, num):
    i = num // (self.n**2)
    j = (num % (self.n**2)) // self.n
    k = (num % (self.n**2)) % self.n
    return [i, j, k]

def from_3dpos_to_linearpos(self, pos):
    return pos[0] * (self.n**2) + pos[1] * (self.n) + pos[2]

def print_value(self):
    self(cube).print_value()

def getIterations(self):
    return self.iteration

def getCurrentValue(self):
    return self.current_value

def getInitialState(self):
    return self.initial_state

def getFinalState(self):
    return self.final_state

def list_of_values(self):
    return self.values
```

1. def \_\_init\_\_(self, n):

Ini adalah kelas yang mengimplementasikan algoritma Hill Climbing untuk optimasi kubus dengan parameter:

- n: Dimensi kubus ( $n \times n \times n$ )
- cube: Instance dari kelas Cube yang merepresentasikan kubus

2. def solve(self):

Fungsi utama yang menjalankan algoritma hill climbing dengan :

- Mencatat nilai awal kubus
- Menyimpan konfigurasi awal
- Melakukan iterasi pencarian solusi yang lebih baik dengan:
- Mencoba menukar setiap pasangan posisi dalam kubus
- Jika menemukan nilai yang lebih baik, menyimpan perubahan
- Berhenti ketika tidak ada perbaikan yang ditemukan
- Mengembalikan daftar pertukaran posisi dan konfigurasi awal

3. def linearpos\_to\_3dpos(self, num):

Fungsi untuk mengkonversi posisi linear (1D) ke posisi 3D dengan :

- Input: posisi linear (0 sampai  $n^3-1$ )
- Output: array [i,j,k] yang merepresentasikan posisi dalam kubus 3D
- Menggunakan operasi pembagian dan modulo untuk mengkonversi

4. def from\_3dpos\_to\_linearpos(self, pos):

Fungsi untuk mengkonversi posisi 3D ke posisi linear dengan :

- Input: array [i,j,k] posisi dalam kubus 3D
- Output: posisi linear (0 sampai  $n^3-1$ )
- Menggunakan formula:  $i * n^2 + j * n + k$

5. def print\_value(self):

Fungsi untuk mencetak nilai kubus saat ini menggunakan metode print\_value() dari kelas Cube.

## 2.2.4 Kelas HillClimbingSearchSteepestAscent

```
from algorithm.Cube import *
import matplotlib.pyplot as plt
import copy
import time

class HillClimbingSearchSteepestAscent:
    def __init__(self, n=5):
        self.n = n
        self(cube = Cube(n)
        self.iterasi = 0
        self.value = 0
        self.values = []

    def solve(self):
        self.value = self(cube.calculate_value()
```

Tugas Besar 1 IF 3170 Intelelegensi Artifisial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search

```
initial_config = copy.deepcopy(self.cube.data)
list_swap_points = []

start_time = time.time()

while True:
    self.values.append(self.value)
    max_value = self.value
    max_pos1 = None
    max_pos2 = None
    found_better = False

    for i in range(self.n**3):
        for j in range(i + 1, self.n**3):
            pos1 = self.linearpos_to_3dpos(i)
            pos2 = self.linearpos_to_3dpos(j)

            self.cube.swap(pos1, pos2)
            new_value = self.cube.calculate_value()

            if new_value > max_value:
                max_value = new_value
                max_pos1 = pos1
                max_pos2 = pos2
                found_better = True

            self.cube.swap(pos1, pos2)

    self.iterasi += 1
    if found_better:
        self.cube.swap(max_pos1, max_pos2)
        self.value = max_value

        list_swap_points.append([self.from_3dpos_to_linearpos(max_pos1), self.from_3dpos_to_linearpos(max_pos2)])
    else:
        break

end_time = time.time()
duration = end_time - start_time
print(f"\n\nSteepest Ascent Algorithm Duration: {duration:.4f} seconds")

print("\nBest value: ", self.values[-1])
print("Number of iterations: ", self.iterasi)
self.plot_value(initial_config, self.cube.data)
return list_swap_points, initial_config

def linearpos_to_3dpos(self, num):
    i = num // (self.n**2)
    j = (num % (self.n**2)) // self.n
    k = (num % (self.n**2)) % self.n
    return [i, j, k]

def from_3dpos_to_linearpos(self, pos):
    return pos[0] * (self.n**2) + pos[1] * (self.n) + pos[2]

def print_value(self):
    self.cube.print_value()

def plot_value(self, initial_cube_data, final_cube_data):
    fig1 = plt.figure(figsize=(12, 6))
```

```
ax1 = fig1.add_subplot(121, projection='3d')
self.cube.plot_number_cube(ax1, initial_cube_data, "Initial Configuration")
ax2 = fig1.add_subplot(122, projection='3d')
self.cube.plot_number_cube(ax2, final_cube_data, "Final Configuration")
ax1.view_init(elev=30, azim=30)
ax2.view_init(elev=30, azim=30)
plt.tight_layout()

plt.figure(figsize=(12, 6))
plt.plot(self.values, marker='o', linestyle='-', color='b')
plt.title("Cube Value Through Hill Climbing Steepest Ascent")
plt.xlabel("Iteration")
plt.ylabel("Value")
plt.grid()

plt.show()
```

1. def \_\_init\_\_(self, n):

Ini adalah kelas yang mengimplementasikan algoritma Hill Climbing dengan strategi Steepest Ascent, dimana:

- n: Dimensi kubus ( $n \times n \times n$ )
- cube: Instance dari kelas Cube
- iterasi: Menghitung jumlah iterasi yang dilakukan
- value: Menyimpan nilai state saat ini

2. def solve(self):

Fungsi utama yang mengimplementasikan Steepest Ascent Hill Climbing dengan :

- Menyimpan nilai awal dan konfigurasi awal
- Pada setiap iterasi:
  - Mencari nilai terbaik dari semua kemungkinan pertukaran
  - Menyimpan posisi dengan peningkatan nilai terbesar
  - Melakukan pertukaran hanya untuk nilai terbaik yang ditemukan
  - Menghitung iterasi
  - Berhenti jika tidak ada perbaikan

3. def linearpos\_to\_3dpos(self, num):

Fungsi untuk mengkonversi posisi linear (1D) ke posisi 3D dengan :

- Input: posisi linear (0 sampai  $n^3-1$ )
- Output: array [i,j,k] yang merepresentasikan posisi dalam kubus 3D
- Menggunakan operasi pembagian dan modulo untuk mengkonversi

4. def from\_3dpos\_to\_linearpos(self, pos):

Fungsi untuk mengkonversi posisi 3D ke posisi linear dengan :

- Input: array [i,j,k] posisi dalam kubus 3D
- Output: posisi linear (0 sampai  $n^3-1$ )

- Menggunakan formula:  $i * n^2 + j * n + k$
5. def print\_value(self):  
Fungsi untuk mencetak nilai kubus saat ini menggunakan metode print\_value() dari kelas Cube.
6. def plot\_value(self, initial\_cube\_data, final\_cube\_data):  
Fungsi untuk melakukan visualisasi statistik berupa objective function pada tiap iterasi dalam bentuk grafik dan juga visualisasi kubus 3d konfigurasi awal dan konfigurasi akhir dengan bantuan kakas matplotlib.

## 2.2.5 Kelas HillClimbingSearchSidewaysMove

```
from algorithm.Cube import *
import matplotlib.pyplot as plt
import random
import copy
import time

class HillClimbingSidewaysMove:
    def __init__(self, max_sideways_moves, n=5):
        self.n = n
        self.cube = Cube(n)
        self.max_sideways_moves = max_sideways_moves
        self.values = []
        self.value = 0
        self.iterasi = 0

    def solve(self):
        self.value = self.cube.calculate_value()
        print(f"Initial Value: {self.value}")
        sideways_moves_count = 0

        initial_config = copy.deepcopy(self.cube.data)
        list_swap_points = []
        sideways_moves = []

        start_time = time.time()

        while True:
            self.values.append(self.value)
            max_value = self.value
            max_pos1 = None
            max_pos2 = None
            found_better = False

            for i in range(self.n**3):
                for j in range(i + 1, self.n**3):
                    pos1 = self.linearpos_to_3dpos(random.randint(0, i))
                    pos2 = self.linearpos_to_3dpos(random.randint(0, j))

                    self.cube.swap(pos1, pos2)

                    if self.cube.calculate_value() > max_value:
                        max_value = self.cube.calculate_value()
                        max_pos1 = pos1
                        max_pos2 = pos2
                        found_better = True

            if not found_better:
                break

            sideways_moves.append((max_pos1, max_pos2))
            sideways_moves_count += 1

            if sideways_moves_count == self.max_sideways_moves:
                break

        end_time = time.time()
        print(f"End Time: {end_time - start_time} seconds")
```

Tugas Besar 1 IF 3170 Intelelegensi Artifisial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search

```
new_value = self.cube.calculate_value()

if new_value > max_value:
    max_value = new_value
    max_pos1 = pos1
    max_pos2 = pos2
    found_better = True
elif new_value == max_value:
    sideways_moves.append((pos1, pos2))

self.cube.swap(pos1, pos2)

self.iterasi += 1
if found_better:
    self.cube.swap(max_pos1, max_pos2)
    self.value = max_value
    print(f"Moved to new position with value: {self.value}")

list_swap_points.append([self.from_3dpos_to_linearpos(max_pos1), self.from_3dpos_to_linearpos(max_pos2)])
elif sideways_moves:
    best_pos1, best_pos2 = sideways_moves[-1]
    self.cube.swap(best_pos1, best_pos2)
    self.value = self.cube.calculate_value()
    sideways_moves_count += 1
    list_swap_points.append([self.from_3dpos_to_linearpos(best_pos1), self.from_3dpos_to_linearpos(best_pos2)])
    print(f"Sideways Move: Swapped {best_pos1} and {best_pos2}. New Value: {self.value}")
    if sideways_moves_count >= self.max_sideways_moves:
        print("Maximum sideways moves reached.")
        break
else:
    break

end_time = time.time()
duration = end_time - start_time
print(f"\n\nHill Climbing Sideways Algorithm Duration: {duration:.4f} seconds")

print("\nBest value: ", self.values[-1])
print("Number of iterations: ", self.iterasi)
self.plot_value(initial_config, self.cube.data)
return list_swap_points, initial_config

def linearpos_to_3dpos(self, num):
    i = num // (self.n**2)
    j = (num % (self.n**2)) // self.n
    k = (num % (self.n**2)) % self.n
    return [i, j, k]

def from_3dpos_to_linearpos(self, pos):
    return pos[0] * (self.n**2) + pos[1] * (self.n) + pos[2]

def print_value(self):
    self.cube.print_value()

def plot_value(self, initial_cube_data, final_cube_data):
    fig1 = plt.figure(figsize=(12, 6))
    ax1 = fig1.add_subplot(121, projection='3d')
    self.cube.plot_number_cube(ax1, initial_cube_data, "Initial Configuration")
    ax2 = fig1.add_subplot(122, projection='3d')
    self.cube.plot_number_cube(ax2, final_cube_data, "Final Configuration")
    ax1.view_init(elev=30, azim=30)
    ax2.view_init(elev=30, azim=30)
```

```
plt.tight_layout()

plt.figure(figsize=(12, 6))
plt.plot(self.values, marker='o', linestyle='-', color='b')
plt.title("Cube Value Through Hill Climbing Sideways Move Ascent")
plt.xlabel("Iteration")
plt.ylabel("Value")
plt.grid()

plt.show()
```

1. def \_\_init\_\_(self, n, max\_sideways\_moves):

Ini adalah kelas yang mengimplementasikan Hill Climbing dengan Sideways Moves, dimana:

- n: Dimensi kubus ( $n \times n \times n$ )
- cube: Instance dari kelas Cube
- max\_sideways\_moves: Batas maksimum langkah sideways yang diperbolehkan

2. def solve(self):

Fungsi utama yang mengimplementasikan Hill Climbing dengan Sideways Moves:

a) Inisialisasi:

- Menyimpan nilai awal
- Menginisialisasi counter untuk sideways moves

b) Pencarian solusi:

- Mencari nilai terbaik dari semua kemungkinan pertukaran
- Menyimpan gerakan sideways yang memiliki nilai sama
- Melakukan pertukaran berdasarkan tiga kondisi:

1. Jika menemukan nilai yang lebih baik:

- Melakukan pertukaran
- Memperbarui nilai current
- Mencatat pertukaran

2. Jika menemukan nilai sama (sideways):

- Melakukan pertukaran sideways
- Menambah counter sideways
- Berhenti jika mencapai batas maksimum

3. Jika tidak ada perbaikan atau sideways:

- Berhenti

3. def linearpos\_to\_3dpos(self, num):

Fungsi untuk mengkonversi posisi linear (1D) ke posisi 3D dengan :

- Input: posisi linear (0 sampai  $n^3-1$ )
- Output: array [i,j,k] yang merepresentasikan posisi dalam kubus 3D
- Menggunakan operasi pembagian dan modulo untuk mengkonversi

4. def from\_3dpos\_to\_linearpos(self, pos):

Fungsi untuk mengkonversi posisi 3D ke posisi linear dengan :

- Input: array [i,j,k] posisi dalam kubus 3D
- Output: posisi linear (0 sampai  $n^3-1$ )
- Menggunakan formula:  $i * n^2 + j * n + k$

5. def print\_value(self):

Fungsi untuk mencetak nilai kubus saat ini menggunakan metode print\_value() dari kelas Cube.

6. def plot\_value(self, initial\_cube\_data, final\_cube\_data):

Fungsi untuk melakukan visualisasi statistik berupa objective function pada tiap iterasi dalam bentuk grafik dan juga visualisasi kubus 3d konfigurasi awal dan konfigurasi akhir dengan bantuan kakas matplotlib.

## 2.2.6 Kelas RandomRestartHillClimbing

```
from algorithm.Cube import *
from algorithm.HillClimbingSearch import *
import matplotlib.pyplot as plt
import time

class RandomRestartHillClimbing:
    def __init__(self, max_restarts, n=5):
        self.n = n
        self.max_restarts = max_restarts
        self(cube) = Cube(5)
        self.all_values = []
        self.restart = 0
        self.best_initial_state = []
        self.best_final_state = []
        self.best_value = 0

    def solve(self):
        listSwaps = []
        listInits = []

        start_time = time.time()
```

```
for i in range(self.max_restarts):
    print(f"\nRandom Restart Hill Climbing iteration {i + 1}")
    self.restart += 1

    cube = HillClimbingSearch(5)
    tempSwap, tempInit = cube.solve()
    listSwaps.append(tempSwap)
    listInits.append(tempInit)

    print(f"\nHill Climbing with {cube.getIterations()} iterations and best value {cube.getCurrentValue()}")


    self.all_values.append(cube.list_of_values())

    if cube.getCurrentValue() >= self.best_value:
        self.best_value = cube.getCurrentValue()
        self.best_initial_state = cube.getInitialState()
        self.best_final_state = cube.getFinalState()

    if (cube.getCurrentValue() == 109):
        print("Solved")
        break

end_time = time.time()
duration = end_time - start_time
print(f"\n\nSteepest Ascent Algorithm Duration: {duration:.4f} seconds")

self.cube = cube.cube

print("\nNumber of restarts: ", self.restart)
self.plot_value()
return listSwaps, listInits

def print_value(self):
    self.cube.print_value()

def plot_value(self):
    # Cube plot
    fig1 = plt.figure(figsize=(12, 6))
    ax1 = fig1.add_subplot(121, projection='3d')
    self.cube.plot_number_cube(ax1, self.best_initial_state, "Best Initial Configuration")
    ax2 = fig1.add_subplot(122, projection='3d')
    self.cube.plot_number_cube(ax2, self.best_final_state, "Best Final Configuration")
    ax1.view_init(elev=30, azim=30)
    ax2.view_init(elev=30, azim=30)
    plt.tight_layout()

    plt.figure(figsize=(12, 6))

    for i, values in enumerate(self.all_values):
        plt.plot(values, marker='o', linestyle='-', label=f"Restart {i + 1}")

    plt.title("Cube Value Through Hill Climbing Iterations for Each Restart")
    plt.xlabel("Iteration")
    plt.ylabel("Cube Value")
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()
```

1. def \_\_init\_\_(self, n, max\_restarts):

Ini adalah kelas yang mengimplementasikan Random-Restart Hill Climbing, dimana:

- n: Dimensi kubus (n x n x n)
- max\_restarts: Jumlah maksimum restart yang akan dilakukan
- cube: Instance dari kelas Cube dengan ukuran tetap 5x5x5

2. def solve(self):

Melakukan iterasi sebanyak max\_restarts sesuai pemanggilan fungsi pada setiap iterasi dipanggil fungsi HillClimbingSearchSteepestAscent

3. def print\_value(self):

Fungsi untuk mencetak nilai kubus saat ini menggunakan metode print\_value() dari kelas Cube.

## 2.2.7 Kelas SimulatedAnnealing

```
import math
import random
import matplotlib.pyplot as plt
from algorithm.Cube import *
import copy
import time

class SimulatedAnnealing:
    def __init__(self, max_iterations = 10000, initial_temperature = 1000, cooling_rate = 0.99, n = 5):
        self.cube = Cube(n)
        self.initial_temperature = initial_temperature
        self.cooling_rate = cooling_rate
        self.list_swap_points = []
        self.probabilities = []
        self.threshold = []
        self.n = n
        self.stuck = 0
        self.values = []
        self.max_iterations = max_iterations
        self.iteration = 0

    def solve(self):
        initial_config = copy.deepcopy(self.cube.data)

        start_time = time.time()

        while True and self.iteration < self.max_iterations:
            T = self.temperature(self.iteration)
            if T <= 0:
                break

            self.compare_state(self.cube.generate_random_point(), self.cube.generate_random_point(), T)

            current_value = self.cube.calculate_value()
            self.values.append(current_value)

            self.iteration += 1
```

```
end_time = time.time()
duration = end_time - start_time
print(f"\n\nSteepest Ascent Algorithm Duration: {duration:.4f} seconds")

print("\nLast value: ", self.values[-1])
print("Number of stuck: ", self.stuck, end = "\n\n")
self.plot_value(initial_config, self.cube.data)
return self.list_swap_points, initial_config

def temperature(self, i):
    return self.initial_temperature * pow(self.cooling_rate, i)

def compare_state(self, pos1, pos2, T):
    current_value = self.cube.calculate_value()
    self.cube.swap(pos1, pos2)
    neighbor_value = self.cube.calculate_value()

    deltaE = (neighbor_value - current_value)

    if deltaE >= 0:
        self.probabilities.append(1.0)
        self.threshold.append(0)
        return
    else:
        self.stuck += 1
        threshold = random.uniform(0, 1)
        self.threshold.append(threshold)
        prob = math.exp(deltaE / T)
        self.probabilities.append(prob)
        if T > 0 and prob >= threshold:
            return
        else:
            self.cube.swap(pos1, pos2)
            self.list_swap_points.append([self.from_3dpos_to_linearpos(pos1), self.from_3dpos_to_linearpos(pos2)])

def from_3dpos_to_linearpos(self, pos):
    return pos[0] * (self.n**2) + pos[1] * (self.n) + pos[2]

def print_value(self):
    self.cube.print_value()

def plot_value(self, initial_cube_data, final_cube_data):
    fig1 = plt.figure(figsize=(12, 6))
    ax1 = fig1.add_subplot(121, projection='3d')
    self.cube.plot_number_cube(ax1, initial_cube_data, "Initial Configuration")
    ax2 = fig1.add_subplot(122, projection='3d')
    self.cube.plot_number_cube(ax2, final_cube_data, "Final Configuration")
    ax1.view_init(elev=30, azim=30)
    ax2.view_init(elev=30, azim=30)
    plt.tight_layout()

    fig2, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

    filtered_probabilities = [prob for prob in self.probabilities if prob < 1.0]
    filtered_indices = [idx for idx, prob in enumerate(self.probabilities) if prob < 1.0]

    ax1.plot(filtered_indices, filtered_probabilities, label='Probability', linestyle='--')
    ax1.set_title("Probability Values")
    ax1.set_xlabel("Iteration")
    ax1.set_ylabel("Probability Values")
```

```
ax1.grid()
ax1.legend()

ax2.plot(self.values, color='g', label='Cube Value', linestyle='--')
ax2.set_title("Cube Values Throughout Simulated Annealing")
ax2.set_xlabel("Iteration")
ax2.set_ylabel("Cube Value")
ax2.grid()
ax2.legend()

plt.show()
```

1. def \_\_init\_\_(self, n, iterations, initial\_temperature = 1000, cooling\_rate = 0.99):  
Ini adalah kelas yang mengimplementasikan Random-Restart Hill Climbing, dimana:
  - n: Dimensi kubus
  - iterations: Jumlah iterasi maksimum
  - initial\_temperature: Suhu awal (default 1000)
  - cooling\_rate: Tingkat pendinginan (default 0.99)
  - list\_swap\_points: Menyimpan semua pertukaran yang dilakukan
  - self.n: Dimensi kubus untuk konversi posisi
2. def solve(self):  
Fungsi utama yang mengimplementasikan SimulatedAnnealing:
  - Menyimpan konfigurasi awal
  - Melakukan iterasi dengan:
    - Menghitung suhu untuk iterasi ini
    - Berhenti jika suhu mencapai 0
    - Membandingkan state dengan dua titik acak
3. def temperature(self, i):  
Menghitung suhu untuk iterasi ke-i  
Menggunakan fungsi exponential decay:  $T = T_0 * (\alpha^i)$ , dengan :
  - T0: initial\_temperature
  - $\alpha$ : cooling\_rate
  - i: iterasi ke-i
4. def compare\_state(self, pos1, pos2, T):  
Fungsi ini implementasi inti dari Simulated Annealing:
  - a) Pengecekan suhu:
    - Berhenti jika suhu sudah 0
  - b) Evaluasi state:
    - Menghitung nilai current state

- Mencoba pertukaran
- Menghitung perubahan nilai (deltaE)

c) Pengambilan keputusan:

- Jika perubahan positif/0:
  - Terima perubahan
- Jika negatif:
  - Hitung probabilitas penerimaan
  - Bandingkan dengan threshold
  - Kembalikan ke state sebelumnya jika ditolak

5. def from\_3dpos\_to\_linearpos(self,pos):

Fungsi untuk mengkonversi posisi 3D ke posisi linear dengan :

- Input: array [i,j,k] posisi dalam kubus 3D
- Output: posisi linear (0 sampai  $n^3-1$ )
- Menggunakan formula:  $i * n^2 + j * n + k$

6. def print\_value(self):

Fungsi untuk mencetak nilai kubus saat ini menggunakan metode print\_value() dari kelas Cube.

7. def plot\_value(self, initial\_cube\_data, final\_cube\_data):

Fungsi untuk melakukan visualisasi statistik berupa objective function pada tiap iterasi dalam bentuk grafik dan juga visualisasi kubus 3d konfigurasi awal dan konfigurasi akhir dengan bantuan kakas matplotlib.

## 2.2.7 Kelas StochasticHillClimbing

```
from algorithm.Cube import *
import matplotlib.pyplot as plt
import copy
import time

class StochasticHillClimbing:
    def __init__(self, iterations, n=5):
        self.n = n
        self(cube) = Cube(n)
        self.iterations = iterations
        self.list_swap_points = []
        self.values = []
        self.num_iterations = 0

    def solve(self):
        initial_configuration = copy.deepcopy(self(cube).data)

        start_time = time.time()

        for i in range(self.iterations):
```

```
    self.num_iterations += 1
    end = self.compare_state(self.cube.generate_random_point(), self.cube.generate_random_point())
    if end:
        break

    end_time = time.time()
    duration = end_time - start_time
    print(f"\n\nSteepest Ascent Algorithm Duration: {duration:.4f} seconds")

    print("\nBest value: ", self.values[-1])
    print("Number of iterations: ", self.num_iterations)

    self.plot_value(initial_configuration, self.cube.data)
    return self.list_swap_points, initial_configuration

def compare_state(self, pos1, pos2):
    current_value = self.cube.calculate_value()
    self.values.append(current_value)
    self.cube.swap(pos1, pos2)
    neighbor_value = self.cube.calculate_value()

    if current_value >= neighbor_value:
        self.cube.swap(pos1, pos2)
    else:
        print(f"Swap {pos1} and {pos2}, current value = {current_value}")
        self.list_swap_points.append([self.from_3dpos_to_linearpos(pos1), self.from_3dpos_to_linearpos(pos2)])

    return neighbor_value == 109

def print_value(self):
    self.cube.print_value()

def from_3dpos_to_linearpos(self, pos):
    return pos[0] * (self.n**2) + pos[1] * (self.n) + pos[2]

def plot_value(self, initial_cube_data, final_cube_data):
    fig1 = plt.figure(figsize=(12, 6))
    ax1 = fig1.add_subplot(121, projection='3d')
    self.cube.plot_number_cube(ax1, initial_cube_data, "Initial Configuration")
    ax2 = fig1.add_subplot(122, projection='3d')
    self.cube.plot_number_cube(ax2, final_cube_data, "Final Configuration")
    ax1.view_init(elev=30, azim=30)
    ax2.view_init(elev=30, azim=30)
    plt.tight_layout()

    plt.figure(figsize=(12, 6))
    plt.plot(self.values, linestyle='--')
    plt.title("Stochastic Hill Climbing")
    plt.xlabel("Iteration")
    plt.ylabel("Objective Function Value")
    plt.grid()

