

LAPORAN TUGAS BESAR 1 IF3170 INTELEGensi ARTIFISIAL
PENCARIAN SOLUSI DIAGONAL MAGIC CUBE DENGAN LOCAL SEARCH

Diajukan sebagai pemenuhan tugas besar 1



Oleh:

Kelompok 22

1. 13522063 - Shazya Audrea Taufik
2. 13522070 - Marzuli Suhada M
3. 13522085 - Zahira Dina Amalia
4. 13522108 - Muhammad Neo Cicero Koda

Dosen Pengampu :

1. Dr. Nur Ulfa Maulidevi, S.T, M.Sc.
2. Dr. Eng. Ayu Purwarianti, S.T., M.T

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

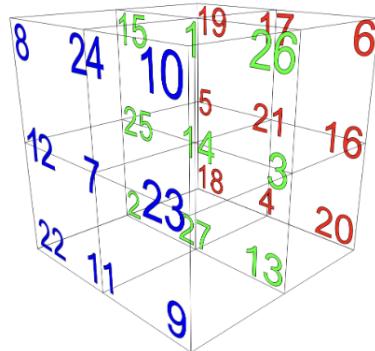
2024

DAFTAR ISI

DAFTAR ISI.....	2
I. DESKRIPSI persoalan.....	3
II. PEMBAHASAN.....	4
1. Objective Function.....	4
2. Implementasi Algoritma.....	8
A. Steepest Ascent Hill Climbing.....	8
B. Hill Climbing with Sideways Move.....	12
C. Stochastic Hill Climbing.....	16
D. Random Restart Hill Climbing.....	19
E. Simulated Annealing.....	21
F. Genetic Algorithm.....	29
3. Hasil Eksperimen dan Analisis.....	34
Hasil Eksperimen.....	34
Test Case 1.....	34
Test Case 2.....	85
Test Case 3.....	136
Analisis.....	188
III. KESIMPULAN DAN SARAN.....	199
1. Kesimpulan.....	199
2. Saran.....	199
IV. PEMBAGIAN TUGAS.....	202
V. REFERENSI.....	203

I. DESKRIPSI PERSOALAN

Masalah yang dihadapi dalam tugas ini adalah menyelesaikan sebuah *diagonal magic cube* berukuran $5 \times 5 \times 5$. *Diagonal magic cube* adalah sebuah kubus yang terdiri dari angka 1 hingga n^3 (pada kasus ini, $5^3 = 125$ angka) tanpa pengulangan dengan n adalah panjang sisi pada kubus tersebut yang diatur sedemikian rupa sehingga jumlah angka pada setiap baris, kolom, tiang, dan diagonal sama dengan suatu nilai yang disebut sebagai *magic number*. *Magic number* tidak harus termasuk dalam angka-angka yang digunakan untuk menyusun kubus, tetapi semua susunan angka dalam kubus yang sebelumnya telah disebutkan harus memenuhi kondisi bahwa jumlahnya sama dengan *magic number*.



Gambar 1. Ilustrasi dari potongan bidang yang ada pada suatu kubus berukuran 3

Dalam permasalahan ini, pencarian konfigurasi angka yang tepat dilakukan menggunakan teknik *local search*. Setiap iterasi dari algoritma *local search* akan melibatkan pertukaran dua angka dalam kubus dengan tujuan untuk mendekatkan hasil ke kondisi yang memenuhi semua aturan dari *diagonal magic cube*. Khusus untuk *genetic algorithm*, boleh dilakukan penukaran posisi lebih dari 2 angka sekaligus dalam satu iterasi (tetapi hanya menukar posisi 2 angka juga diperbolehkan).