    plt.show()
```

1. def \_\_init\_\_(self, n, iterations):

Ini adalah kelas yang mengimplementasikan StochasticHillClimbing, dimana:

- Membuat instance kubus dengan ukuran  $n \times n \times n$
- Menentukan jumlah iterasi maksimum

- Membuat list kosong untuk menyimpan titik-titik pertukaran

2. def solve(self):

Fungsi utama yang mengimplementasikan StochasticHillClimbing:

- Menyimpan konfigurasi awal kubus
- Melakukan iterasi sebanyak yang ditentukan
- Pada setiap iterasi, membandingkan dua titik random
- Berhenti jika solusi optimal ditemukan (end = True)
- Mengembalikan list pertukaran titik dan konfigurasi awal

3. def compare\_state(self, pos1, pos2):

Fungsi ini implementasi inti dari StochasticHillClimbing:

- Menghitung nilai kubus saat ini
- Mencoba menukar dua posisi
- Menghitung nilai setelah pertukaran
- Jika nilai baru lebih buruk, kembalikan ke posisi semula
- Menyimpan pertukaran yang dilakukan
- Mengecek apakah nilai optimal (109) sudah tercapai

4. def from\_3dpos\_to\_linearpos(self,pos):

Fungsi untuk mengkonversi posisi 3D ke posisi linear dengan :

- Input: array [i,j,k] posisi dalam kubus 3D
- Output: posisi linear (0 sampai  $n^3-1$ )
- Menggunakan formula:  $i * n^2 + j * n + k$

5. def print\_value(self):

Fungsi untuk mencetak nilai kubus saat ini menggunakan metode print\_value() dari kelas Cube.

## 2.3 Hasil eksperimen dan analisis

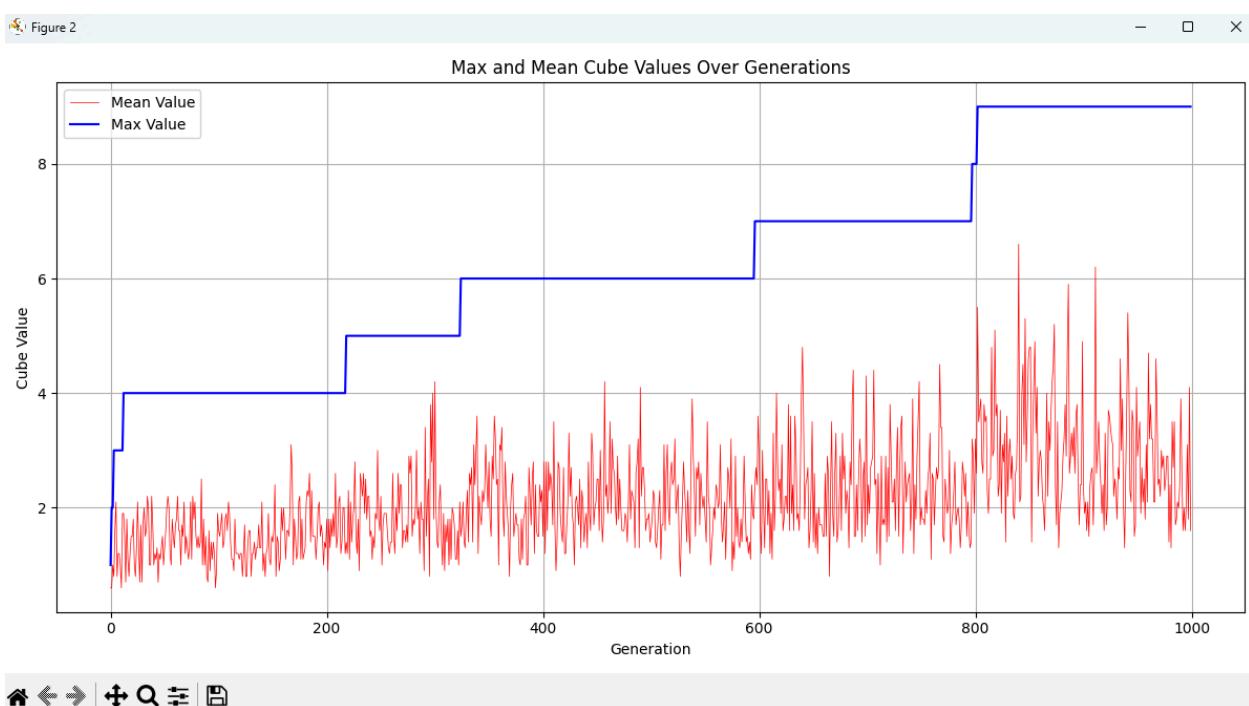
### 2.3.1 Genetic Algorithm

Untuk Genetic Algorithm kami memilih untuk memvisualisasikan 2 kubus pada populasi awal dan populasi akhir dengan nilai objective function tertinggi dari masing-masing populasi. Selain itu untuk statistik kami memvisualisasikan nilai objective function maksimum untuk tiap iterasi (generasi) dengan ditandai garis biru pada grafik dan juga memvisualisasikan nilai rata-rata objective function pada populasi

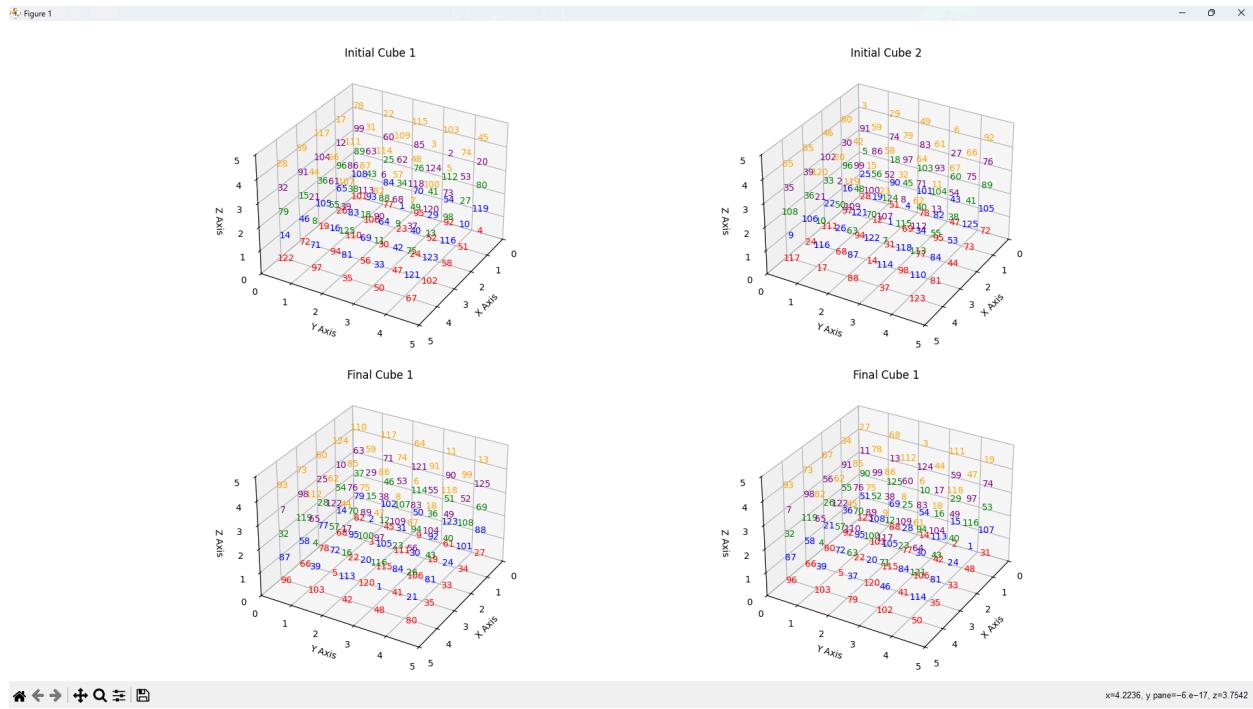
1. Populasi sebagai kontrol dengan jumlah 10

a. Iterasi 1000 kali

i. Percobaan 1



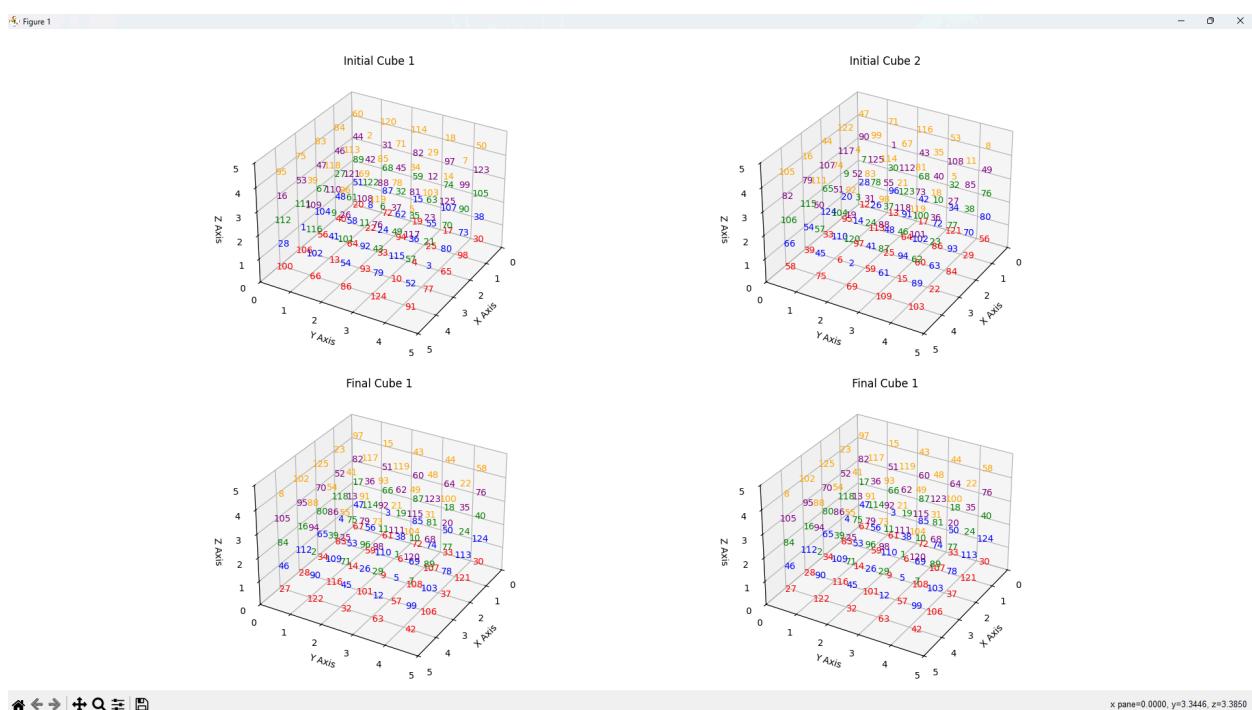
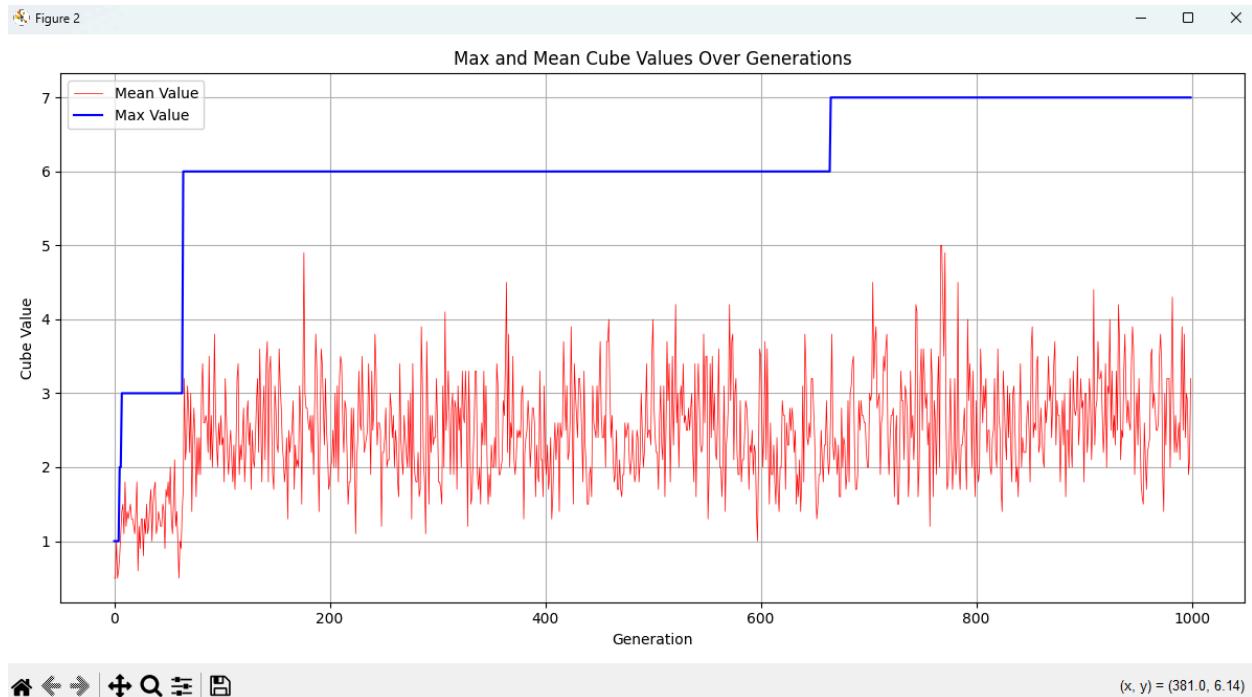
Tugas Besar 1 IF 3170 Intelegrasi Artificial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search



**RESULT**  
**Genetic Algorithm Duration:** 14.9223 seconds  
**Best value:** 9  
**Number of population:** 10  
**Number of iterations:** 1000

Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 9 dan durasi eksekusinya adalah 14.9223 detik

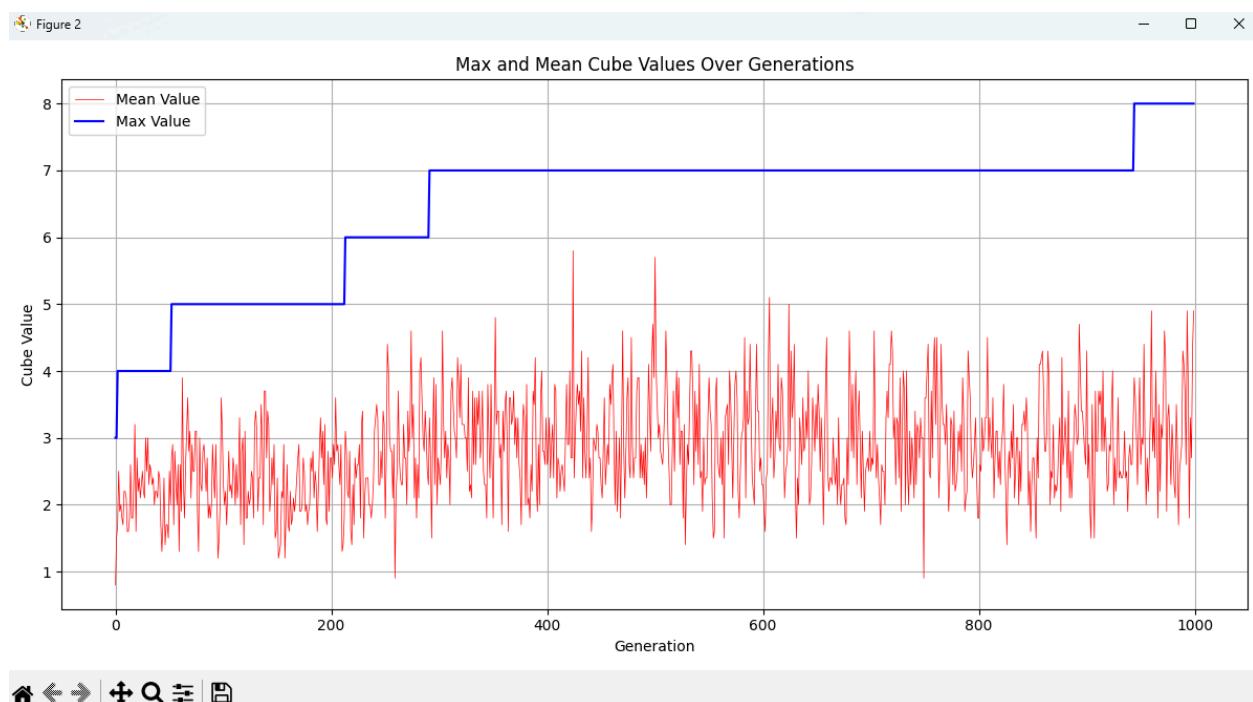
## ii. Percobaan 2



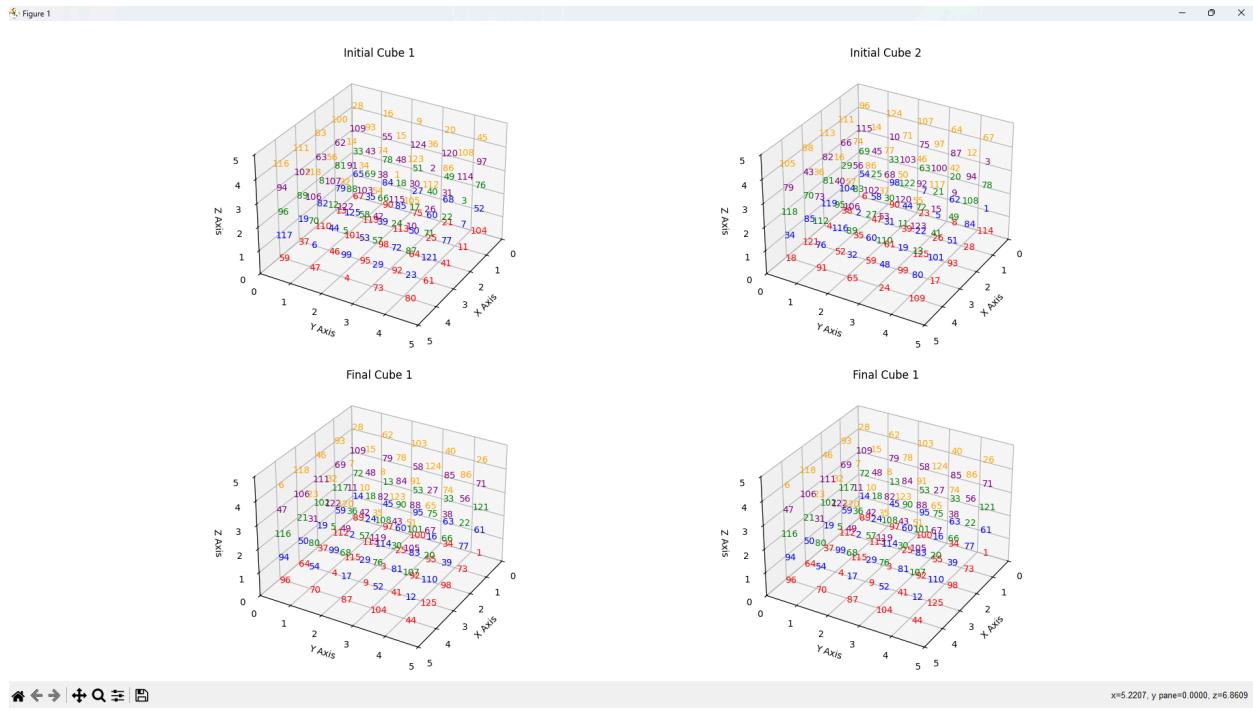
```
RESULT
Genetic Algorithm Duration: 14.3200 seconds
Best value: 7
Number of population: 10
Number of iterations: 1000
```

Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 7 dan durasi eksekusinya adalah 14.3200 detik

### iii. Percobaan 3



Tugas Besar 1 IF 3170 Intelegrasi Artificial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search



```

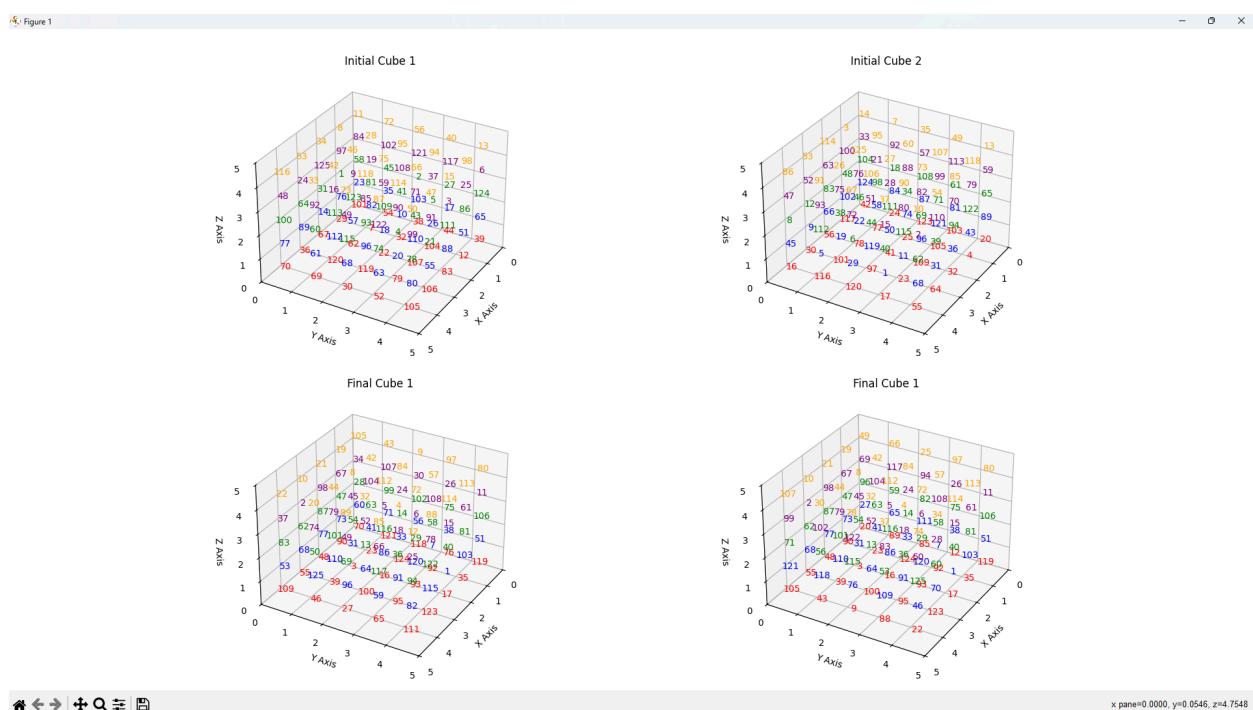
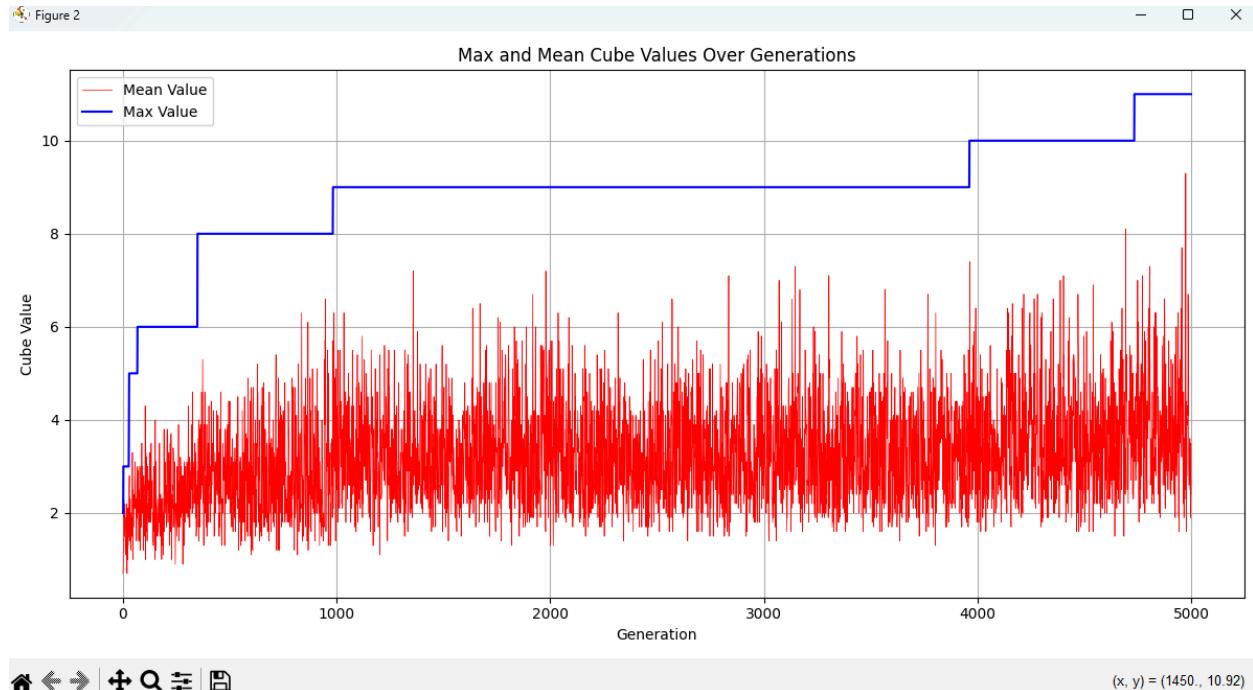
RESULT
Genetic Algorithm Duration: 14.4838 seconds
Best value: 8
Number of population: 10
Number of iterations: 1000

```

Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 8 dan durasi eksekusinya adalah 14.4838 detik

- b. Iterasi 5000 kali
  - i. Percobaan 1

Tugas Besar 1 IF 3170 Intelegrasi Artifisial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search



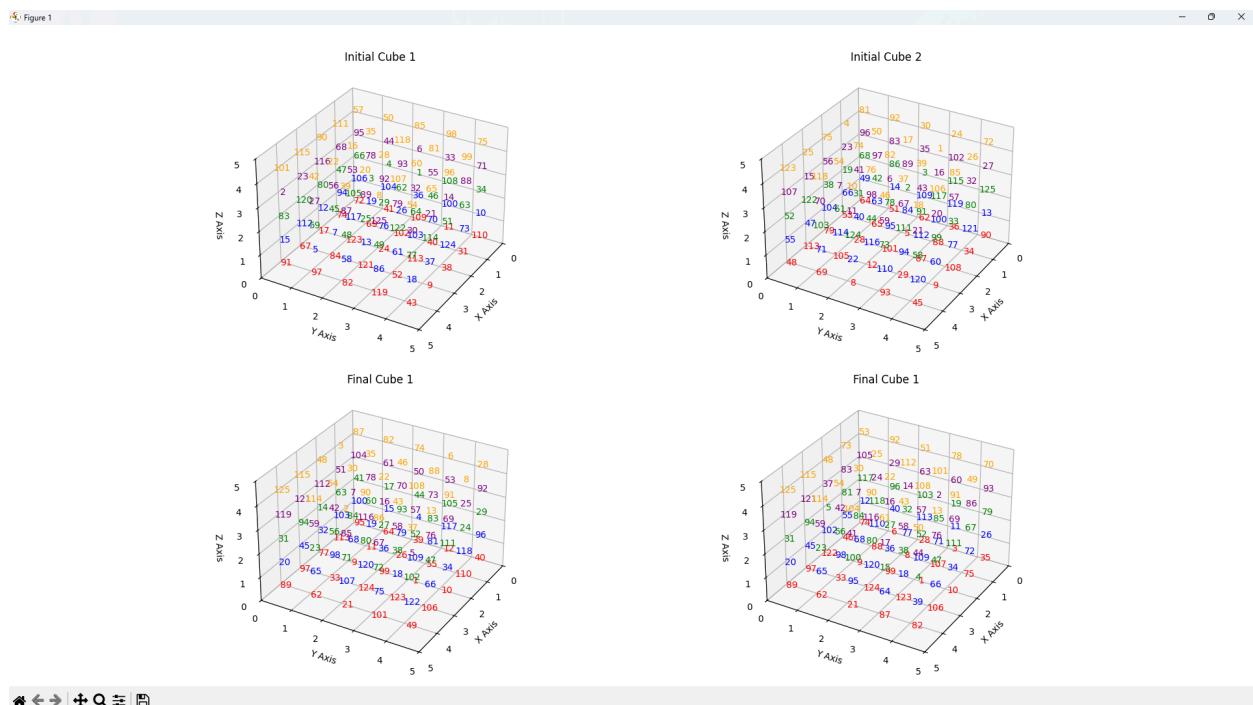
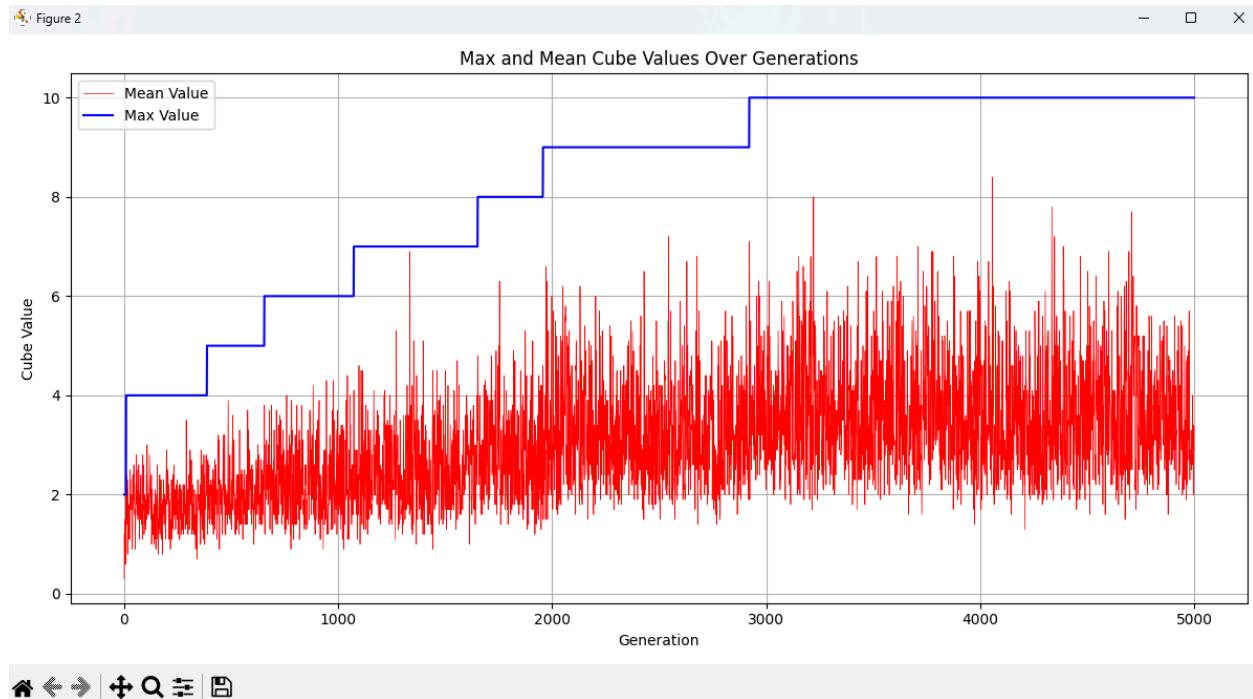
```

RESULT
Genetic Algorithm Duration: 72.7242 seconds
Best value: 11
Number of population: 10
Number of iterations: 5000

```

Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 5000 dan didapatkan best objective function nya sebesar 11 dan durasi eksekusinya adalah 72.7242 detik

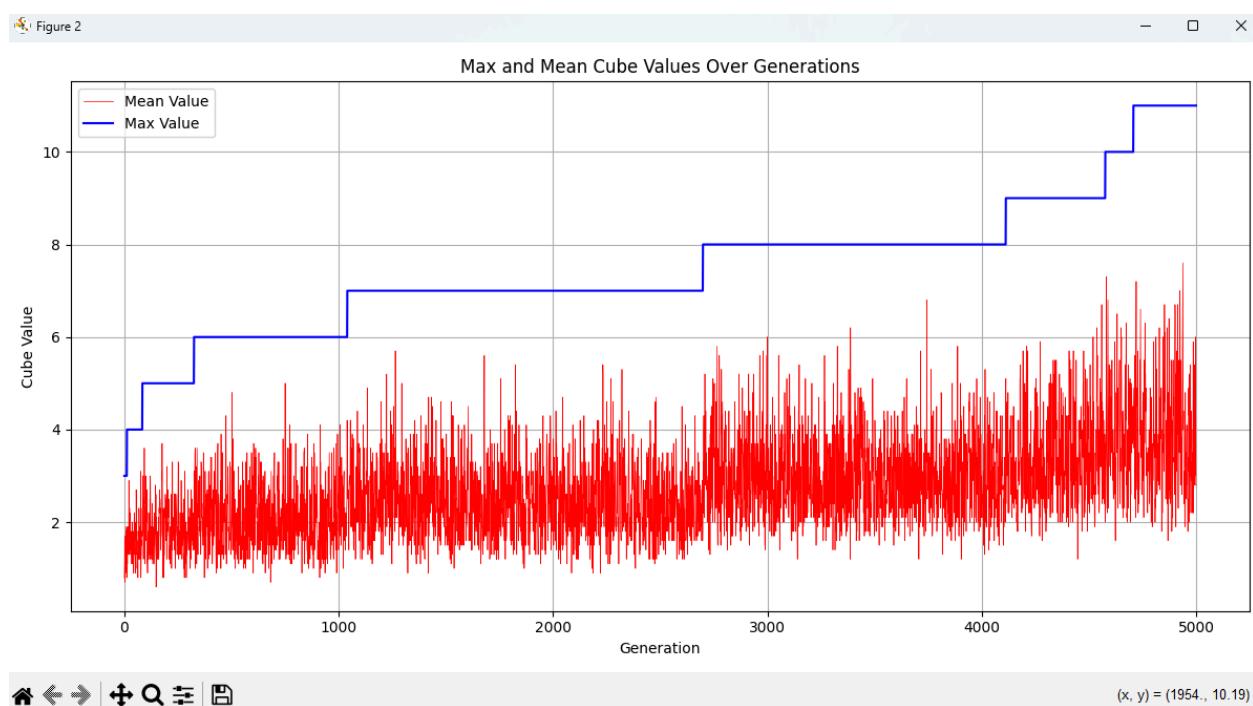
## ii. Percobaan 2



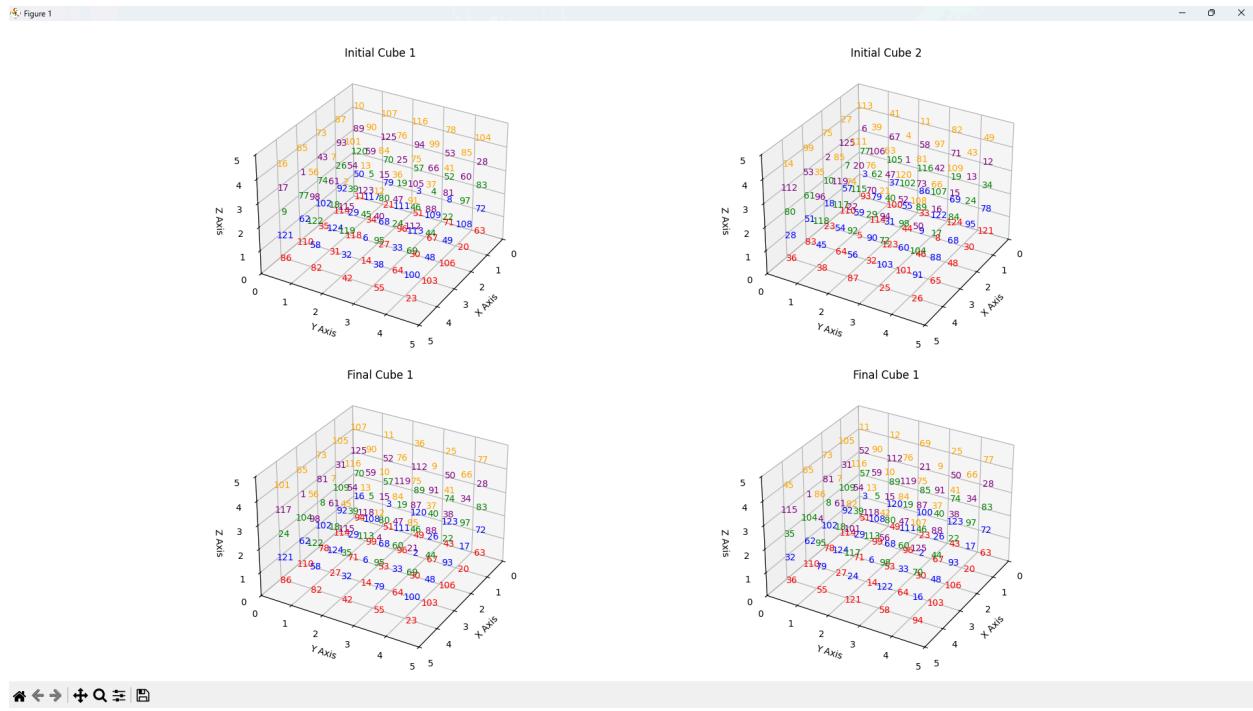
```
RESULT
Genetic Algorithm Duration: 71.6420 seconds
Best value: 10
Number of population: 10
Number of iterations: 5000
```

Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 5000 dan didapatkan best objective function nya sebesar 10 dan durasi eksekusinya adalah 71.6420 detik

### iii. Percobaan 3



Tugas Besar 1 IF 3170 Intelegrasi Artifisial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search



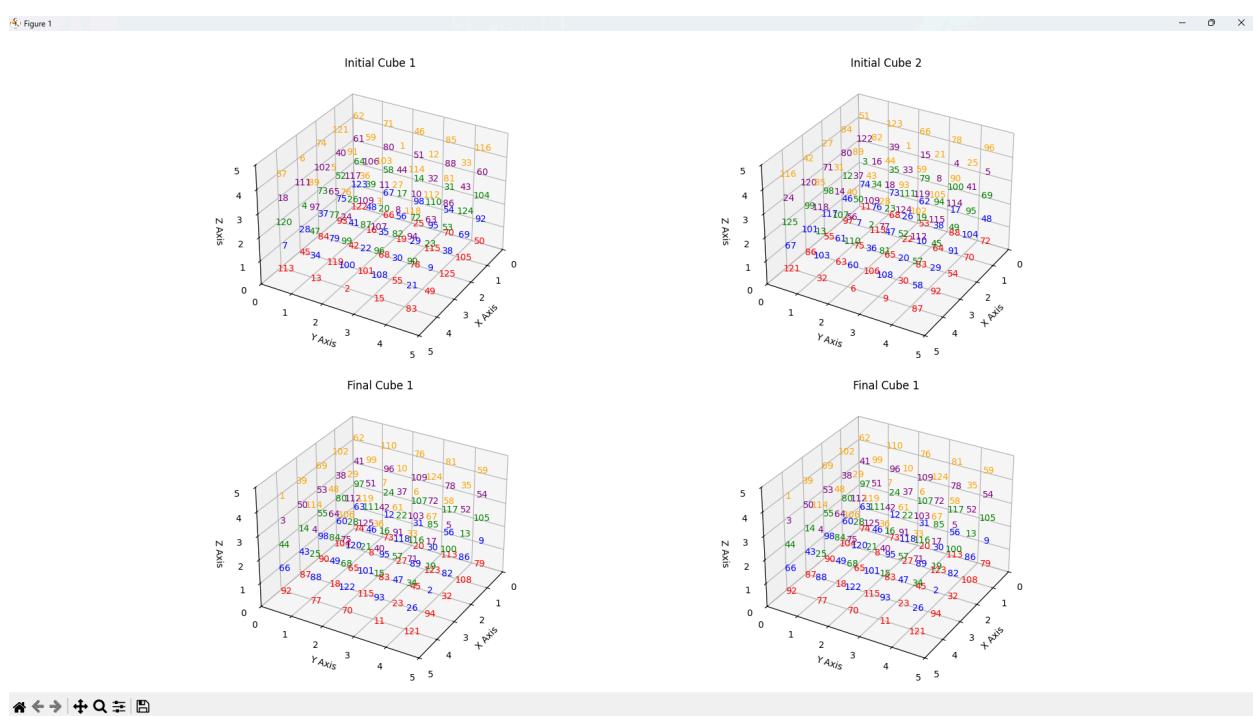
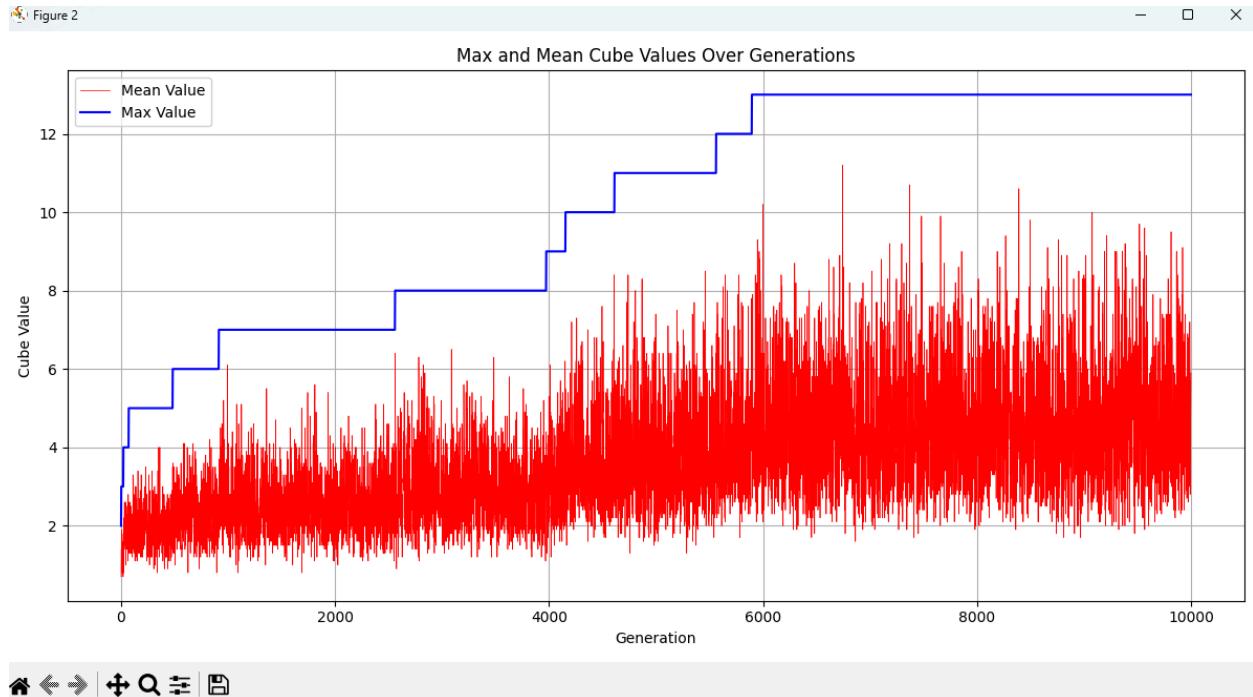
**RESULT**  
**Genetic Algorithm Duration:** 75.5701 seconds  
**Best value:** 11  
**Number of population:** 10  
**Number of iterations:** 5000

Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 5000 dan didapatkan best objective function nya sebesar 11 dan durasi eksekusinya adalah 75.5701 detik

c. Iterasi 10000 kali

i. Percobaan 1

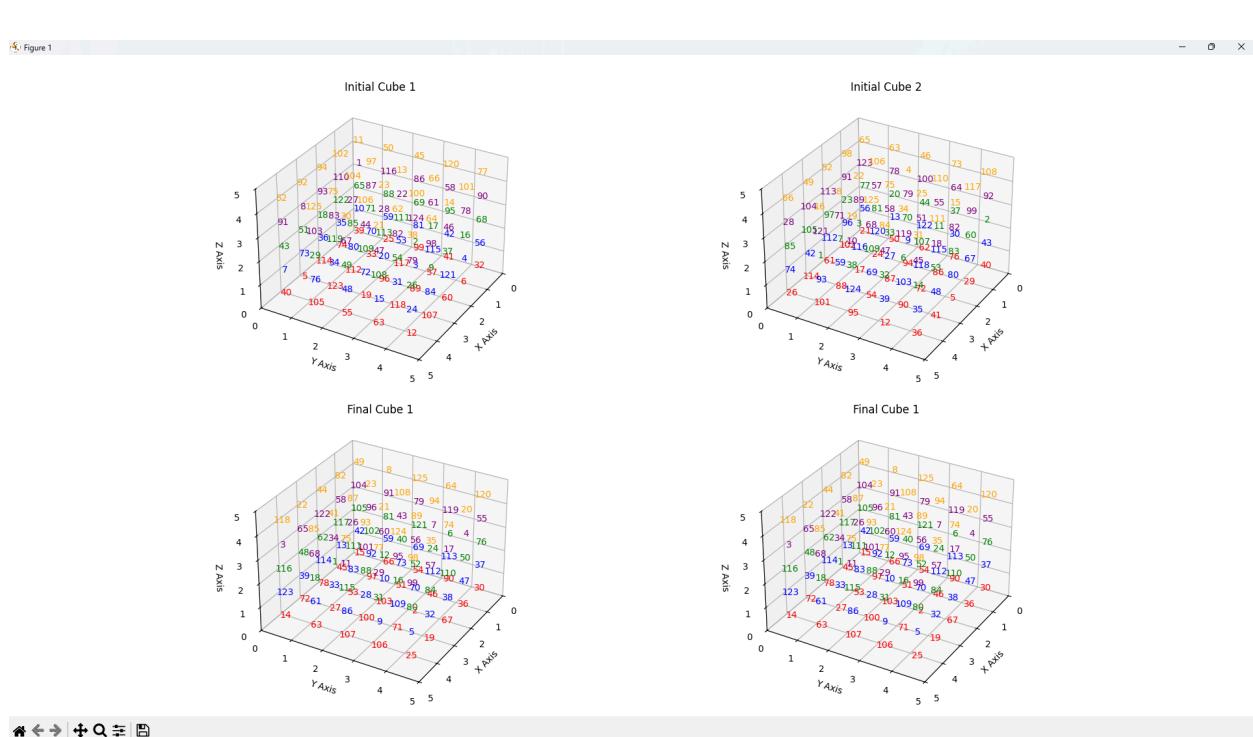
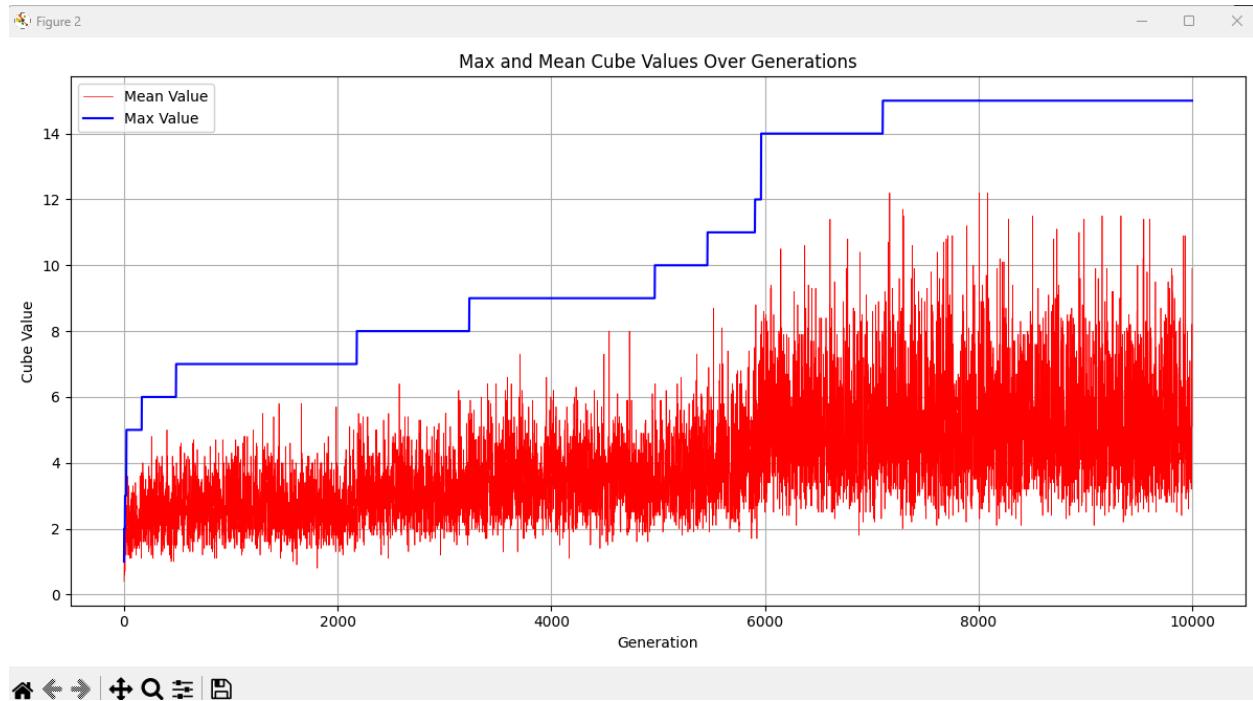
Tugas Besar 1 IF 3170 Intelektualisasi  
Pencarian Solusi Diagonal Magic Cube dengan Local Search