II. PEMBAHASAN

1. Objective Function

Dalam pencarian solusi untuk *Diagonal Magic Cube* menggunakan *Local Search*, fungsi *objective* digunakan untuk menghitung nilai kesalahan (*error*) dari sebuah *cube* dalam upaya mencapai kondisi "*magic*." *Diagonal Magic Cube* adalah susunan kubus 3D di mana setiap baris, kolom, *layer*, dan diagonal memiliki jumlah angka yang sama, yang disebut *magic number*. Pada fungsi ini, kesalahan dihitung dengan cara menghitung dua parameter. Parameter pertama adalah jumlah baris, kolom, *layer*, dan diagonal yang tidak sama dengan *magic number* dan parameter kedua adalah selisih semua baris, kolom, *layer*, dan diagonal dari *magic number*. Parameter utama yang akan diminimalkan adalah parameter pertama. Jika parameter pertama sudah tidak bisa diminimalkan lagi, parameter kedua akan diminimalkan. Penjelasan lengkap mengenai fungsi *calculate_objective_value* adalah sebagai berikut:

1. Menghitung Kesalahan pada Baris, Kolom, dan Layer:

- Fungsi ini dimulai dengan menghitung *total_error*, yaitu jumlah *error* untuk setiap baris (*row*), kolom (*column*), dan layer (*layer*) di dalam kubus.
- *self.row_sums*, *self.col_sums*, dan *self.layer_sums* masing-masing berisi jumlah dari semua elemen pada setiap baris, kolom, dan layer di dalam kubus. Jumlah tersebut disimpan secara eksplisit untuk meminimasi waktu perhitungan *objective function*.
- *row_error_count*, *col_error_count*, dan *layer_error_count* menghitung banyaknya baris, kolom, dan layer yang jumlahnya tidak sesuai dengan *magic number*.

2. Menghitung Kesalahan pada Diagonal:

- *diagonal_error_count* diinisialisasi dengan nilai nol dan akan dihitung untuk setiap diagonal yang tidak sesuai dengan *magic number*.
- Fungsi menghitung enam jenis diagonal potongan bidang di dalam *cube* yang mungkin:

- Main Diagonal: Diagonal dari kiri atas ke kanan bawah pada layer yang sama.
 - Anti Diagonal: Diagonal dari kanan atas ke kiri bawah pada layer yang sama.
 - Vertical Diagonal: Diagonal yang melintasi layer dari atas ke bawah pada baris tertentu.
 - Vertical Anti Diagonal: Diagonal yang melintasi layer dari atas ke bawah dalam arah sebaliknya pada baris tertentu.
 - Depth Diagonal: Diagonal yang melintasi layer dari depan ke belakang pada kolom tertentu.
 - Depth Anti Diagonal: Diagonal yang melintasi layer dari depan ke belakang dalam arah sebaliknya pada kolom tertentu.
- Fungsi juga menghitung empat diagonal ruang, yaitu ($n = 5$):
 - Diagonal dari sudut (0, 0, 0) hingga (n-1, n-1, n-1)
 - Diagonal dari sudut (0, 0, n-1) hingga (n-1, n-1, 0)
 - Diagonal dari sudut (0, n-1, 0) hingga (n-1, 0, n-1)
 - Diagonal dari sudut (0, n-1, n-1) hingga (n-1, 0, 0)

3. Menghitung Total Kesalahan:

- total_error_count adalah jumlah total kesalahan dari semua baris, kolom, layer, dan diagonal yang tidak sesuai dengan *magic number*.

4. Nilai Akhir:

- Fungsi mengembalikan total_error (jumlah kesalahan keseluruhan dari perbedaan nilai dengan *magic number*) dan total_error_count (jumlah bagian dalam kubus yang belum sesuai dengan *magic number*).
- Kedua nilai ini dapat digunakan oleh algoritma *Local Search* untuk menilai kualitas solusi saat ini dan mengarahkan perbaikan solusi pada iterasi selanjutnya.

Secara keseluruhan, semakin kecil nilai total_error dan total_error_count, semakin dekat solusi saat ini menuju *Diagonal Magic Cube*.