```
RESULT
Genetic Algorithm Duration: 142.3989 seconds
Best value: 13
Number of population: 10
Number of iterations: 10000
```

Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 10000 dan didapatkan best objective function nya sebesar 13 dan durasi eksekusinya adalah 142.3989 detik

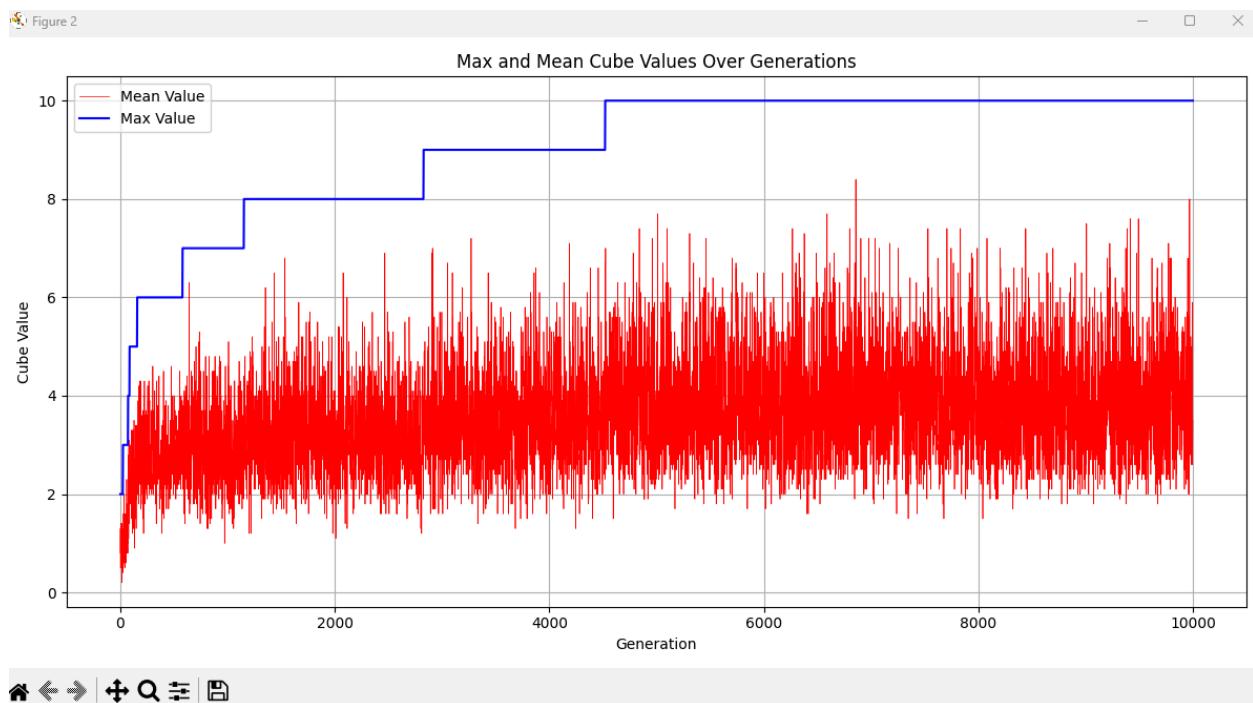
## ii. Percobaan 2



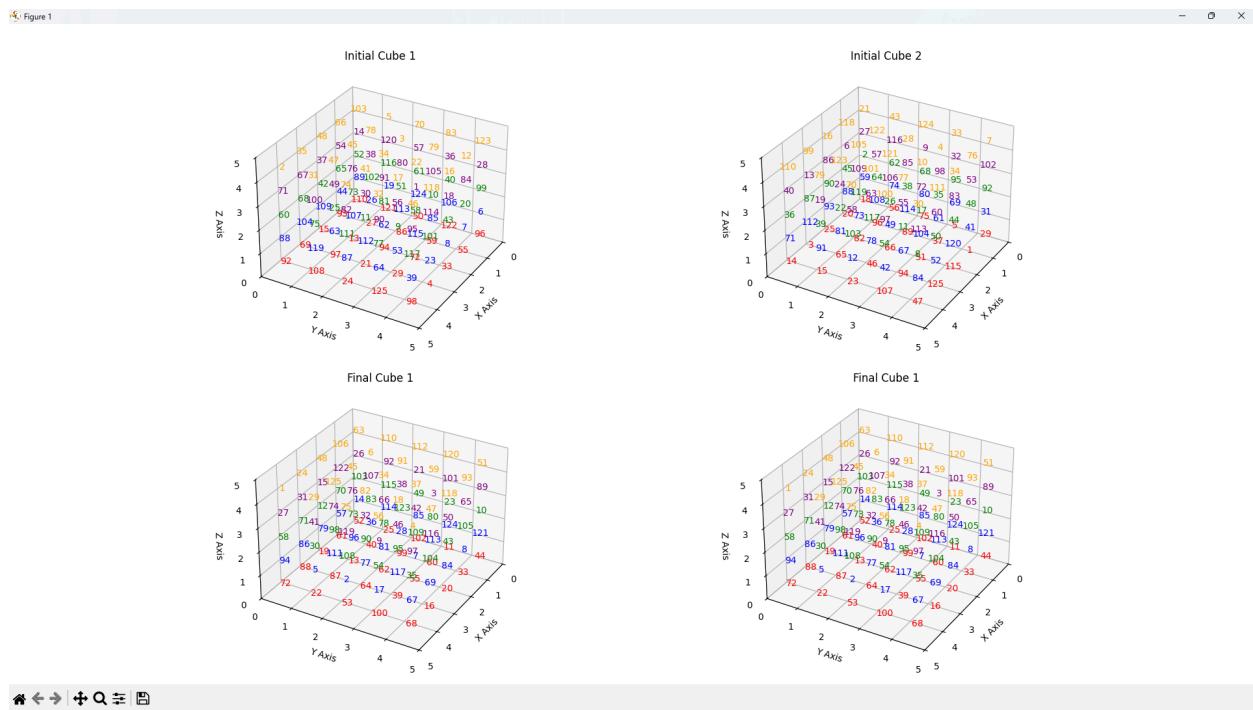
**RESULT**  
**Genetic Algorithm Duration:** 141.4595 seconds  
**Best value:** 15  
**Number of population:** 10  
**Number of iterations:** 10000

Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 10000 dan didapatkan best objective function nya sebesar 15 dan durasi eksekusinya adalah 141.4595 detik

### iii. Percobaan 3



Tugas Besar 1 IF 3170 Intelegrasi Artifisial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search



### RESULT

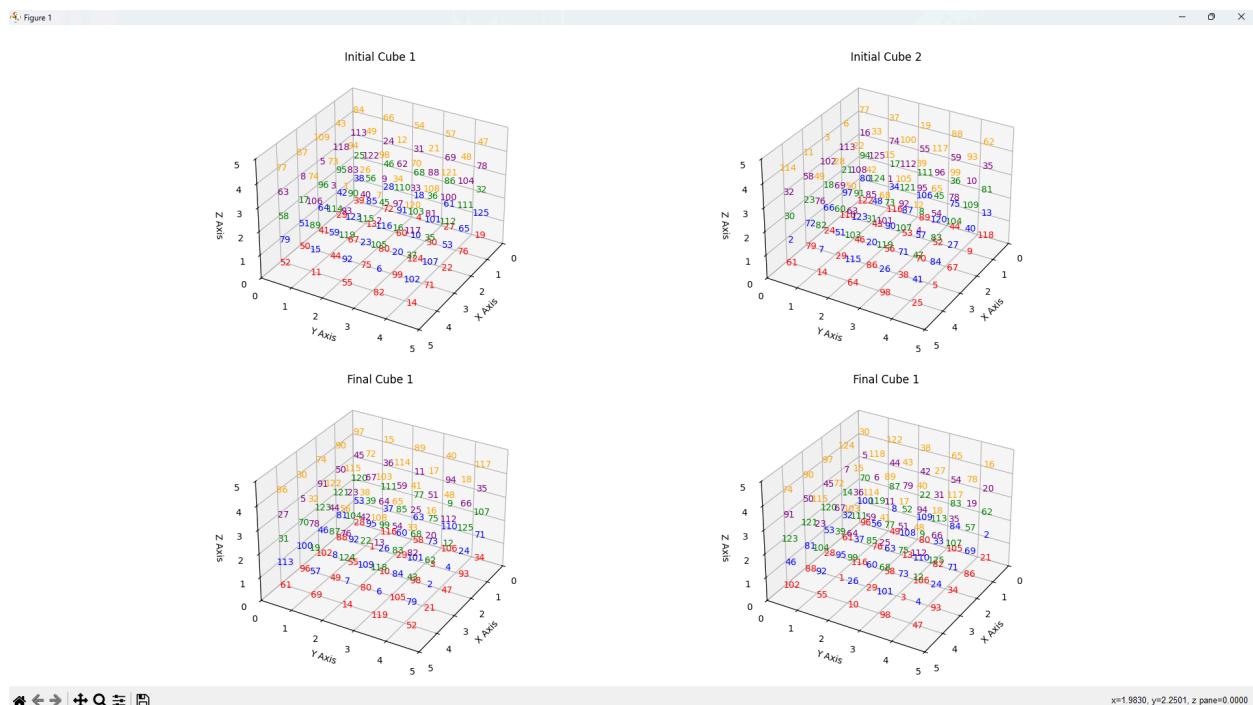
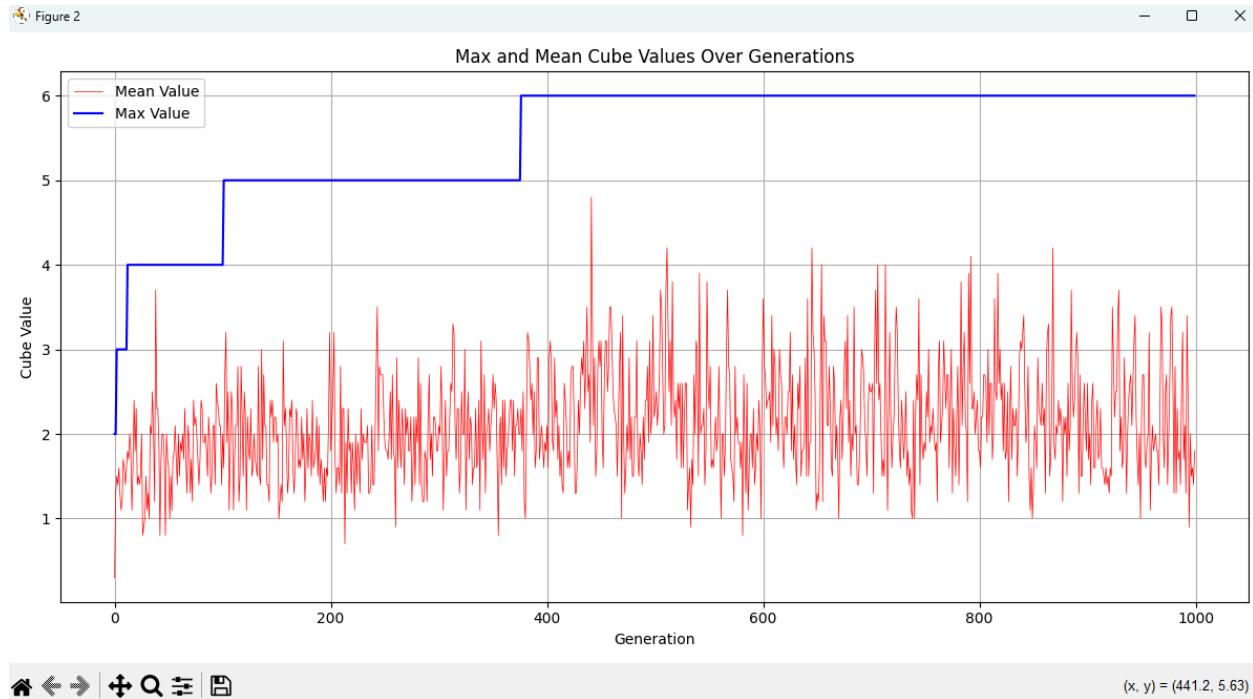
Genetic Algorithm Duration: 143.7858 seconds  
 Best value: 10  
 Number of population: 10  
 Number of iterations: 10000

Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 10000 dan didapatkan best objective function nya sebesar 10 dan durasi eksekusinya adalah 143.7858 detik

## 2. Iterasi sebagai kontrol dengan jumlah 1000

### a. Populasi 10

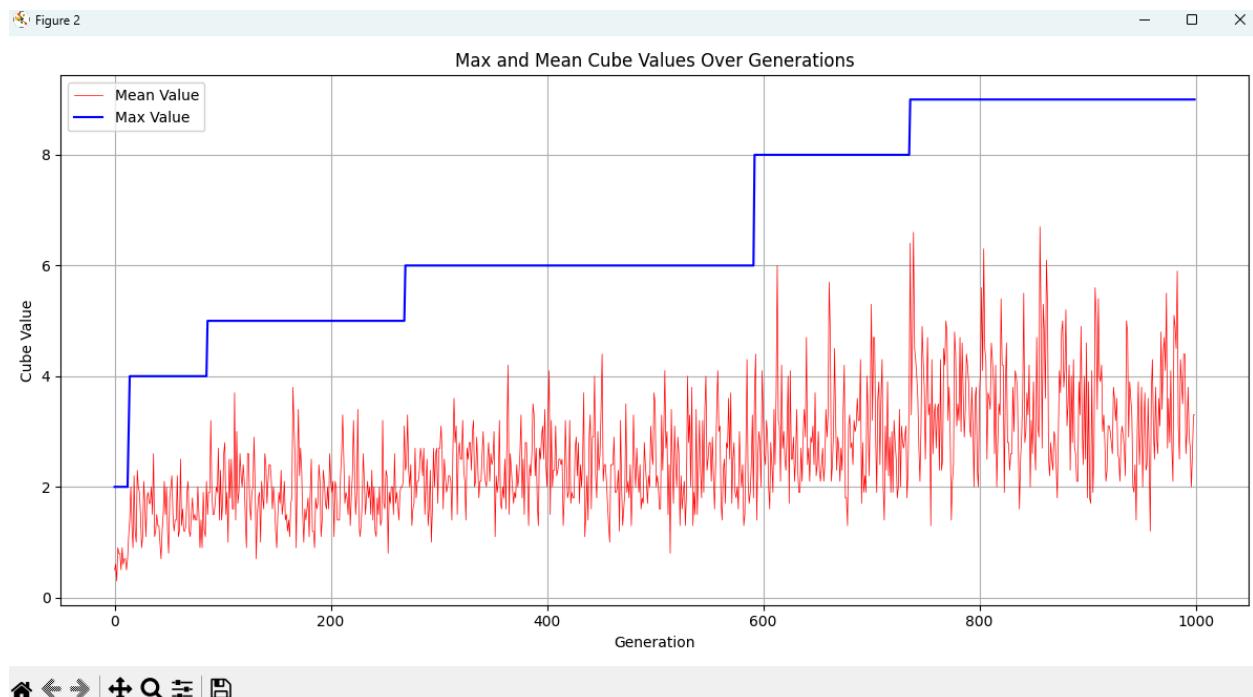
#### i. Percobaan 1



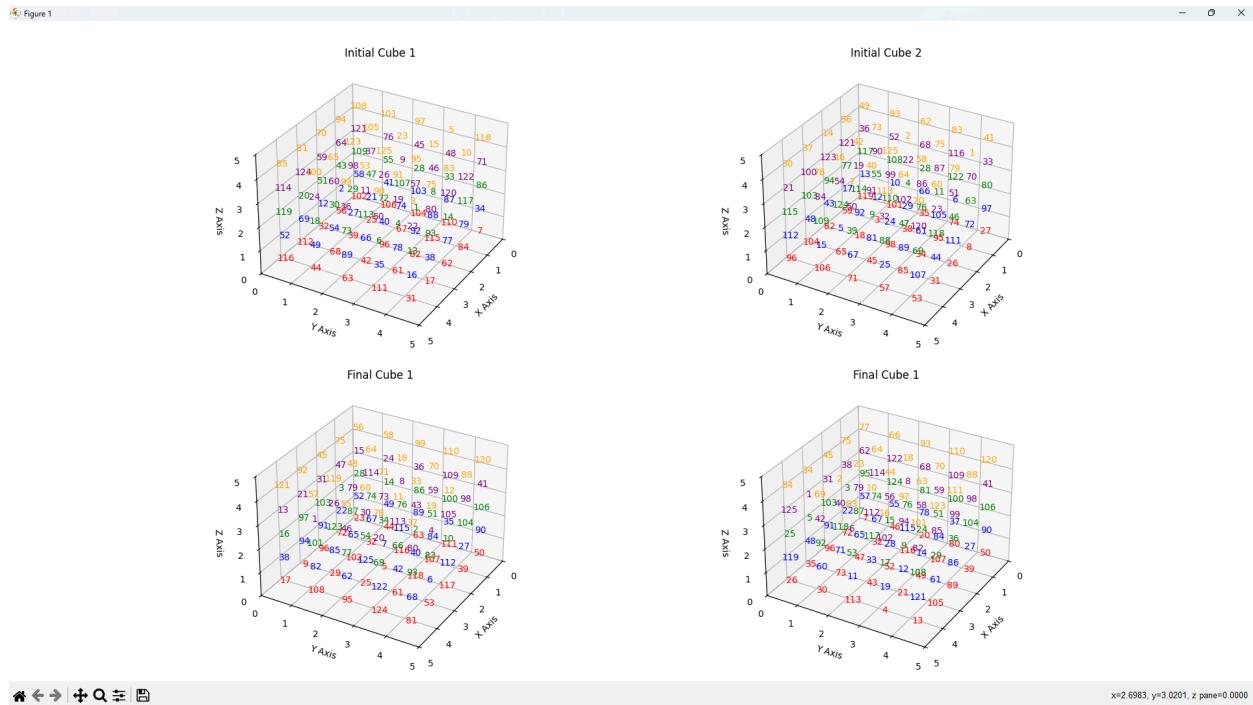
**RESULT**  
**Genetic Algorithm Duration:** 14.6259 seconds  
**Best value:** 6  
**Number of population:** 10  
**Number of iterations:** 1000

Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 6 dan durasi eksekusinya adalah 14.6259 detik

ii. Percobaan 2



Tugas Besar 1 IF 3170 Intelegrasi Artifisial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search



```

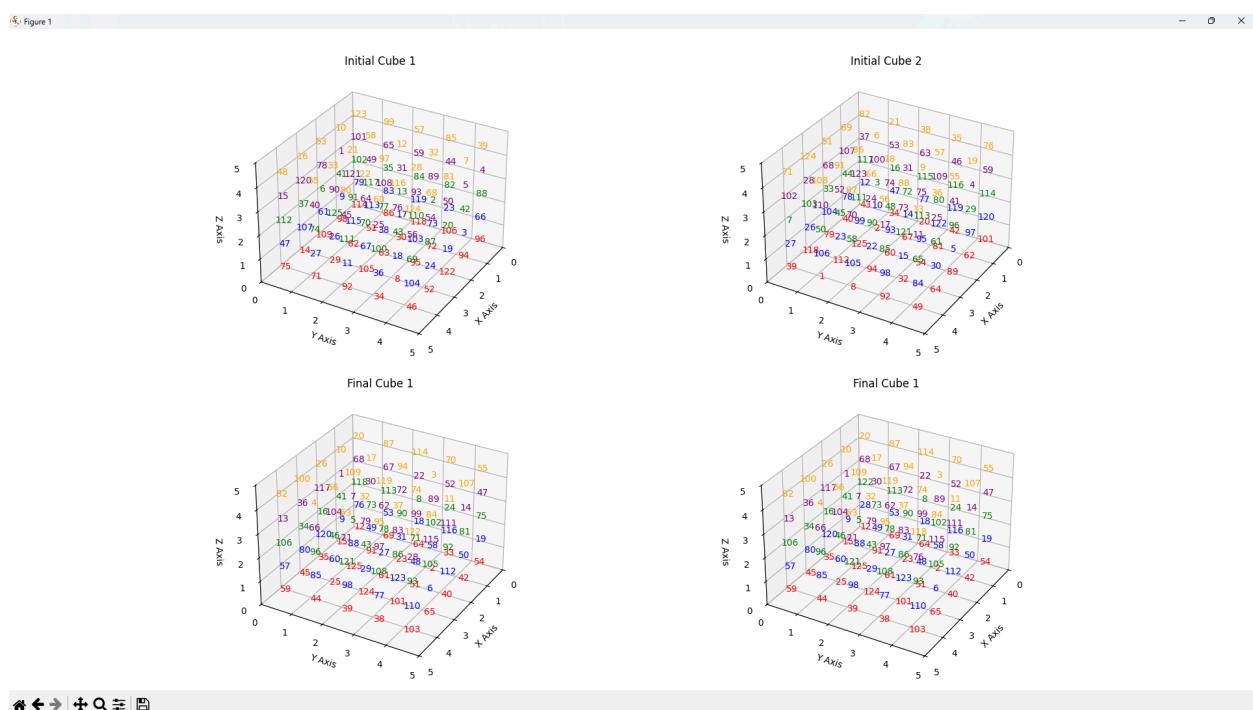
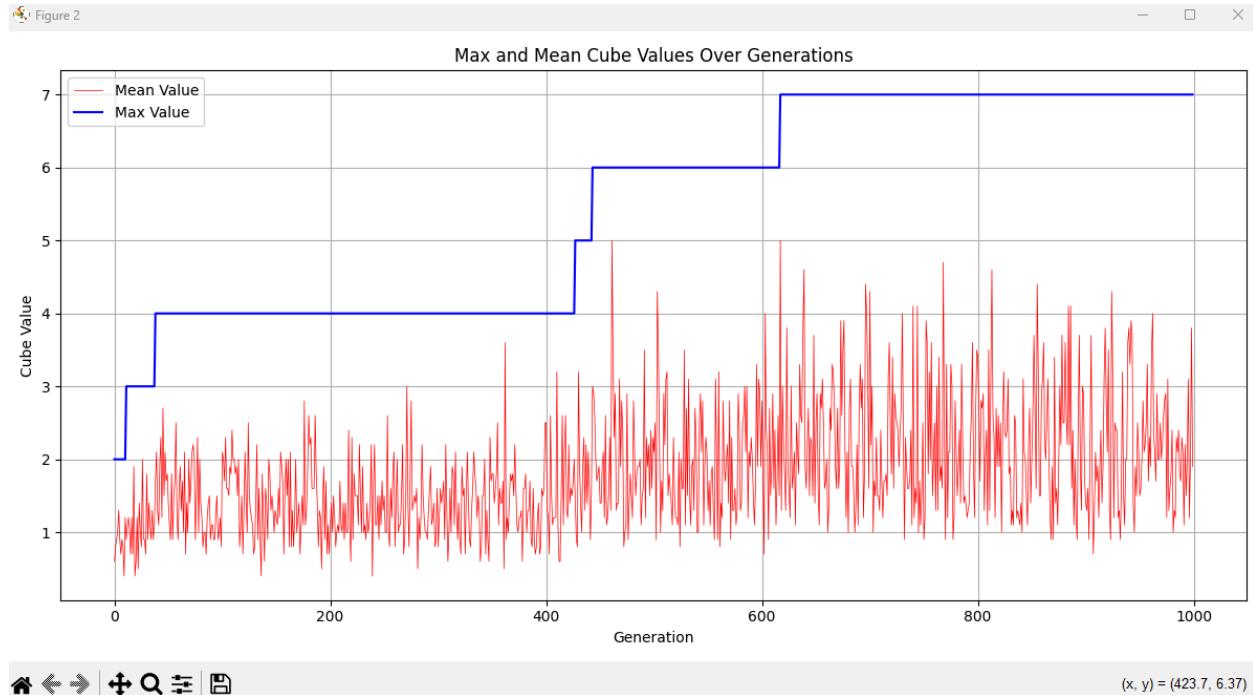
RESULT
Genetic Algorithm Duration: 14.6242 seconds
Best value: 9
Number of population: 10
Number of iterations: 1000

```

Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 9 dan durasi eksekusinya adalah 14.6242 detik

### iii. Percobaan 3

Tugas Besar 1 IF 3170 Intelegrasi Artifisial  
 Pencarian Solusi Diagonal Magic Cube dengan Local Search

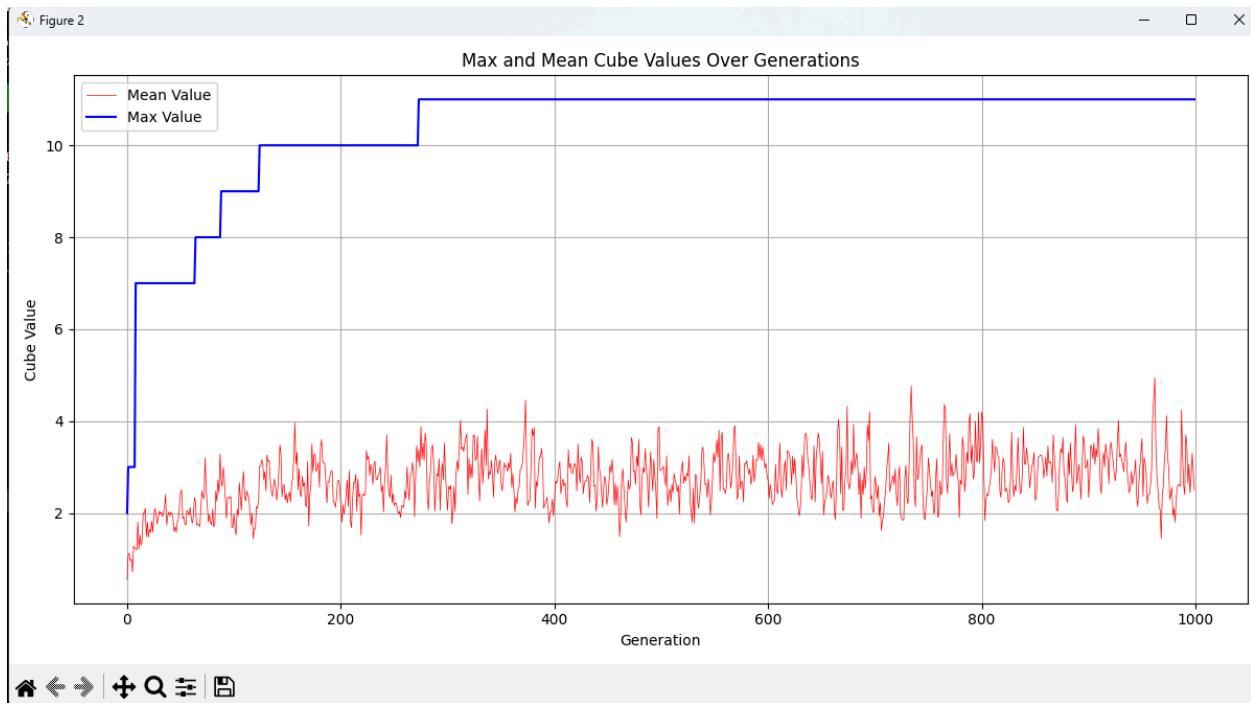


**RESULT**  
**Genetic Algorithm Duration: 14.0448 seconds**  
**Best value: 7**  
**Number of population: 10**  
**Number of iterations: 1000**

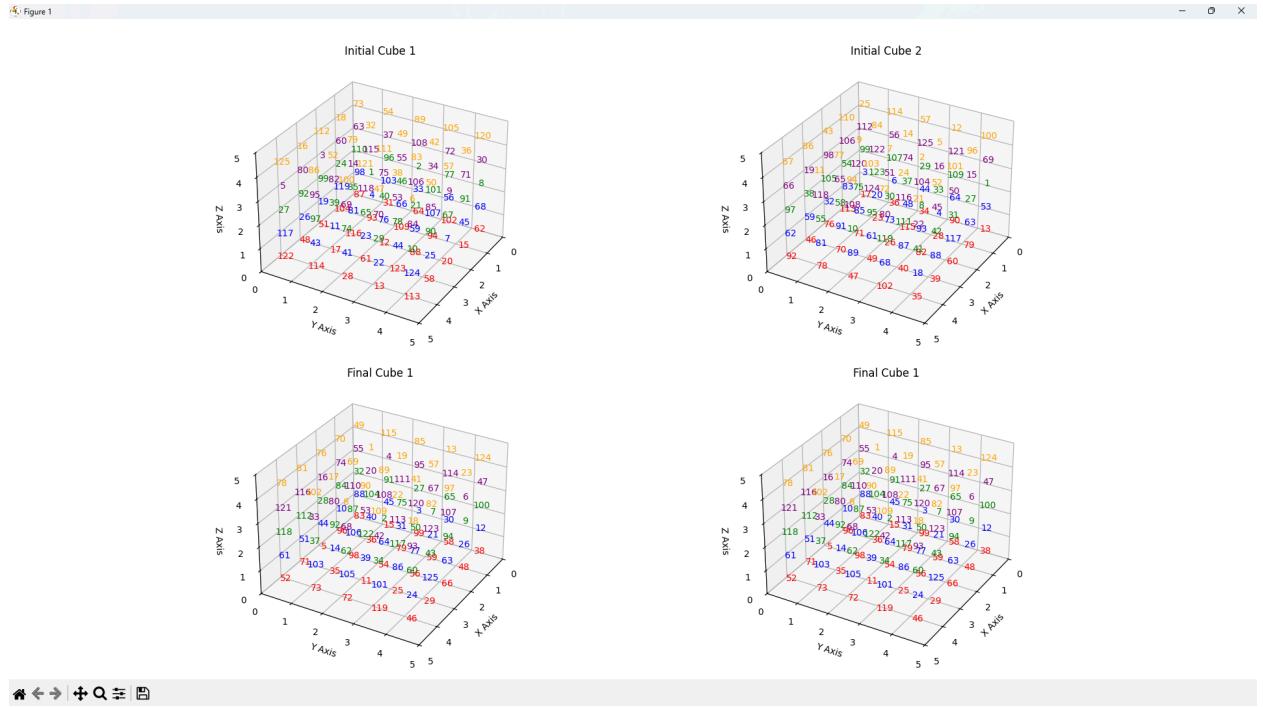
Pada percobaan ini digunakan populasi 10 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 7 dan durasi eksekusinya adalah 14.0448 detik

b. Populasi 50

i. Percobaan 1



Tugas Besar 1 IF 3170 Intelegrasi Artifisial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search

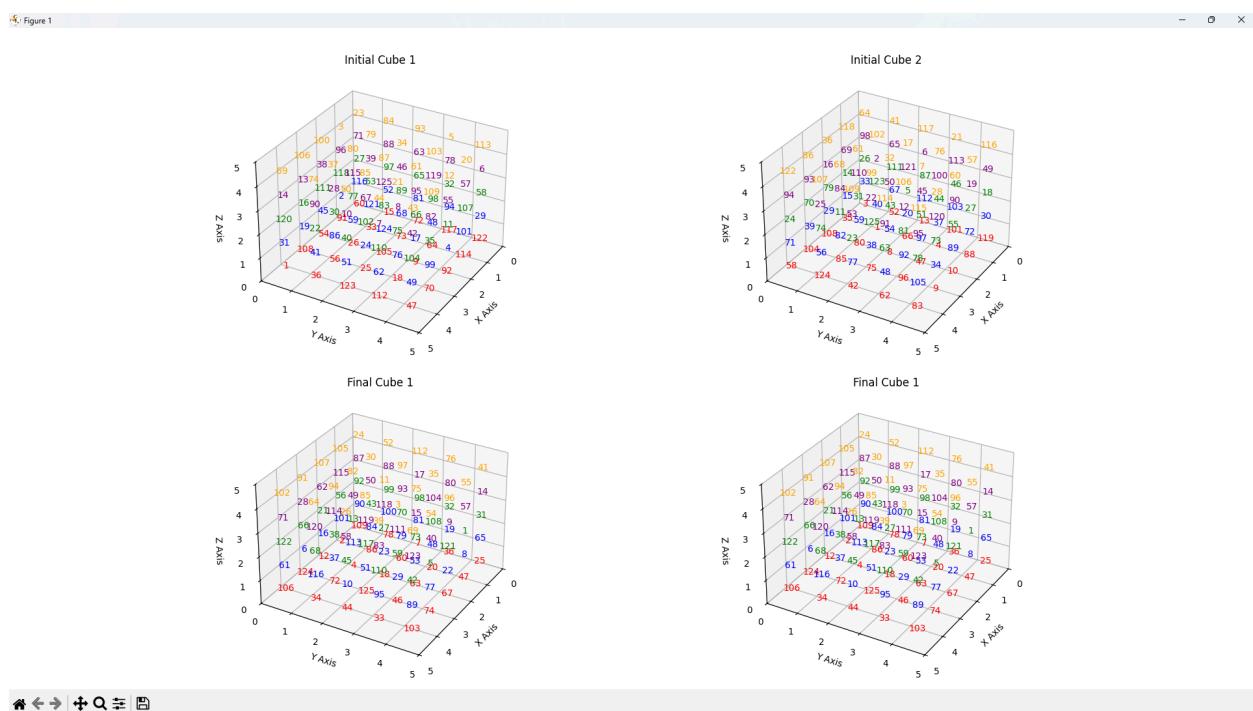
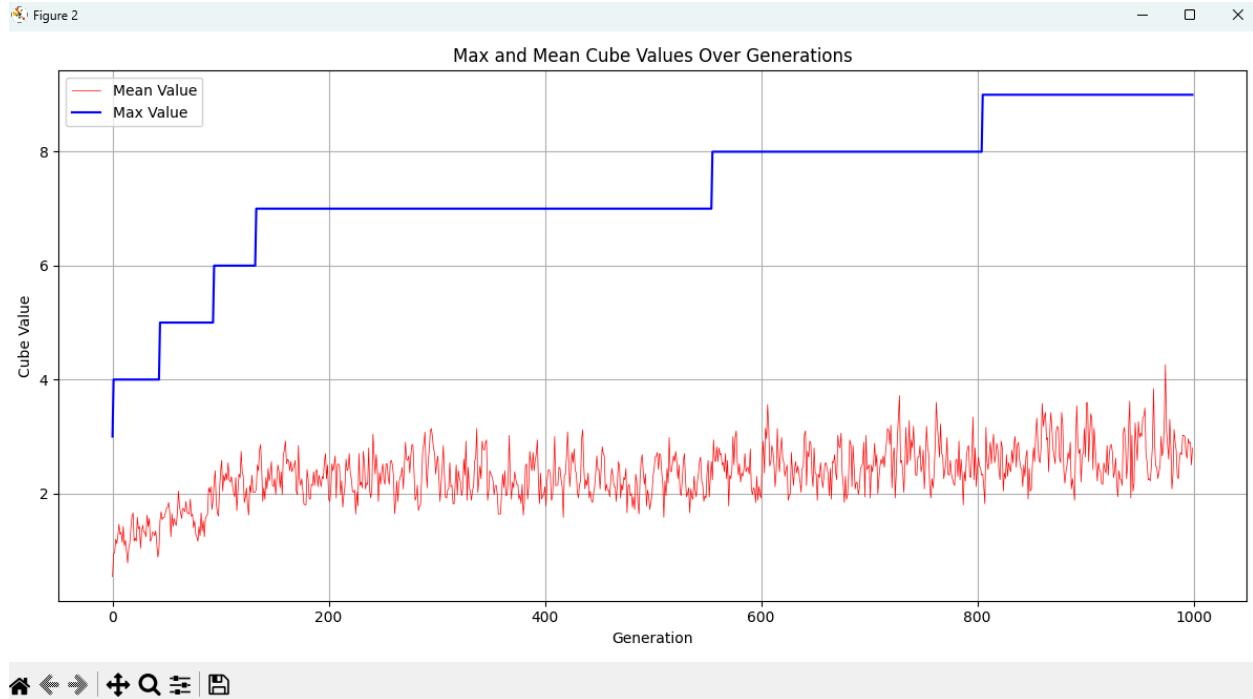


**RESULT**  
**Genetic Algorithm Duration:** 70.7311 seconds  
**Best value:** 11  
**Number of population:** 50  
**Number of iterations:** 1000

Pada percobaan ini digunakan populasi 50 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 11 dan durasi eksekusinya adalah 70.7311 detik

ii. Percobaan 2

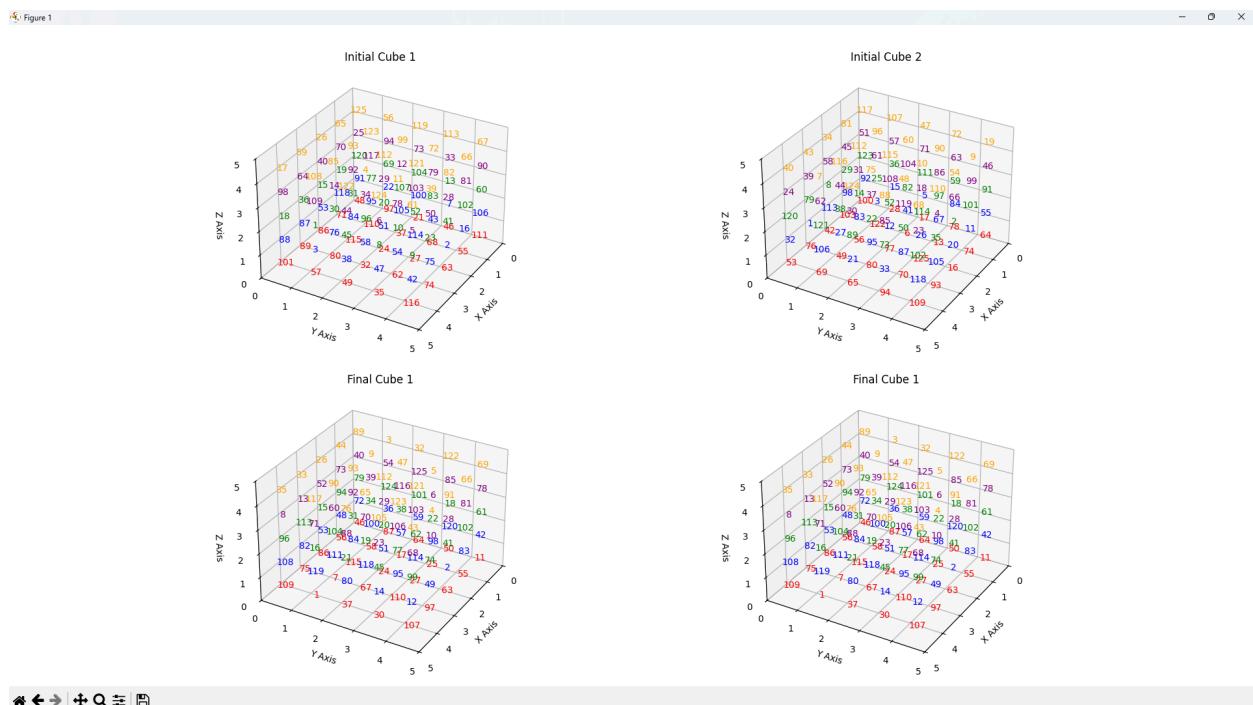
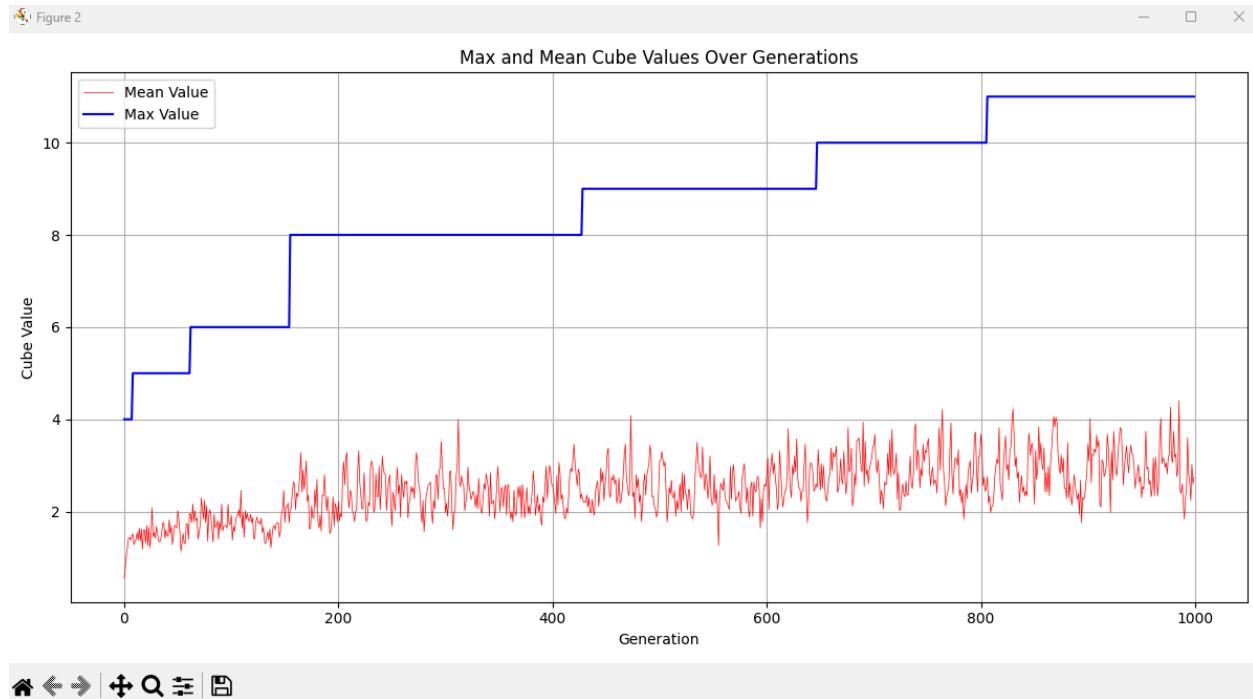
Tugas Besar 1 IF 3170 Intelegrasi Artificial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search



**RESULT**  
**Genetic Algorithm Duration:** 70.2689 seconds  
**Best value:** 9  
**Number of population:** 50  
**Number of iterations:** 1000

Pada percobaan ini digunakan populasi 50 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 9 dan durasi eksekusinya adalah 70.2689 detik

### iii. Percobaan 3

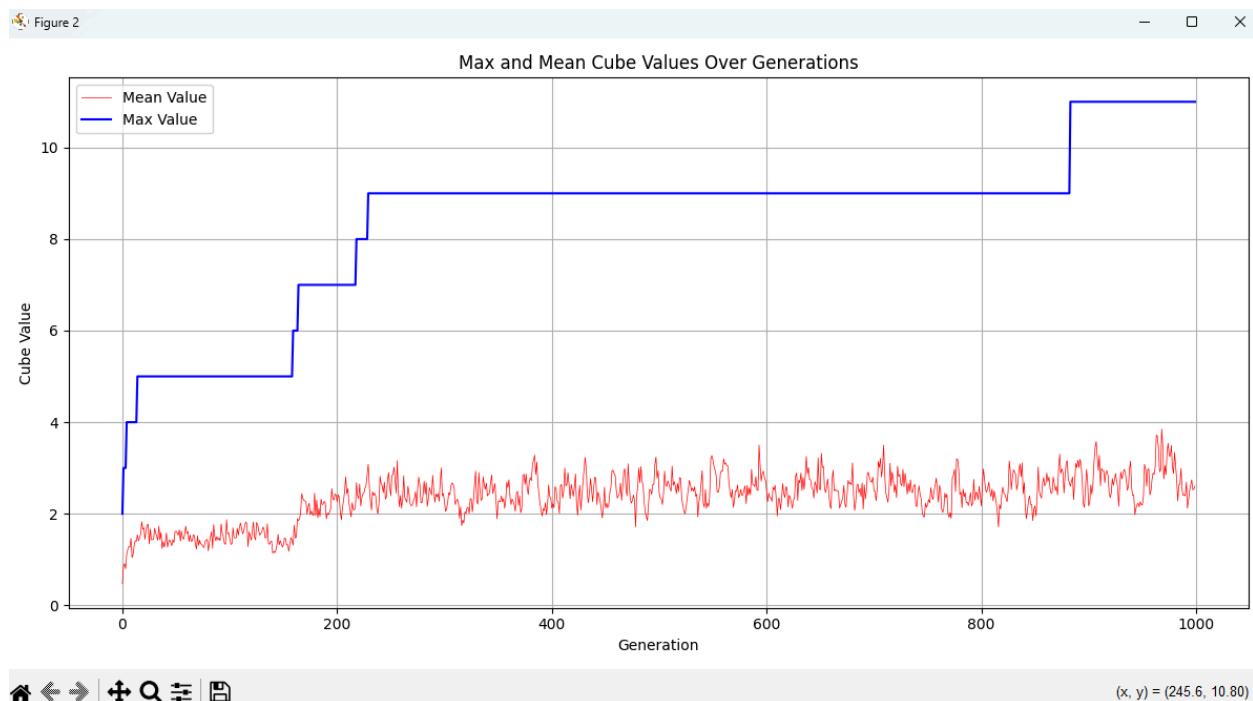


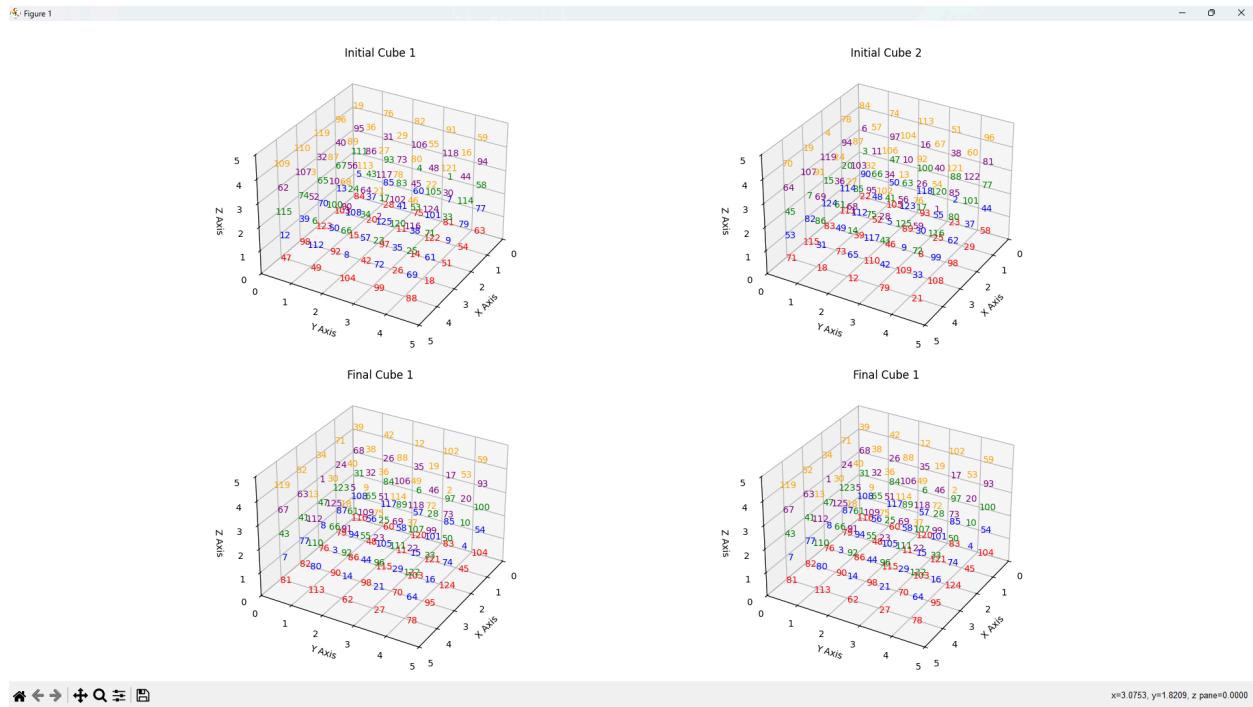
```
RESULT
Genetic Algorithm Duration: 70.9919 seconds
Best value: 11
Number of population: 50
Number of iterations: 1000
```

Pada percobaan ini digunakan populasi 50 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 11 dan durasi eksekusinya adalah 70.9919 detik

c. Populasi 100

i. Percobaan 1





```

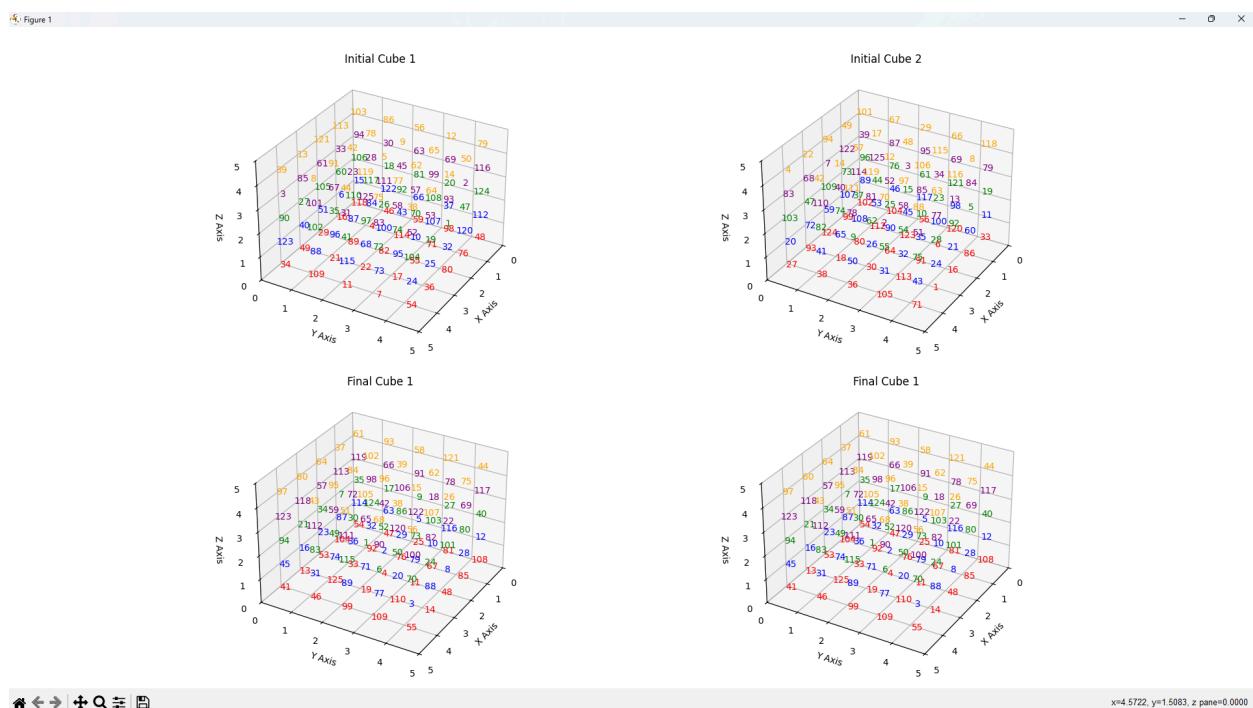
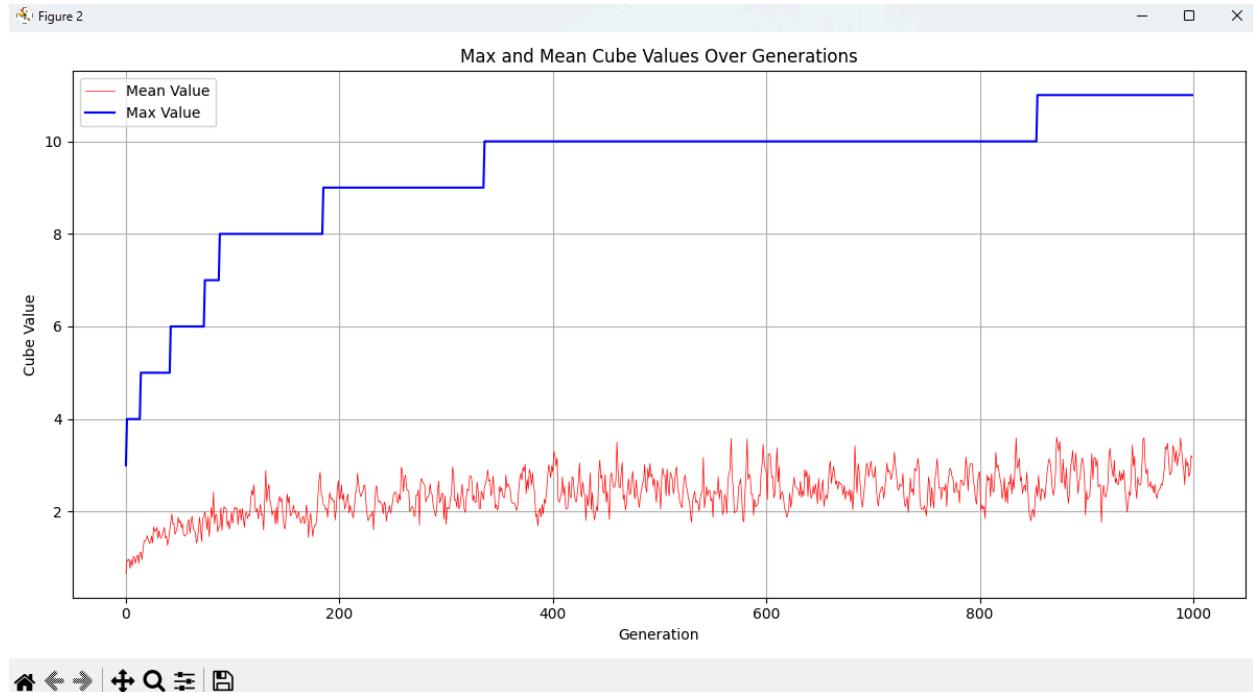
RESULT
Genetic Algorithm Duration: 144.4806 seconds
Best value: 11
Number of population: 100
Number of iterations: 1000

```

Pada percobaan ini digunakan populasi 100 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 11 dan durasi eksekusinya adalah 144.4806 detik

## ii. Percobaan 2

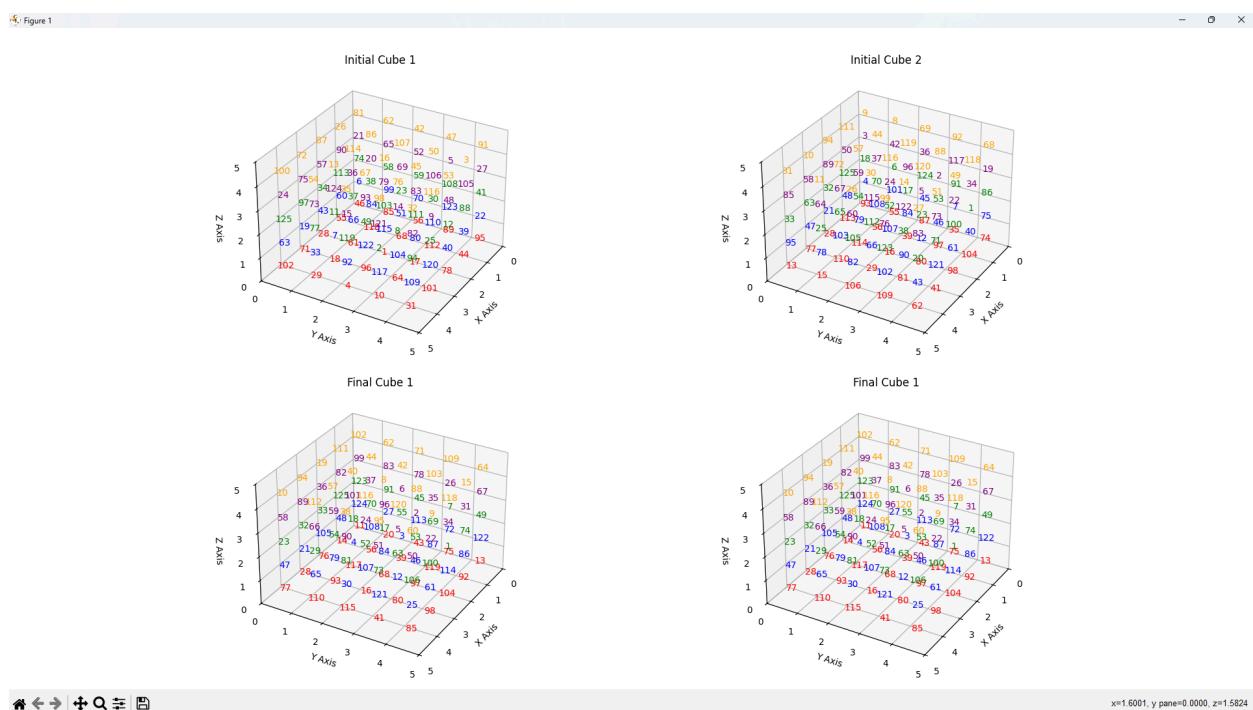
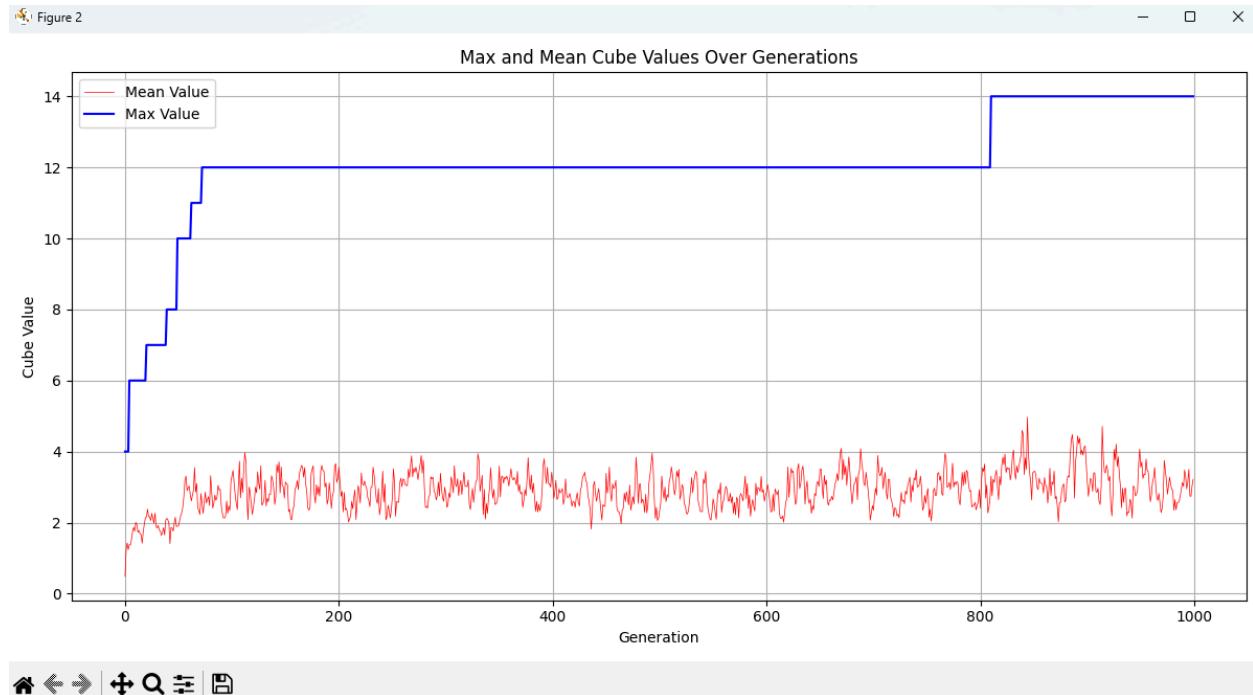
Tugas Besar 1 IF 3170 Intelegrasi Artifisial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search



**RESULT**  
**Genetic Algorithm Duration: 142.9879 seconds**  
**Best value: 11**  
**Number of population: 100**  
**Number of iterations: 1000**

Pada percobaan ini digunakan populasi 100 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 11 dan durasi eksekusinya adalah 142.9879 detik

### iii. Percobaan 3



```
RESULT
Genetic Algorithm Duration: 142.0463 seconds
Best value: 14
Number of population: 100
Number of iterations: 1000
```

Pada percobaan ini digunakan populasi 100 dengan jumlah iterasi 1000 dan didapatkan best objective function nya sebesar 14 dan durasi eksekusinya adalah 142.0463 detik

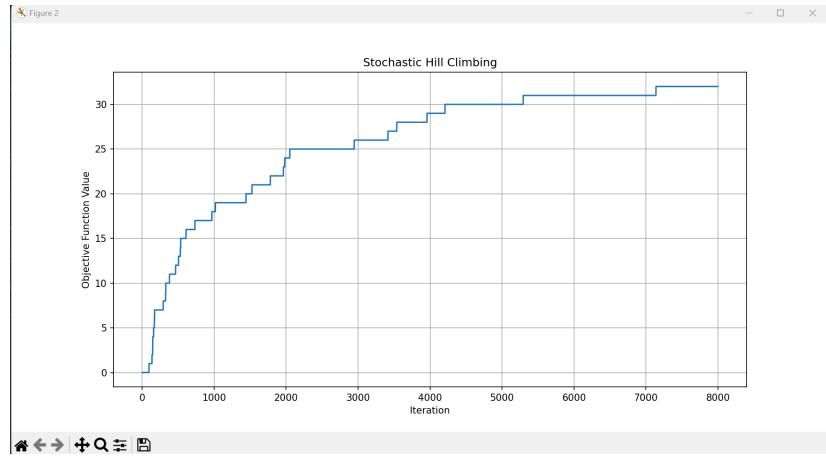
Dapat dilihat dari hasil eksperimen implementasi Genetic Algorithm pada permasalahan pencarian Magic Cube berukuran 5x5x5 didapatkan fakta bahwa semakin besar iterasi atau jumlah generasi yang dihasilkan maka semakin baik solusi yang diberikan oleh algoritma. Selain itu semakin besar populasi pada eksperimen maka semakin baik pula hasil yang diberikan oleh algoritma. Perlu dikehui pula bahwa algoritma ini memiliki kemungkinan terjebak pada suatu lokal optima apabila populasi memiliki variasi yang cenderung rendah. Untuk mengatasi hal ini tentunya dibutuhkan metode crossover dan dilakukannya mutasi agar algoritma ini dapat mengeksplorasi solusi lebih jauh.

Untuk konsistensi algoritma Genetic Algorithm ini cenderung fluktuatif namun lumayan dapat diprediksi yakni untuk nilai populasi yang semakin besar maka waktu yang dibutuhkan akan semakin besar namun hasil yang diberikan akan lebih baik. Begitu pula untuk nilai iterasi yang semakin besar juga akan membuat waktu eksekusi semakin lama namun akan memberikan hasil yang lebih baik.

### 2.3.2 Stochastic Algorithm

Stochastic algorithm diimplementasikan dengan mencari successor secara random lalu apabila successor tersebut memiliki value yang lebih besar dari current state, maka successor tersebut menjadi current state. Percobaan dilakukan sebanyak 4 kali. Percobaan 1-3 menggunakan 8000 iterasi dan percobaan ke 4 menggunakan 50.000 iterasi.

#### 1. Percobaan 1



```

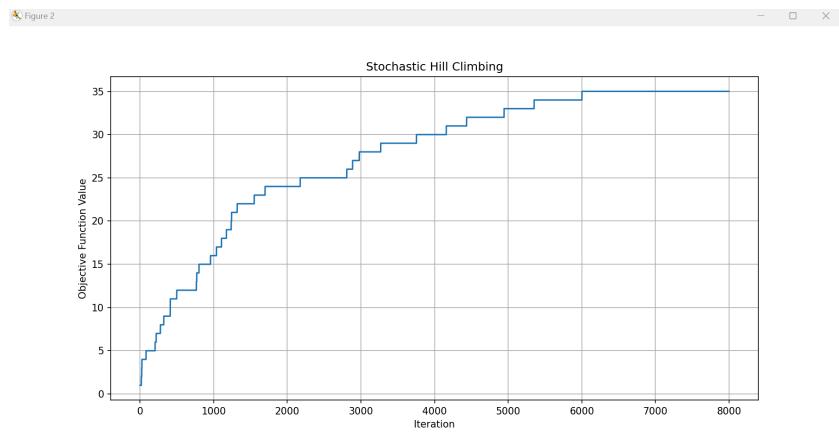
Steepest Ascent Algorithm Duration: 14.3466 seconds

Best value: 32
Number of iterations: 8000
Last State:
61 45 87 103 81 62 17 18 29 10 108 84 78 70 122 30 79 74 113 7 48 90 2 80 95
22 99 116 42 85 73 102 97 31 12 21 34 60 104 67 123 63 83 24 100 76 3 52 92 107
49 71 94 47 37 44 28 50 98 106 40 69 117 54 35 105 8 38 20 26 77 68 16 66 111
125 57 6 25 120 33 96 89 51 46 112 88 15 91 9 4 55 65 119 72 41 19 13 118 124
58 43 11 86 36 121 14 23 56 101 109 64 1 59 82 53 114 32 39 110 93 27 5 75 115

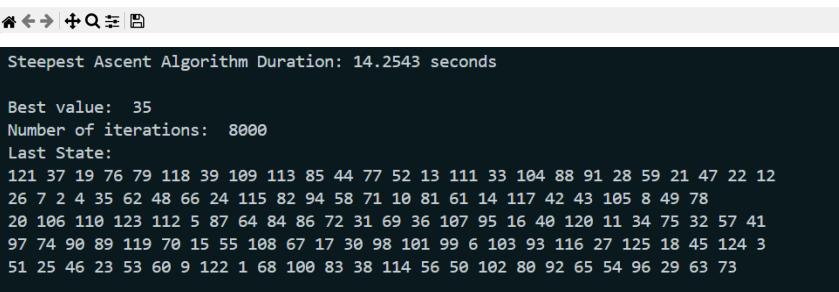
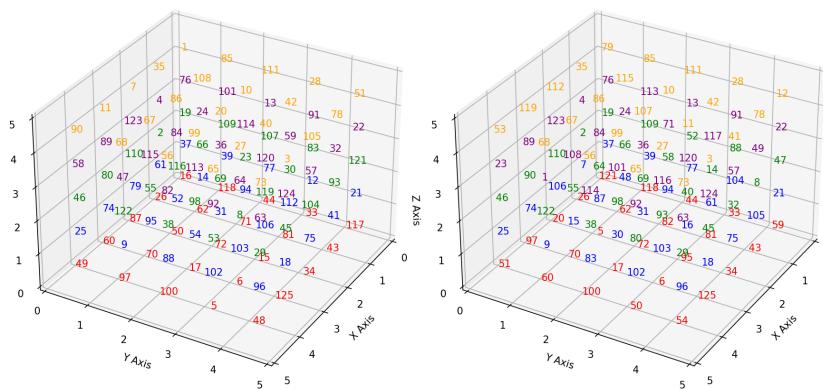
```

Didapatkan hasil bahwa durasinya 14.34 detik dengan value terbaik 32 dari 109

## 2. Percobaan 2



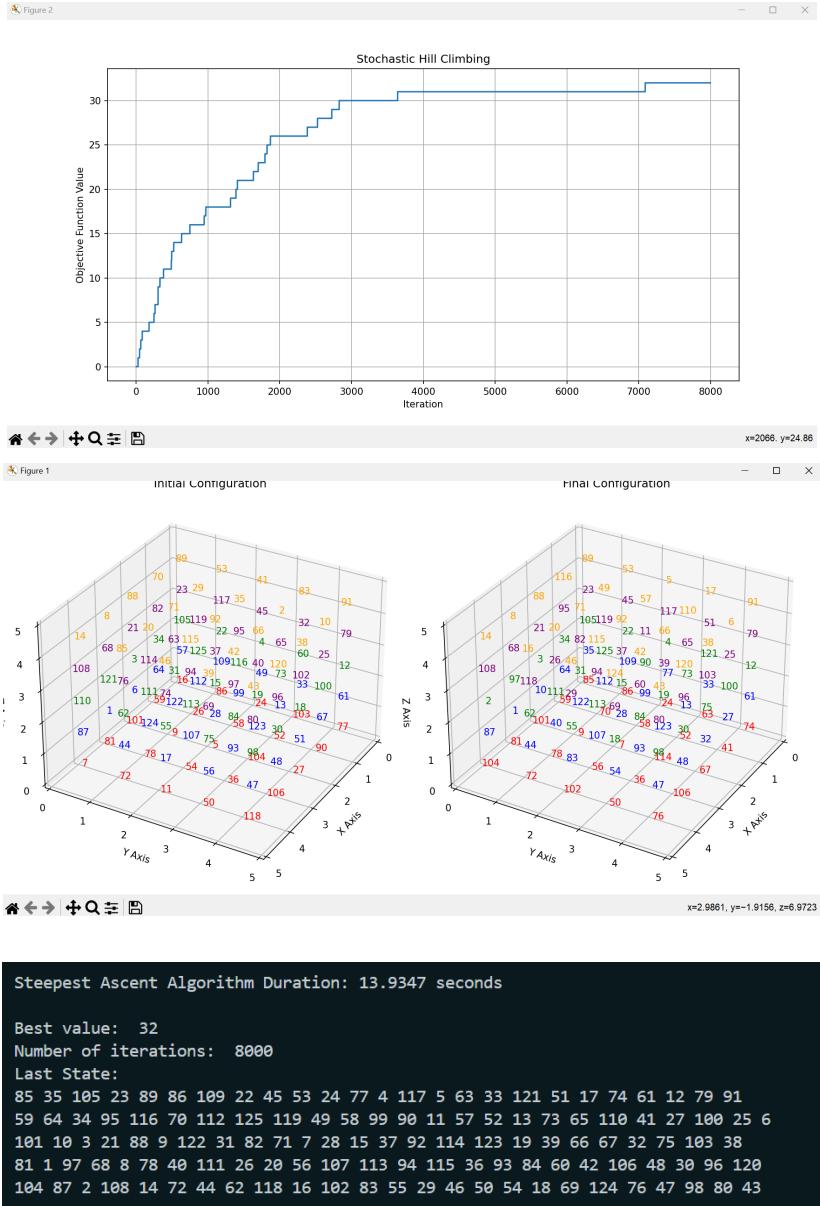
Initial Configuration      Final Configuration



Didapatkan hasil bahwa durasinya 14.25 detik dengan value terbaik 35 dari 109

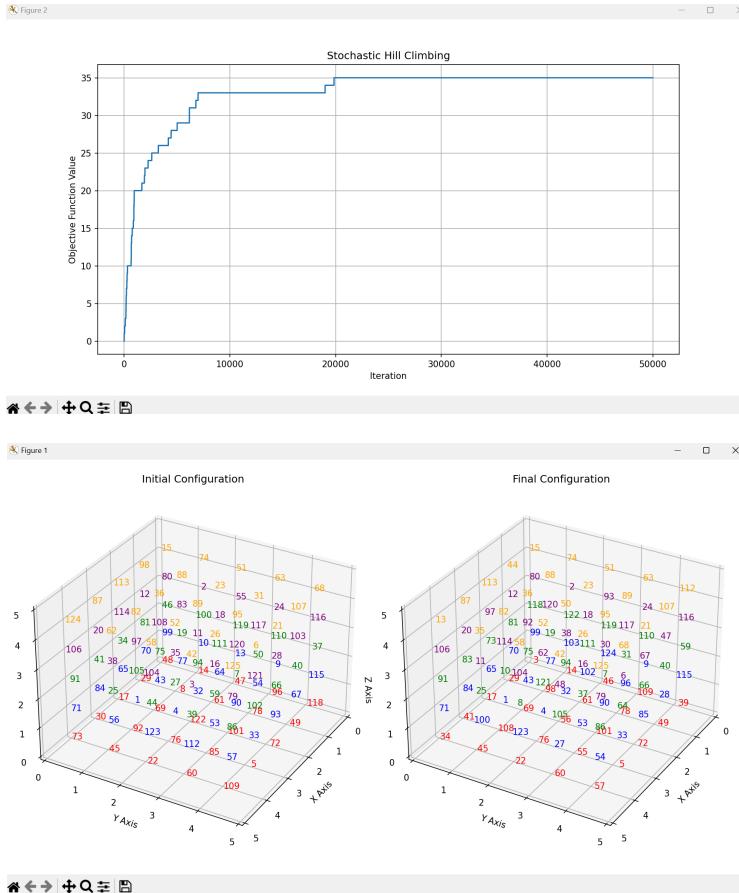
## 3. Percobaan 3

Tugas Besar 1 IF 3170 Intelelegensi Artifisial  
 Pencarian Solusi Diagonal Magic Cube dengan Local Search



Didapatkan hasil bahwa durasinya 13.93 detik dengan value terbaik 32 dari 109

#### 4. Percobaan 4



```

Steepest Ascent Algorithm Duration: 85.3862 seconds
Best value: 35
Number of iterations: 50000
Last State:
3 99 118 80 15 14 103 122 2 74 46 124 119 93 51 109 9 110 24 63 39 115 59 116 112
29 70 81 12 44 98 77 19 120 88 61 102 111 18 23 78 96 31 117 89 49 28 40 47 107
17 65 73 97 113 69 43 75 92 36 56 32 94 38 50 101 90 7 30 95 72 85 66 67 21
41 84 83 20 87 108 1 10 114 82 76 4 121 62 52 55 53 37 16 26 5 33 64 6 68
34 71 91 106 13 45 100 25 11 35 22 123 8 104 58 60 27 105 48 42 57 54 86 79 125

```

Didapatkan hasil bahwa durasinya 85.38 detik dengan value terbaik 35 dari 109

## Analisis

- Algoritma ini mencapai best valuenya sekitar 30 - 35. Ini masih jauh dari optimum global yang membutuhkan value 109. Yang berarti bahwa algoritma ini masih sering terjebak di optimum local pada value 30-35. Ini terjadi karena algoritma stochastic sangat bergantung terhadap probabilitas akibat pemilihan successornya yang secara random.
- Durasi yang dibutuhkan untuk menyelesaikan algoritma ini sesuai dengan banyaknya total iterasi yang diperlukan. Semakin banyak total iterasi maka akan semakin lama pula

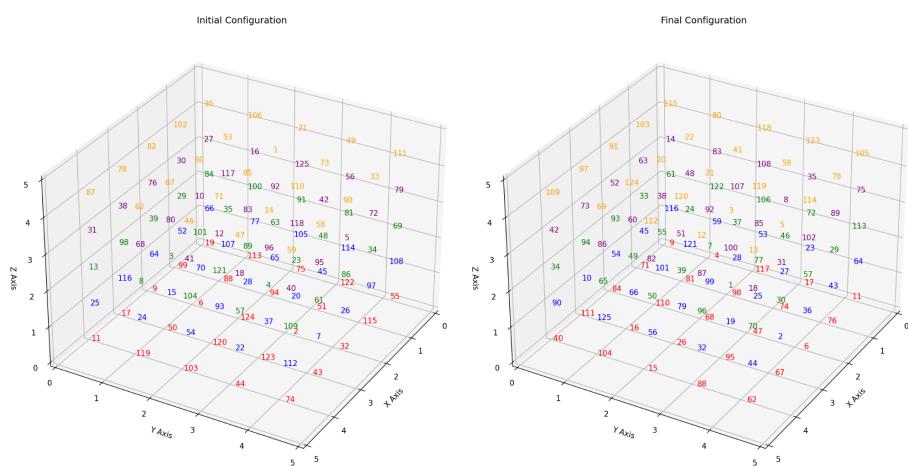
waktu yang dibutuhkan. Namun banyaknya iterasi ini tetap tidak akan mempengaruhi best valuenya karena mentok hanya ada di 30an (terjebak di optimum local). Sehingga algoritma ini paling baik dilakukan dengan maks iterasi 10000 kali karena setelah iterasi tersebut, best value cenderung datar dan tidak bertambah lagi.

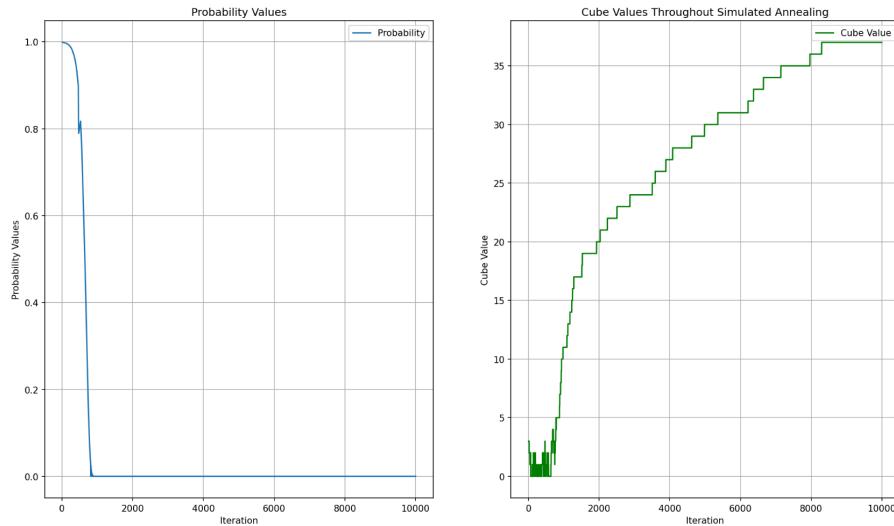
- Algoritma ini cukup konsisten dengan menghasilkan objective function value maksimal sebesar 30-35.

### 2.3.3 Simulated Annealing Algorithm

Berikut adalah hasil dari implementasi algoritma Simulated Annealing, sebagai berikut

#### 1. Percobaan 1





```
Running Simulated Annealing Algorithm

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

Steepest Ascent Algorithm Duration: 16.7155 seconds

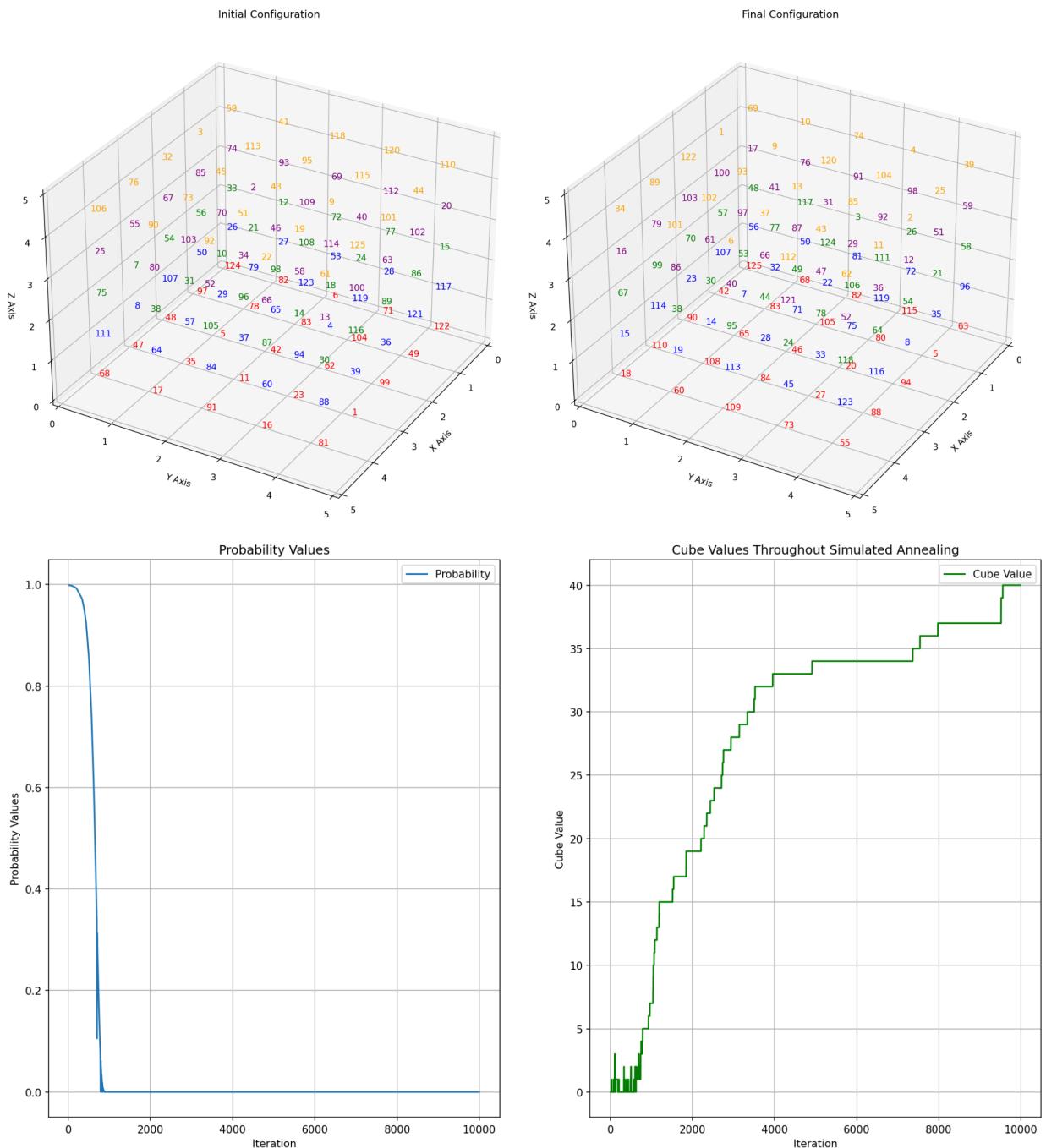
Last value: 37
Number of stuck: 8223

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

Pada percobaan 1, diperoleh nilai objektif tertinggi adalah 37 dengan durasi pencarian selama 16,7155 detik. Pada proses pencarian, algoritma mengalami stuck pada lokal optima sebanyak 8223 kali.

## 2. Percobaan 2

Tugas Besar 1 IF 3170 Intelelegensi Artifisial  
 Pencarian Solusi Diagonal Magic Cube dengan Local Search



```
Running Simulated Annealing Algorithm

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

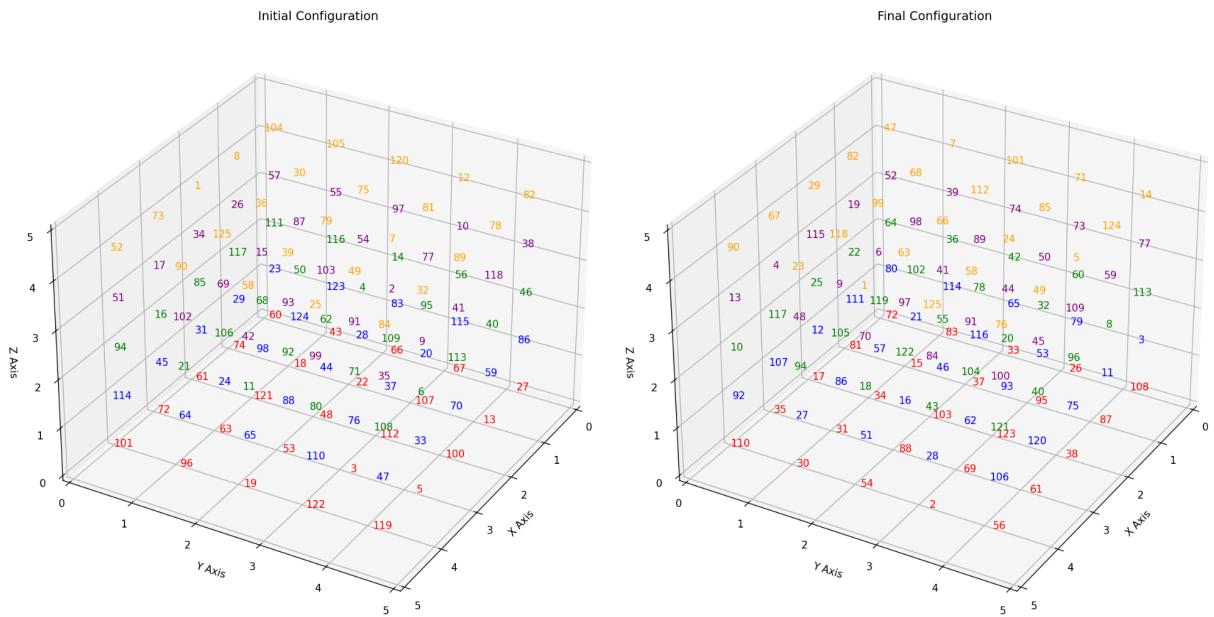
Steepest Ascent Algorithm Duration: 17.6988 seconds

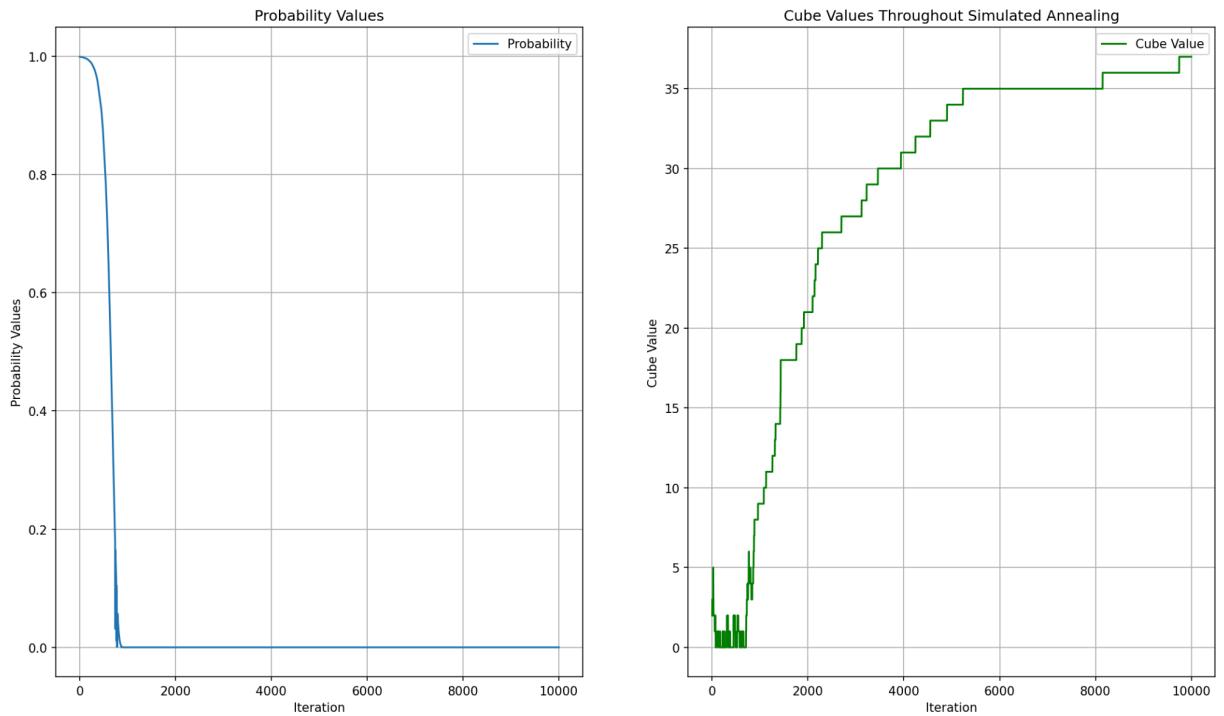
Last value: 40
Number of stuck: 8367

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

Pada percobaan 2, diperoleh nilai objektif tertinggi adalah 40 dengan durasi pencarian selama 17,6988 detik. Pada proses pencarian, algoritma mengalami stuck pada lokal optima sebanyak 8367 kali.

### 3. Percobaan 3





```
Running Simulated Annealing Algorithm

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

Steepest Ascent Algorithm Duration: 17.2217 seconds

Last value: 37
Number of stuck: 8239

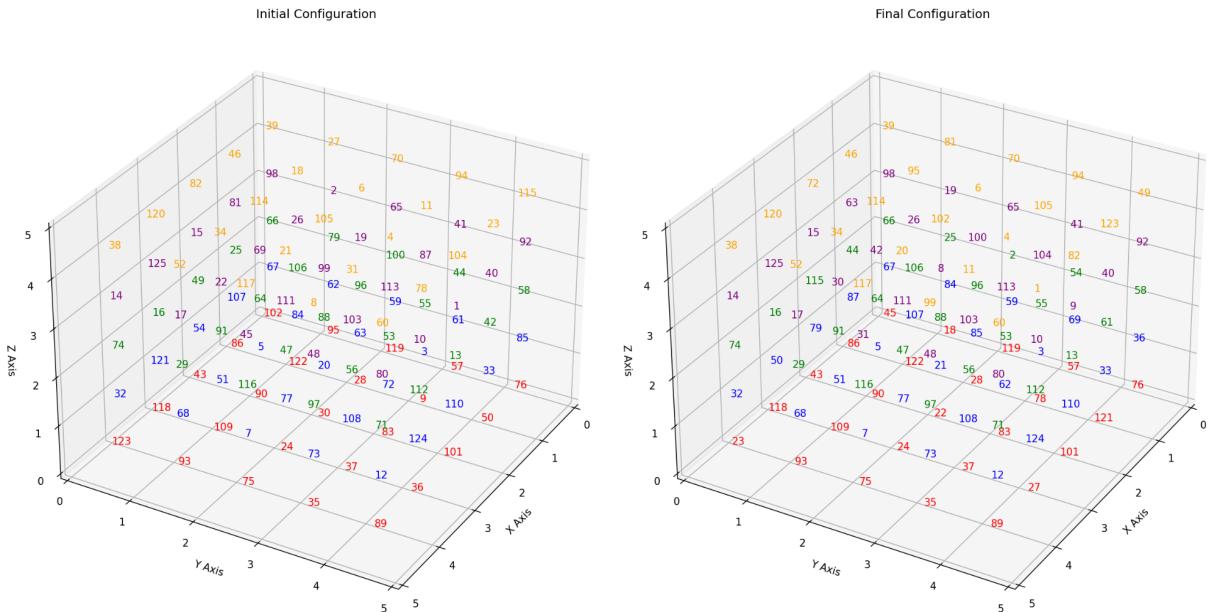
Last State:
72 80 64 52 47 83 114 36 39 7 33 65 42 74 101 26 79 60 73 71 108 3 113 77 14
81 111 22 19 82 15 21 102 98 68 37 116 78 89 112 95 53 32 50 85 87 11 8 59 124
17 12 25 115 29 34 57 119 6 99 103 46 55 41 66 123 93 20 44 24 38 75 96 109 5
35 107 117 4 67 31 86 105 9 118 88 16 122 97 63 69 62 104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

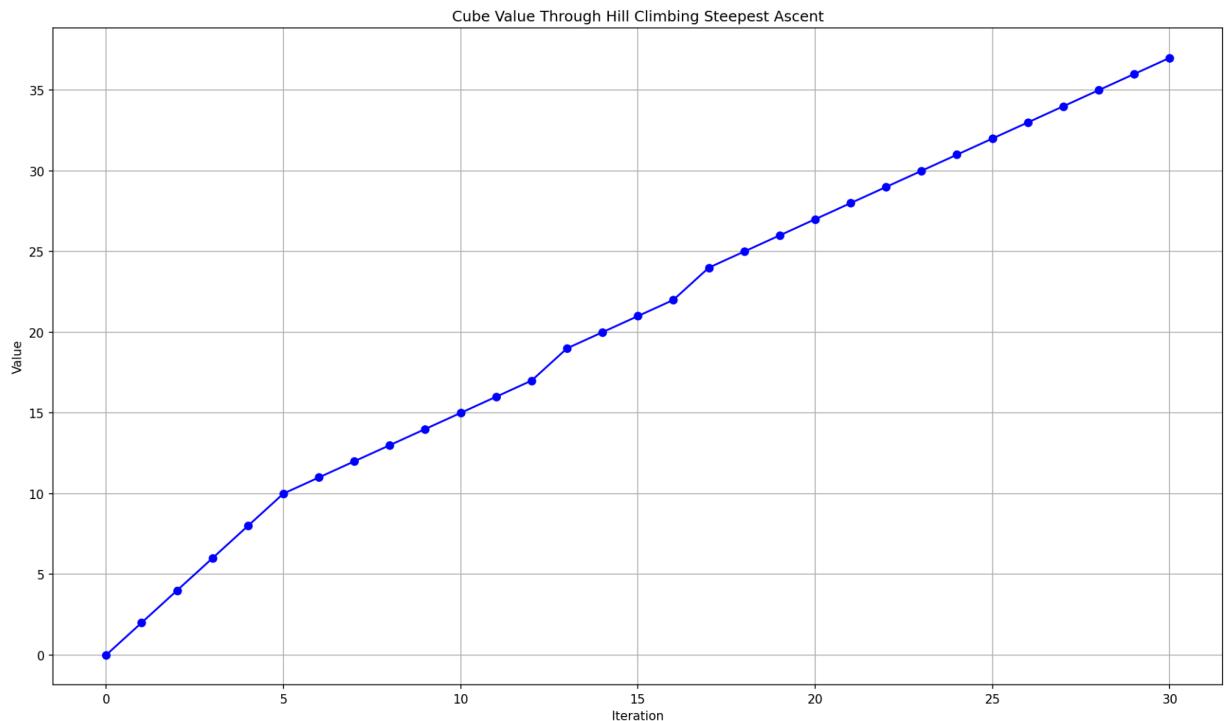
Pada percobaan 3, diperoleh nilai objektif tertinggi adalah 37 dengan durasi pencarian selama 17,2217 detik. Pada proses pencarian, algoritma mengalami stuck pada lokal optima sebanyak 8239 kali.

Hasil dari percobaan Simulated Annealing Algorithm menunjukkan bahwa algoritma ini masih jauh dari mencapai global optima yang diinginkan, yaitu 109. Dalam tiga percobaan yang dilakukan, nilai objektif tertinggi yang diperoleh hanya mencapai 40, sementara dua percobaan lainnya menghasilkan nilai yang sama, yaitu 37. Hal ini menunjukkan adanya jarak yang signifikan antara hasil yang dicapai dan global optima, dengan selisih yang mencapai 69 hingga 72. Frekuensi stuck pada lokal optima yang tinggi, yaitu antara 8223 hingga 8367 kali, mengindikasikan bahwa algoritma ini tidak cukup efektif dalam menjelajahi ruang pencarian dan cenderung terjebak dalam lokal optima. Durasi pencarian yang ada cukup stabil, berkisar antara 16.7155 hingga 17.6988 detik. Algoritma lebih baik dari stochastic karena memungkinkan untuk turun sebentar (mengambil nighbor yang nilainya lebih rendah) untuk mencapai value yang lebih tinggi. Meskipun algoritma ini lebih baik dari Stochastic Algorithm, algoritma ini masih diperlukan optimasi lebih lanjut mengenai pengaturan suhu dan metode cooling schedule yang digunakan untuk memperoleh hasil yang lebih baik.

### 2.3.4 Steepest Ascent Algorithm

#### 1. Percobaan 1





```
Running Steepest Ascent Algorithm

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

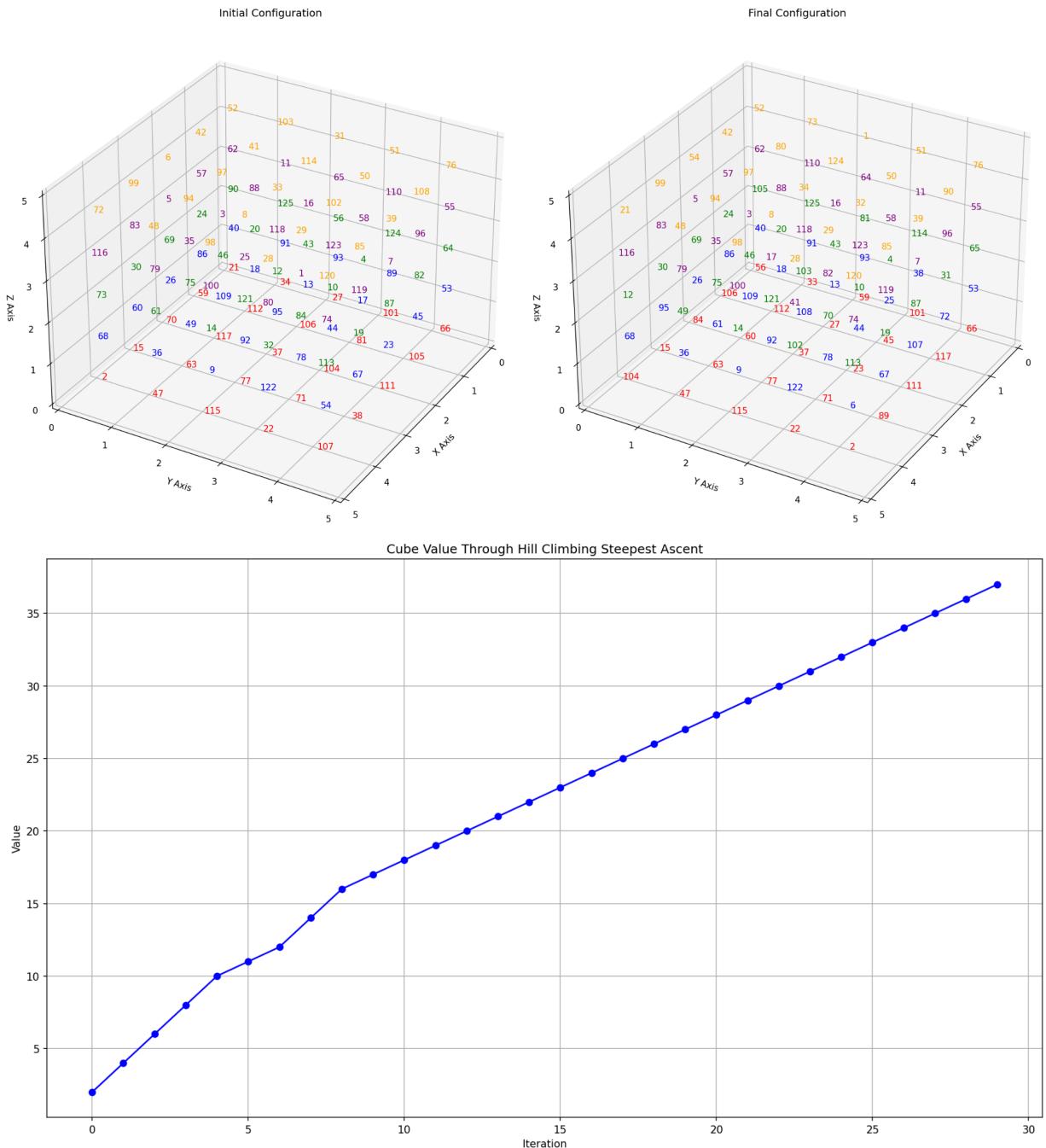
Steepest Ascent Algorithm Duration: 131.2110 seconds

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

Pada percobaan 1, diperoleh nilai objektif tertinggi adalah 37 dengan durasi pencarian selama 131,2110 detik. Pada proses pencarian, algoritma menjalankan iterasi sebanyak 31.

## 2. Percobaan 2

Tugas Besar 1 IF 3170 Intelegrasi Artifisial  
 Pencarian Solusi Diagonal Magic Cube dengan Local Search



```
Running Steepest Ascent Algorithm

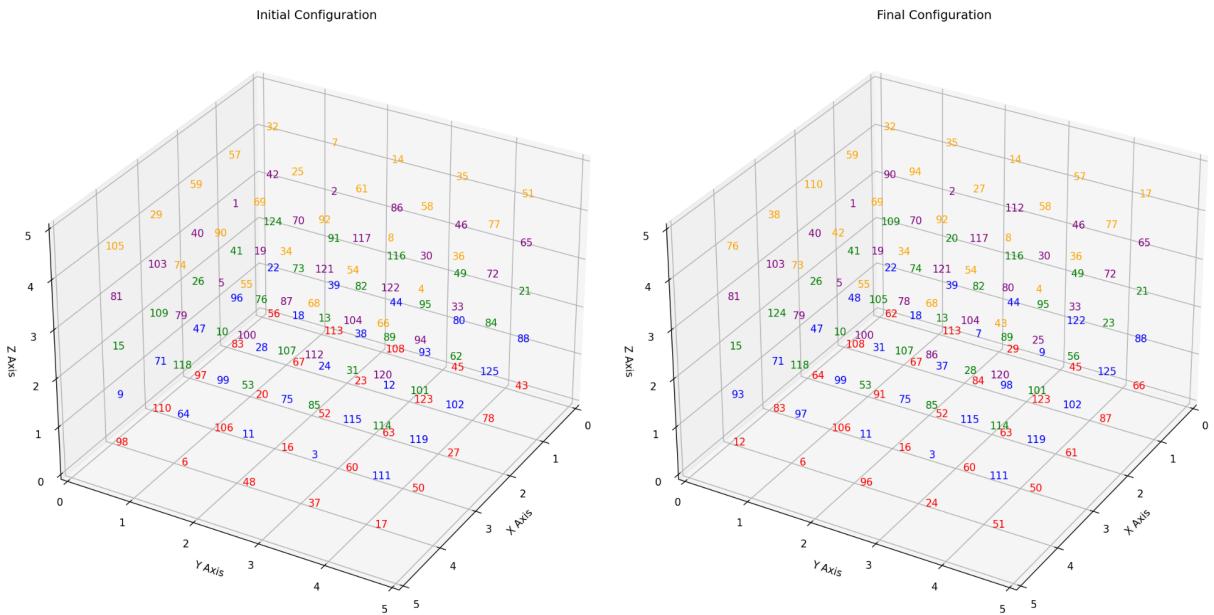
Initial State:
21 40 90 62 52 34 91 125 11 103 27 93 56 65 31 101 89 124 110 51 66 53 64 55 76
59 86 24 57 42 112 18 20 88 41 106 13 43 16 114 81 17 4 58 50 105 45 82 96 108
70 26 69 5 6 117 109 46 3 97 37 95 12 118 33 104 44 10 123 102 111 23 87 7 39
15 60 30 83 99 63 49 75 35 94 77 92 121 25 8 71 78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

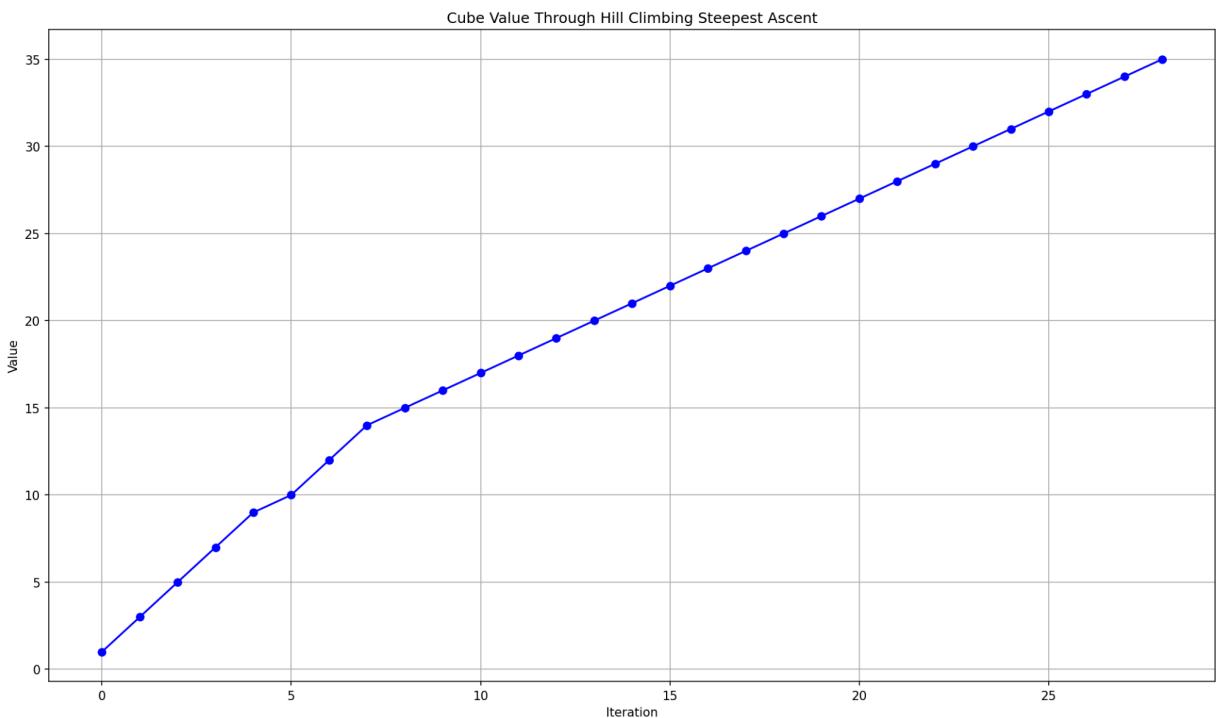
Steepest Ascent Algorithm Duration: 135.9449 seconds

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

Pada percobaan 2, diperoleh nilai objektif tertinggi adalah 37 dengan durasi pencarian selama 135,9449 detik. Pada proses pencarian, algoritma menjalankan iterasi sebanyak 30.

### 3. Percobaan 3





```
Running Steepest Ascent Algorithm

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

Steepest Ascent Algorithm Duration: 122.5843 seconds

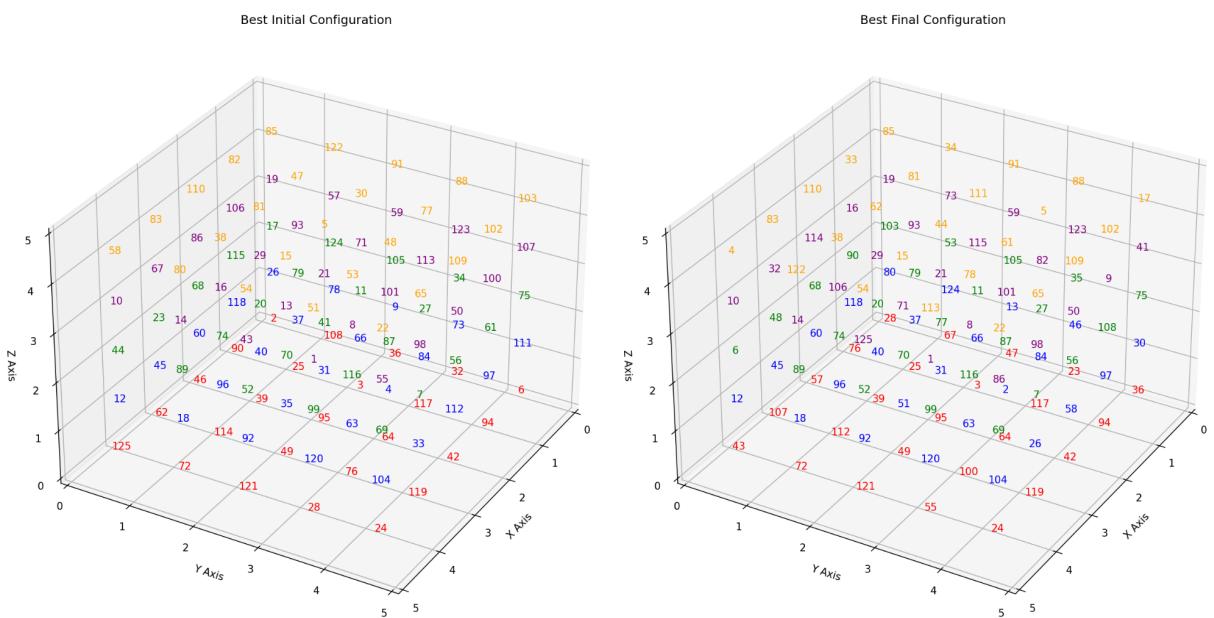
Best value: 35
Number of iterations: 29
Last State:
62 22 109 90 32 113 39 20 2 35 29 44 116 112 14 45 122 49 46 57 66 88 21 65 17
108 48 41 1 59 67 18 74 70 94 84 7 82 117 27 123 9 95 30 58 87 125 23 72 77
64 47 26 40 110 91 31 105 19 69 52 37 13 121 92 63 98 89 80 8 61 102 56 33 36
83 71 124 103 38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

Pada percobaan 3, diperoleh nilai objektif tertinggi adalah 35 dengan durasi pencarian selama 122,5843 detik. Pada proses pencarian, algoritma menjalankan iterasi sebanyak 29.

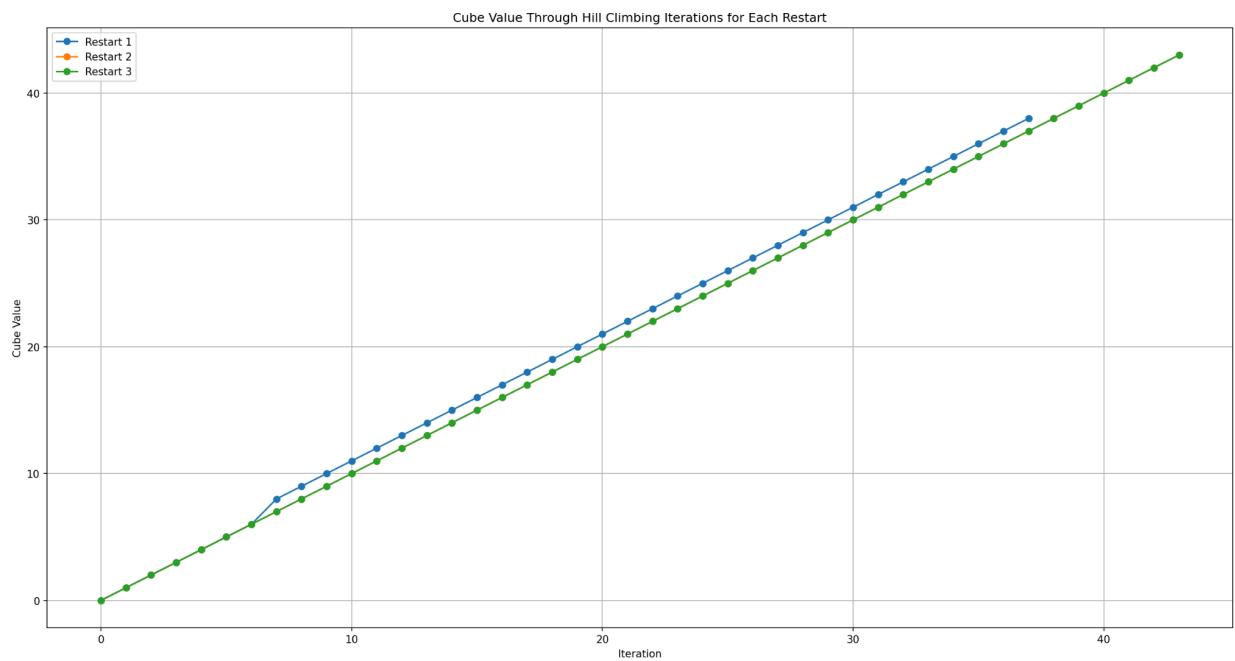
Hasil dari percobaan Steepest Ascent Algorithm menunjukkan hasil yang cukup konsisten, meskipun tidak mencapai global optima yang diinginkan, yaitu 109. Selama tiga percobaan, algoritma ini berhasil mendapat nilai objektif tertinggi yang mencapai 37 dalam dua dari tiga percobaan, sementara pada percobaan ketiga, nilai tertinggi yang didapat adalah 35. Ini menunjukkan bahwa meskipun algoritma ini bisa memberikan hasil yang konsisten, meskipun masih ada jarak yang signifikan antara hasil yang diperoleh dan goal optimal, dengan selisih mencapai 72 hingga 74. Dalam hal durasi pencarian, waktu yang dihabiskan cukup panjang, berkisar antara 122,5843 hingga 135,9449 detik, dengan iterasi yang dijalankan sebanyak 29 hingga 31 kali. Meskipun durasi pencarian terlihat stabil, lambatnya proses ini menunjukkan bahwa setiap iterasi yang dilakukan harus mencari solusi ruang pencarian yang cukup besar.

### 2.3.5 Random Restart Hill Climbing Algorithm

#### 1. Percobaan 1



Tugas Besar 1 IF 3170 Intelelegensi Artifisial  
Pencarian Solusi Diagonal Magic Cube dengan Local Search



```
Running Random Restart Hill Climbing Algorithm

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

Random Restart Hill Climbing iteration 1

Hill Climbing with 38 iterations and best value 38

Random Restart Hill Climbing iteration 2

Hill Climbing with 44 iterations and best value 43

Random Restart Hill Climbing iteration 3

Hill Climbing with 44 iterations and best value 43

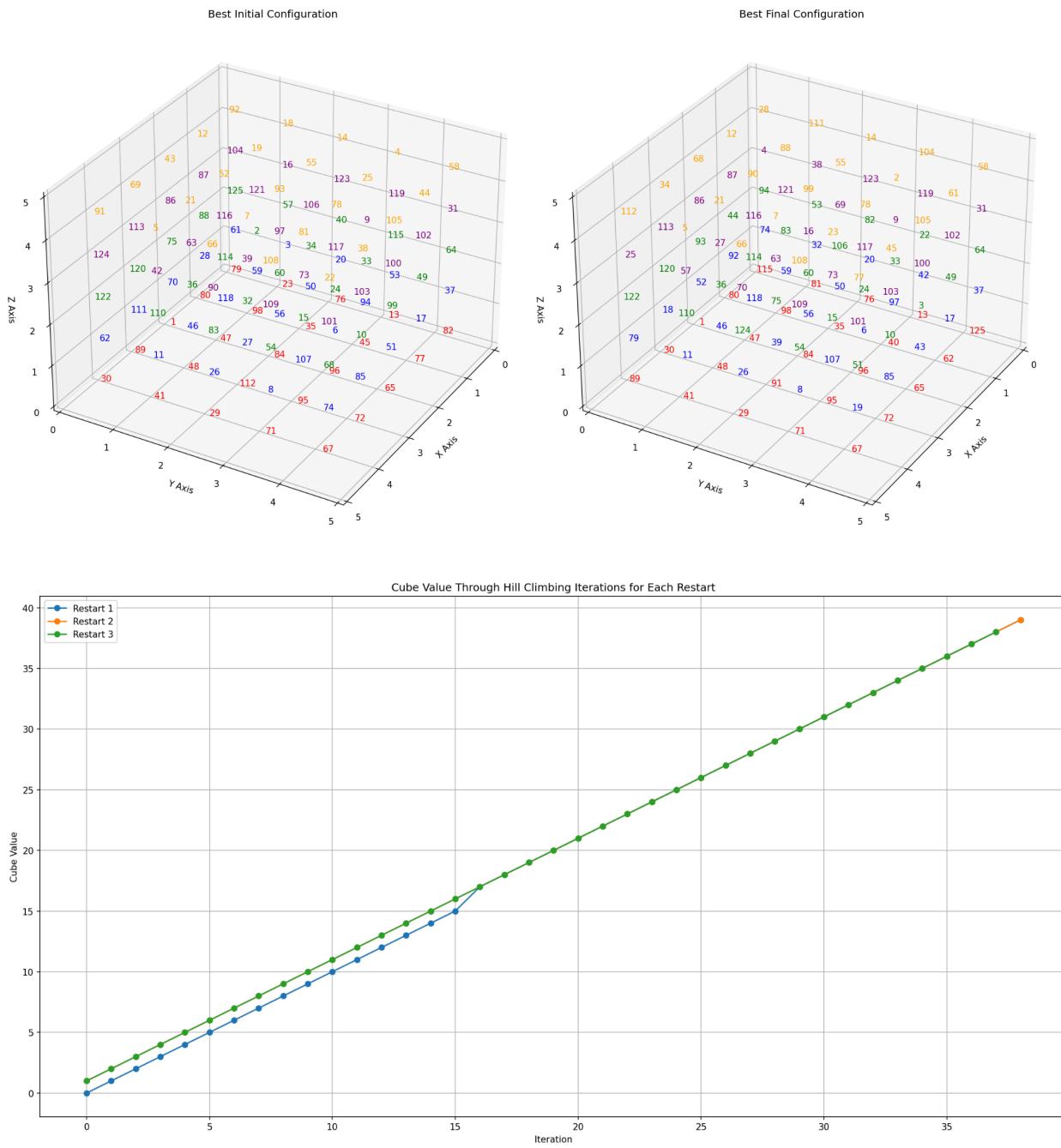
Steepest Ascent Algorithm Duration: 189.9062 seconds

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

Pada percobaan 1, dengan melakukan restart sebanyak 3 kali, diperoleh nilai objektif tertinggi adalah 43, dengan masing-masingnya adalah 38, 43, dan 43. Ketiga pencarian ini total memerlukan durasi 189,9062 detik. Pada proses pencarian, algoritma menjalankan iterasi sebanyak masing-masing 38, 44, dan 44 iterasi.

## 2. Percobaan 2

Tugas Besar 1 IF 3170 Intelegrasi Artifisial  
 Pencarian Solusi Diagonal Magic Cube dengan Local Search



```
Running Random Restart Hill Climbing Algorithm

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

Random Restart Hill Climbing iteration 1

Hill Climbing with 39 iterations and best value 39

Random Restart Hill Climbing iteration 2

Hill Climbing with 39 iterations and best value 39

Random Restart Hill Climbing iteration 3

Hill Climbing with 38 iterations and best value 38

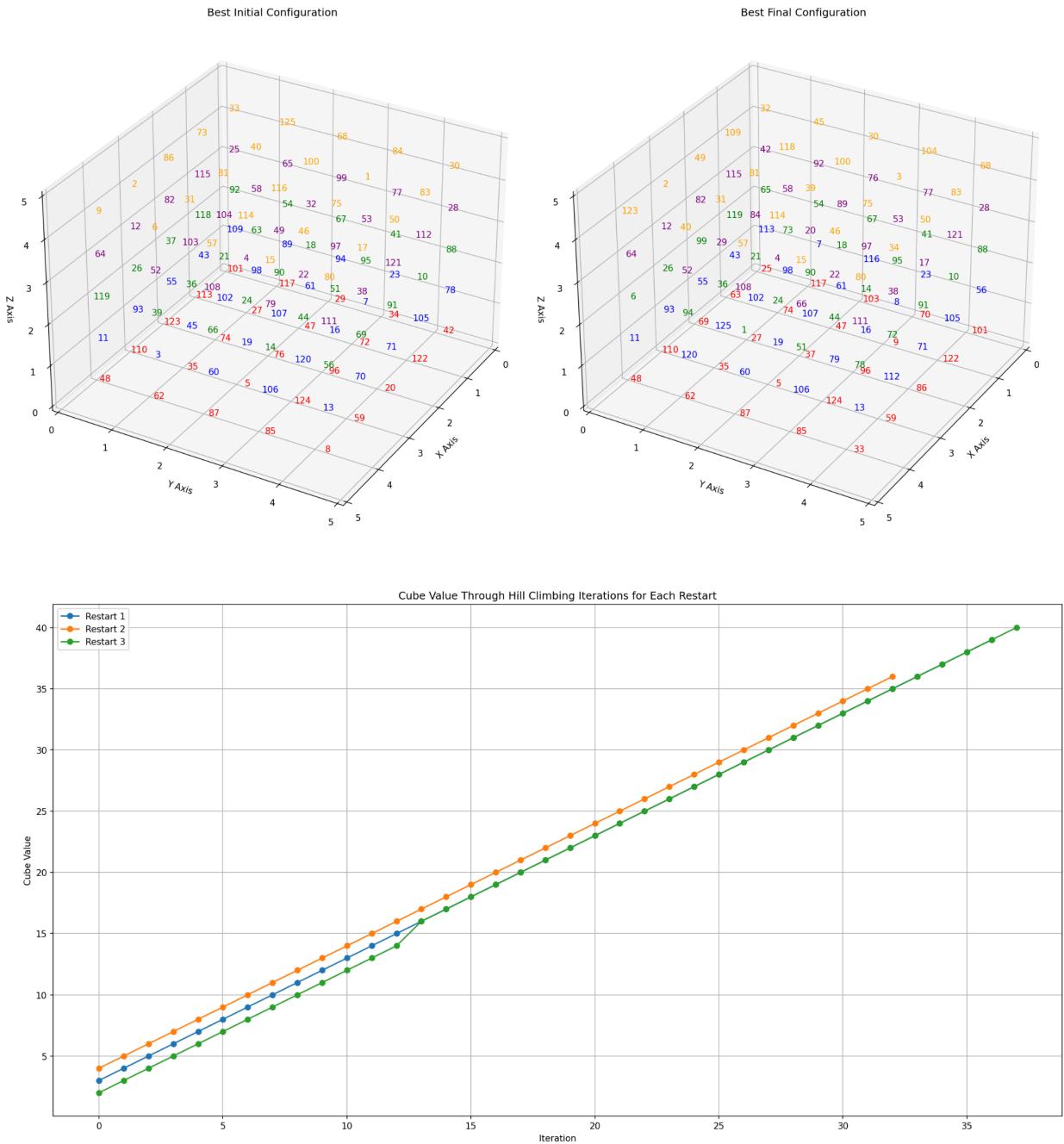
Steepest Ascent Algorithm Duration: 160.8866 seconds

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

Pada percobaan 1, dengan melakukan restart sebanyak 3 kali, diperoleh nilai objektif tertinggi adalah 43, dengan masing-masingnya adalah 38, 43, dan 43. Ketiga pencarian ini total memerlukan durasi 189,9062 detik. Pada proses pencarian, algoritma menjalankan iterasi sebanyak masing-masing 38, 44, dan 44 iterasi.

### 3. Percobaan 3

Tugas Besar 1 IF 3170 Intelegrasi Artifisial  
 Pencarian Solusi Diagonal Magic Cube dengan Local Search



```
Running Random Restart Hill Climbing Algorithm

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

Random Restart Hill Climbing iteration 1

Hill Climbing with 37 iterations and best value 39

Random Restart Hill Climbing iteration 2

Hill Climbing with 33 iterations and best value 36

Random Restart Hill Climbing iteration 3

Hill Climbing with 38 iterations and best value 40

Steepest Ascent Algorithm Duration: 157.7692 seconds

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

Pada percobaan 3, dengan melakukan restart sebanyak 3 kali, diperoleh nilai objektif tertinggi adalah 39, dengan masing-masingnya adalah 39, 36, dan 40. Ketiga pencarian ini total memerlukan durasi 157,7692 detik. Pada proses pencarian, algoritma menjalankan iterasi sebanyak masing-masing 37, 33, dan 38 iterasi.

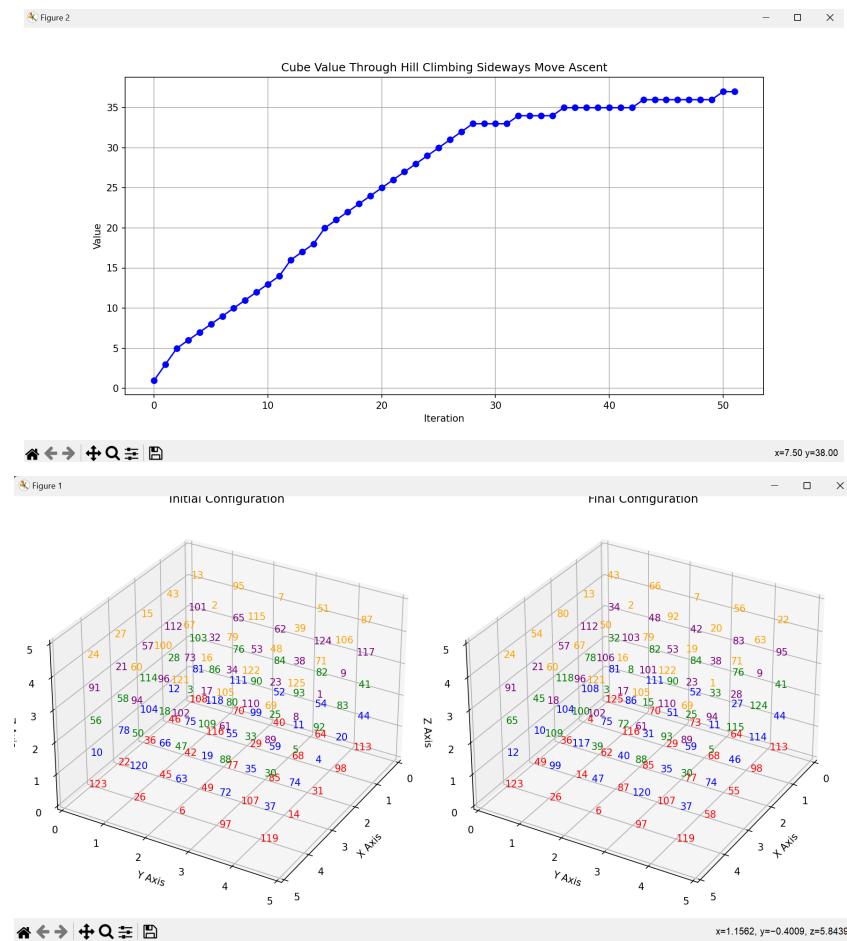
Hasil dari percobaan Random Restart Algorithm menunjukkan pada percobaan pertama, algoritma berhasil mencapai nilai objektif tertinggi sebesar 43 dengan melakukan restart sebanyak tiga kali, dengan hasil setiap pencarian masing-masing adalah 38, 43, dan 43. Total waktu yang diperlukan untuk ketiga pencarian ini adalah 189,9062 detik, dengan iterasi yang dijalankan masing-masing sebanyak 38, 44, dan 44 iterasi. Hasil ini menunjukkan kemampuan algoritma untuk mengeksplorasi ruang pencarian yang lebih luas berkat metode restart, menghasilkan nilai yang lebih baik dari sebelumnya dan mencapai 43 yang mendekati target optimal. Namun, pada percobaan ketiga, performa algoritma

menunjukkan capaian yang sedikit lebih rendah, dengan nilai objektif tertinggi sebesar 39, yang diperoleh dari hasil pencarian 39, 36, dan 40. Total durasi untuk ketiga pencarian ini lebih efisien, menghabiskan waktu 157,7692 detik dan menjalankan iterasi sebanyak 37, 33, dan 38 kali. Meskipun hasilnya tidak setinggi percobaan pertama, pengurangan waktu pencarian menunjukkan bahwa metode random restart dapat memberikan hasil yang bervariasi namun masih dalam kisaran yang wajar. Konsistensi hasil pada percobaan pertama dan ketiga menunjukkan bahwa Random Restart Algorithm dapat memberikan solusi yang lebih baik melalui eksplorasi yang lebih beragam dengan hasil yang cenderung stabil.

### 2.3.6 Hill Climbing Sideways Move Algorithm

Algoritma ini mirip dengan steepest Ascent, namun memungkinkan untuk memilih successor dengan nilai yang sama atau memungkinkan untuk bergerak secara mendatar (dilihat dari plot objective function vauenya). Ada tambahan limitasi juga yaitu maksimal sideways move, jika algoritma sudah melakukan sideways move atau memilih successor yang nilainya sama dengan current, maka program akan stop.

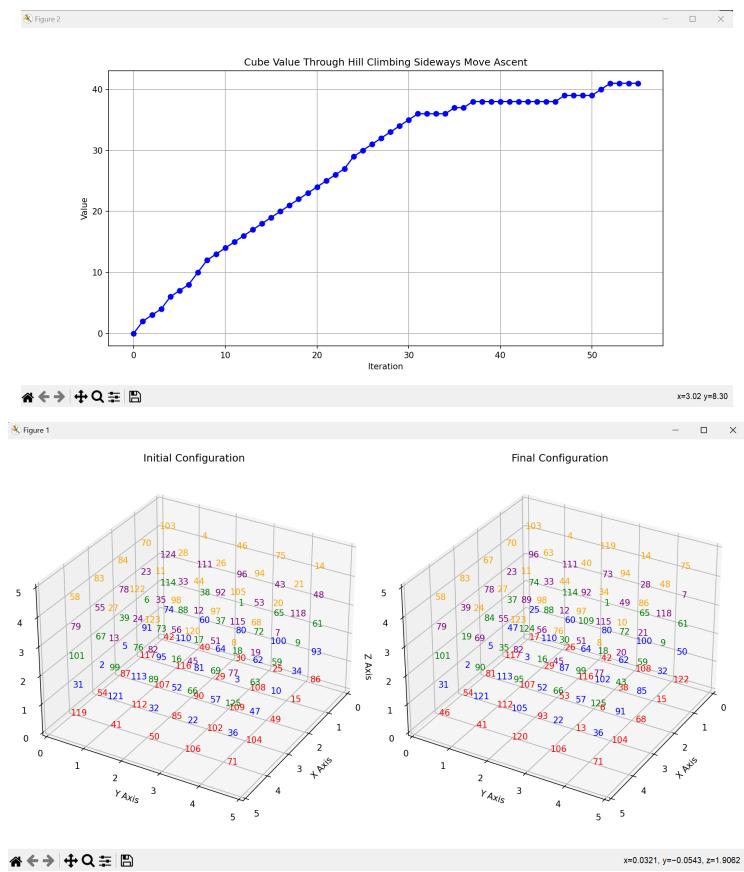
#### 1. Percobaan 1



```
Hill Climbing Sideways Algorithm Duration: 220.6256 seconds
Best value: 37
Number of iterations: 52
Last State:
125 81 32 34 43 70 111 82 48 66 73 52 84 42 7 64 27 76 83 56 113 44 41 95 22
4 108 78 112 13 116 86 8 103 2 29 51 90 53 92 68 11 33 38 20 98 114 124 9 63
36 104 118 57 80 62 75 3 106 50 85 31 15 101 79 77 59 25 23 19 55 46 115 28 71
49 10 45 21 54 14 117 100 96 67 87 40 72 17 16 107 35 93 110 122 58 74 5 94 1
123 12 65 91 24 26 99 109 18 60 6 47 39 102 121 97 120 88 61 105 119 37 30 89 69
```

Diperlukan waktu 220 detik dan menghasilkan best value 37 dengan iterasinya 52

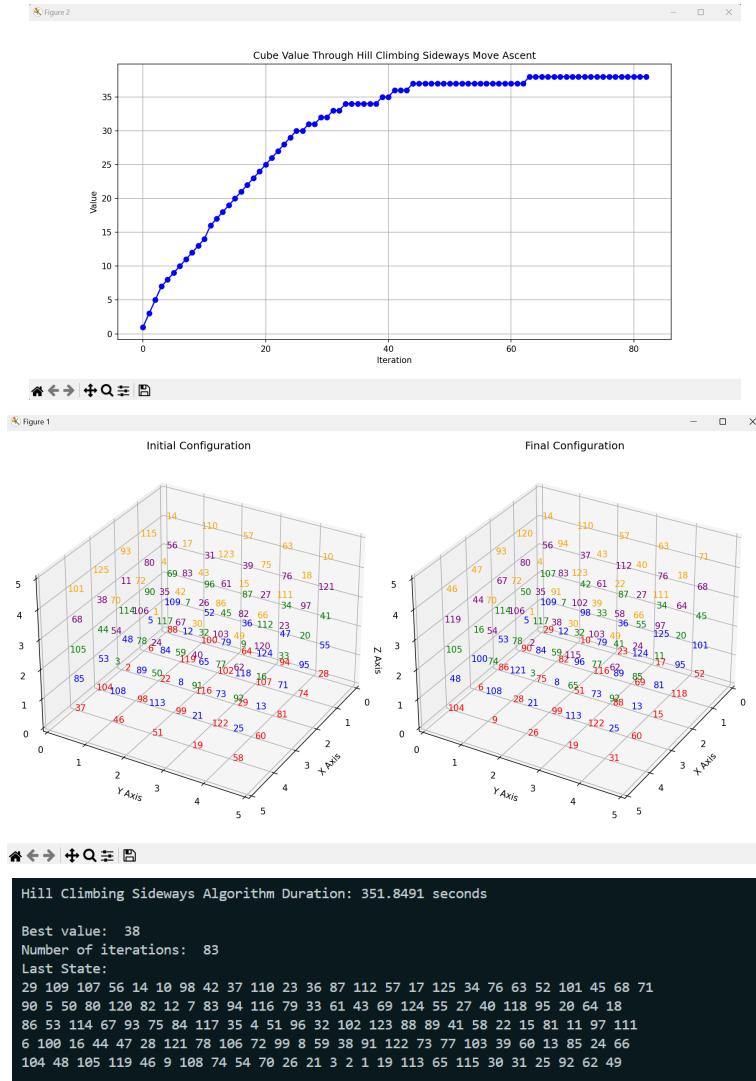
## 2. Percobaan 2



```
Hill Climbing Sideways Algorithm Duration: 237.3453 seconds
Best value: 41
Number of iterations: 56
Last State:
17 25 74 96 103 26 60 114 111 4 42 80 1 73 119 108 100 65 28 14 122 50 61 7 75
117 47 37 23 70 29 110 88 33 63 116 64 109 92 40 38 62 72 49 94 15 32 9 118 48
81 5 84 78 67 107 3 124 89 11 53 87 30 12 44 6 102 18 115 34 68 85 59 21 86
54 2 19 39 83 112 113 35 55 27 93 52 16 56 98 13 57 99 51 97 104 91 43 20 10
46 31 101 79 58 41 121 90 69 24 120 105 95 82 123 106 22 66 45 76 71 36 125 77 8
```

Diperlukan waktu 237 detik dan menghasilkan best value 41 dengan iterasinya 56

## 3. Percobaan 3



Diperlukan waktu 351 detik dan menghasilkan best value 38 dengan iterasi nya 83

## Analisis

- Algoritma ini mencapai best valuenya sekitar 37 - 41. Ini masih jauh dari optimum global yang membutuhkan value 109. Yang berarti bahwa algoritma ini masih sering terjebak di optimum local pada value 37-41. Algoritma ini memiliki peluang untuk bergerak secara mendatar ketika bertemu solusi setara. Namun, algoritma ini tetap terbatas dalam eksplorasi ruang solusi secara keseluruhan sehingga masih masuk dalam optimum local.
- Durasi yang dibutuhkan untuk menyelesaikan algoritma ini sesuai dengan banyaknya maksimal sideways move yang diperbolehkan. Semakin banyak total sideways move, maka akan semakin lama pula waktu yang dibutuhkan. Namun banyaknya total sideways

move ini tetap tidak akan mempengaruhi best valuenya karena mentok hanya ada di 37-41 (terjebak di optimum local).

- Algoritma ini cukup konsisten dengan menghasilkan objective function value maksimal sebesar 37-41 walaupun maksimal sideways movenya bervariasi.
- Algoritma ini lebih baik dari steepest ascent dari segi objective function maksimal yang dicapai, namun akan mengorbankan waktu dan jumlah iterasinya.

## 2.4 Analisis Keseluruhan

Algoritma steepest ascent dihasilkan dengan generate semua kemungkinan successor dan pilih satu terbaik untuk dijadikan neighbor. Lalu apabila neighbor tersebut nilainya lebih besar dari current, maka jadikan menjadi current state. Karena algoritma ini harus generate semua kemungkinan successor, maka waktu yang dibutuhkan menjadi lama yaitu 100an detik.

Algoritma sideways move memperbaiki algoritma steepest ascent dengan memperbolehkan mengambil successor yang nilainya sama dengan current state sehingga memungkinkan untuk mendapat best value yang lebih besar. Namun hal ini tentunya mengorbankan jumlah iterasi dan durasi yang diperlukan. Sideways move memerlukan durasi yang lebih lama sekitar 200an detik.

Algoritma Stochastic hanya perlu membuat 1 successor random lalu bandingkan dengan current state, apabila valuenya lebih besar maka jadikan current state. Karena ia tidak perlu menggenerate semua kemungkinan successor maka waktu yang diperlukan lebih cepat dari steepest ascent yaitu sekitar belasan detik saja

Algoritma Simulated Annealing memperbaiki stochastic dengan memperbolehkan mengambil successor yang nilainya lebih buruk dari current state dengan harapan dapat mencapai value yang lebih tinggi setelahnya. Karena hal ini, maka hasil yang didapat menjadi lebih bagus dan waktu yang diperlukan juga tidak jauh berbeda.

Algoritma Random Restart dilakukan dengan memanggil Hill Climbing Search, apabila tidak menghasilkan optimum global, maka akan dilakukan pencarian ulang sebanyak iterasi

maksimum yang telah ditentukan. Algoritma ini mengatasi kelemahan algoritma Hill Climbing yang akan berhenti ketika tidak dapat menemukan neighbor yang lebih baik dari state nya sekarang. Akibatnya algoritma ini memiliki kemungkinan untuk terjebak di lokal optima lebih kecil daripada Algoritma Hill Climbing.

Algoritma Genetic Algorithm memiliki pendekatan yang berbeda dari algoritma-algoritma local search lain yakni dengan meniru mekanisme evolusi yang terjadi pada seleksi alam. Algoritma ini juga memiliki mekanisme untuk melakukan pencatatan terhadap beberapa kandidat solusi yang disebut populasi yang akan dilakukan proses pemilihan secara random untuk menjadi parent yang kemudian akan dipasangkan oleh parent lain dan dilakukan crossover dan mutasi untuk mendapatkan offspring. Offspring-offspring tersebut kemudian menggantikan populasi awal dan proses tersebut akan diulang sampai iterasi maksimal. Algoritma ini memiliki kompleksitas waktu lebih tinggi dari yang lain karena harus melakukan pencatatan terhadap banyak kandidat solusi dan lebih bergantung kepada keberuntungan. Namun algoritma ini dapat memberikan banyak solusi sekaligus tidak seperti algoritma lain.

## BAB 3

### KESIMPULAN DAN SARAN

#### 3.1 Kesimpulan

Dari Algoritma local search yang telah diimplementasikan pada tugas besar kali ini yaitu pencarian solusi Magic Cube berukuran 5x5x5, algoritma Simulated Annealing merupakan algoritma dengan performa yang paling baik diantara yang lain. Hal ini disebabkan Simulated Annealing memiliki kemampuan untuk meninggalkan lokal optima. Berbeda dengan algoritma pencarian lokal lain yang cenderung terjebak pada lokal optima, Simulated Annealing memiliki kemampuan menjelajahi ruang pencarian yang lebih luas. Namun, secara keseluruhan, dari seluruh algoritma pencarian lokal yang dilakukan, tidak ada satupun yang berhasil menemukan solusi dengan sempurna. Ada beberapa alasan, salah satunya kompleksitas ruang pencarian. Ruang pencarian yang dihasilkan oleh Magic Cube berukuran 5x5x5 sangat tinggi, yang berarti jumlah kemungkinan konfigurasi sangat banyak. Pengaturan dan pencarian solusi di ruang yang sangat besar ini membuat algoritma sulit untuk menjelajahi setiap kemungkinan secara menyeluruh dalam waktu yang terbatas. Meskipun Simulated Annealing dapat menjelajahi lebih luas dibandingkan algoritma pencarian lokal lainnya, masih ada kemungkinan bahwa solusi optimal terletak di area yang belum dijelajahi. Selain itu, alasan lain adalah keterbatasan waktu dan iterasi yang dilakukan. Ini membuat sulit bagi algoritma untuk menjelajahi setiap kemungkinan secara menyeluruh dalam waktu yang terbatas.

#### 3.2 Saran

Pengerjaan tugas besar ini sudah sebagian besar terselesaikan dengan tersisa bonus load file yang seharusnya dapat dikerjakan namun karena kurangnya manajemen waktu dari kelompok kami yang mengakibatkan tidak sempat untuk dilakukan. Untuk selanjutnya tugas-tugas kedepannya harus dapat membagi waktu dan tugas sehingga dapat terselesaikan secara optimal. Untuk tugas besar kedua ini sudah lumayan menguji pemahaman mahasiswa untuk dapat mengaplikasikan ilmu yang dipelajari di kuliah, namun problem yang diberikan yakni magic cube yang berukuran 5x5x5 menurut kami problem spacenya lumayan besar sehingga sulit untuk mencapai solusi global optimum. Oleh karena itu untuk tugas besar selanjutnya dapat diberikan problem space yang lebih mudah dan lebih viable untuk diselesaikan.

## BAB 4

### PEMBAGIAN TUGAS

No	Anggota	Tugas
1	Rici Trisna Putra (13522026)	Algoritma Stochastic Hill Climbing, Implementasi Kelas Cube, Visualisasi kubus
2	Francesco Michael Kusuma (13522038)	Algoritma Simulated Annealing, Algoritma Sideways Move, Plotting Statistik
3	Keanu Amadeus Gonza Wrahatno (13522082)	Algoritma Random Restart Hill Climbing, Algoritma Hill Climbing, Algoritma Steepest Ascent
4	Dimas Bagoes Hendrianto (13522112)	Algoritma Genetic Algorithm, Video animasi

### REFERENSI

Russell, Stuart, and Peter Norvig. Artificial Intelligence: A Modern Approach. 4th ed. Hoboken, NJ: Pearson, 2020.

<https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>

<https://www.magischvierkant.com/three-dimensional-eng/magic-features/>

<https://www.baeldung.com/cs/simulated-annealing>

<https://www.scirp.org/journal/paperinformation?paperid=78834>

<https://www.geeksforgeeks.org/local-search-algorithm-in-artificial-intelligence/>

<https://youtu.be/7hDZyH2E4Yw?si=kH8N0e0C9YN70L0m>

<https://youtu.be/uOj5UNhCPuo?si=kn-zSKMx6Cj6TL77>