Berikut adalah cuplikan kode dari objective function yang diimplementasikan:

```
def calculate_objective_value(self):
    total_error = (np.abs(self.row_sums - self.magic_number).sum() +
                   np.abs(self.col_sums - self.magic_number).sum() +
                   np.abs(self.layer_sums - self.magic_number).sum())

    row_error_count = np.sum(np.abs(self.row_sums -
                                    self.magic_number) > 0)
    col_error_count = np.sum(np.abs(self.col_sums -
                                    self.magic_number) > 0)
    layer_error_count = np.sum(np.abs(self.layer_sums -
                                    self.magic_number) > 0)

    diagonal_error_count = 0

    for i in range(self.n):
        main_diagonal_sum = self(cube[i, range(self.n),
range(self.n)].sum())
        anti_diagonal_sum = self(cube[i, range(self.n),
range(self.n-1, -1, -1)].sum())
        vertical_diagonal_sum = self(cube[range(self.n), i,
range(self.n)].sum())
        vertical_anti_diagonal_sum = self(cube[range(self.n), i,
range(self.n-1, -1, -1)].sum())
        depth_diagonal_sum = self(cube[range(self.n), range(self.n),
i].sum())
        depth_anti_diagonal_sum = self(cube[range(self.n),
range(self.n-1, -1, -1), i].sum())

        main_diagonal_error = abs(main_diagonal_sum -
self.magic_number)
```

```
        anti_diagonal_error = abs(anti_diagonal_sum -
self.magic_number)
        vertical_diagonal_error = abs(vertical_diagonal_sum -
self.magic_number)
        vertical_anti_diagonal_error = abs(vertical_anti_diagonal_sum -
self.magic_number)
        depth_diagonal_error = abs(depth_diagonal_sum -
self.magic_number)
        depth_anti_diagonal_error = abs(depth_anti_diagonal_sum -
self.magic_number)

        if main_diagonal_error > 0:
            total_error += main_diagonal_error
            diagonal_error_count += 1

        if anti_diagonal_error > 0:
            total_error += anti_diagonal_error
            diagonal_error_count += 1

        if vertical_diagonal_error > 0:
            total_error += vertical_diagonal_error
            diagonal_error_count += 1

        if vertical_anti_diagonal_error > 0:
            total_error += vertical_anti_diagonal_error
            diagonal_error_count += 1

        if depth_diagonal_error > 0:
            total_error += depth_diagonal_error
            diagonal_error_count += 1

        if depth_anti_diagonal_error > 0:
            total_error += depth_anti_diagonal_error
            diagonal_error_count += 1
```

```

ruang_diagonals = [
    self.cube[range(self.n), range(self.n), range(self.n)].sum(),
    self.cube[range(self.n), range(self.n), range(self.n-1, -1,
-1)].sum(),
    self.cube[range(self.n), range(self.n-1, -1, -1),
range(self.n)].sum(),
    self.cube[range(self.n), range(self.n-1, -1, -1),
range(self.n-1, -1, -1)].sum()
]

ruang_error_count = 0
for ruang_diagonal_sum in ruang_diagonals:
    ruang_error = abs(ruang_diagonal_sum - self.magic_number)
    if ruang_error > 0:
        total_error += ruang_error
        ruang_error_count += 1

total_error_count = row_error_count + col_error_count +
layer_error_count + diagonal_error_count + ruang_error_count

return total_error, total_error_count

```

2. Implementasi Algoritma

A. Steepest Ascent Hill Climbing

Algoritma *Steepest Ascent Hill Climbing* adalah metode *local search* yang digunakan untuk menemukan solusi optimal atau mendekati optimal dengan melakukan pencarian bertahap dalam ruang solusi. Proses dimulai dari sebuah solusi awal yang dievaluasi menggunakan fungsi objektif yang sebelumnya telah dijelaskan. Setelah itu, algoritma akan mengevaluasi nilai *objective value* dari semua tetangga kubus saat ini yang diperoleh dengan menukar posisi dua angka pada kubus. Dari semua tetangga yang tersedia, algoritma memilih tetangga dengan *cost terendah*.

Jika ditemukan solusi *neighbor* yang lebih baik, algoritma berpindah ke solusi tersebut dan mengulangi proses evaluasi *neighbor*. Algoritma ini terus berlanjut hingga tidak ada lagi solusi tetangga yang lebih baik, yaitu ketika algoritma telah mencapai titik optimum lokal. Algoritma *Steepest Ascent Hill Climbing* memiliki keterbatasan karena cenderung terjebak di optimum lokal, terutama dalam masalah dengan banyak puncak atau solusi suboptimal sehingga solusi yang ditemukan mungkin bukan solusi terbaik. Pada implementasi ini, algoritma Steepest Ascent Hill Climbing direpresentasikan dalam kelas *SteepestAscent*. Berikut adalah implementasi algoritma Steepest Ascent Hill Climbing:

- 1) Kelas *SteepestAscent* beserta konstruktornya

Proses	Menginisialisasi kelas <i>SteepestAscent</i> , atribut <i>initial_cube</i> dengan konfigurasi kubus pada awal pencarian, serta <i>objective_history</i> yang menyimpan nilai <i>objective value</i> untuk setiap iterasi.
Output	<i>Instance</i> kelas <i>SteepestAscent</i>
<pre>class SteepestAscent: def __init__(self, initial_cube): self(cube = initial_cube self.objective_history = []</pre>	

- 2) Fungsi *get_neighbor*

Proses	Mengenumerasi semua <i>neighbor</i> kubus yang ada dan memilih <i>neighbor</i> dengan <i>objective value</i> terbaik. Pemilihan tersebut berdasarkan jumlah bagian kubus yang <i>error</i> tersedikit terlebih dahulu, dan dilanjutkan dengan jumlah selisih bagian kubus dengan <i>magic number</i> jika dua <i>neighbor</i> memiliki jumlah bagian <i>error</i> yang sama.
---------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Output	<i>Neighbor</i> terbaik dari kubus
	<pre> def get_neighbor(self): best_neighbor = None best_cost, best_error_count = self(cube).calculate_objective_value() for i1 in range(self(cube).n): for j1 in range(self(cube).n): for k1 in range(self(cube).n): for i2 in range(self(cube).n): for j2 in range(self(cube).n): for k2 in range(self(cube).n): if (i1, j1, k1) != (i2, j2, k2): neighbor = self(cube).copy() neighbor.swap(i1, j1, k1, i2, j2, k2) neighbor_cost, neighbor_error_count = neighbor.calculate_objective_value() if neighbor_error_count < best_error_count: best_neighbor = neighbor best_cost = neighbor_cost best_error_count = neighbor_error_count elif neighbor_error_count == best_error_count and neighbor_cost < best_cost: best_neighbor = neighbor best_cost = neighbor_cost return best_neighbor </pre>

3) Fungsi plot_results

Proses	Menampilkan <i>plot</i> fungsi objektif terhadap banyak iterasi serta menambahkan <i>label</i> dan legenda
Output	<i>Plot</i> fungsi objektif terhadap banyak iterasi
<pre>def plot_results(self): plt.figure(figsize=(12, 5)) plt.plot(self.objective_history, label='Objective Value (total_error)') plt.xlabel("Iterations") plt.ylabel("Objective Value") plt.title("Objective Function Over Iterations") plt.legend() plt.show()</pre>	

4) Fungsi hill_climb

Proses	Melakukan algoritma <i>hill climbing</i> dengan mengambil <i>neighbor</i> terbaik dan mencari <i>neighbor</i> terbaik dari <i>neighbor</i> tersebut hingga nilai <i>objective value neighbor</i> tidak lebih baik dari nilai <i>objective value</i> kubus saat ini.
Output	Konfigurasi akhir kubus setelah pencarian, jumlah selisih <i>magic number</i> dengan setiap bagian kubus, jumlah bagian kubus yang masih salah, serta durasi pencarian algoritma.
<pre>def hill_climb(self): start_time = time.time() i = 0 while True: best_cost, best_error_count = self.cube.calculate_objective_value()</pre>	

```

        self.objective_history.append(best_error_count)

        neighbor = self.get_neighbor()
        if not neighbor or
(neighbor.calculate_objective_value()[0] >= best_cost and
neighbor.calculate_objective_value()[1] >= best_error_count):
            break

        self(cube = neighbor
        i += 1

        print(f"Jumlah iterasi: {i}")

        end_time = time.time()
        duration = end_time - start_time
        final_cost, final_error_count =
self(cube).calculate_objective_value()
        self.plot_results()
        return self(cube), final_cost, final_error_count, duration
    
```

B. Hill Climbing with Sideways Move

Algoritma Hill Climbing with Sideways Move memiliki langkah yang mirip dengan Steepest Ascent Hill Climbing, namun memungkinkan *sideways move*, yaitu berpindah ke *neighbor* dengan *objective value* yang sama dengan kubus saat ini. Berikut adalah implementasi algoritma tersebut:

- 1) Kelas SidewaysMove beserta konstruktornya

Proses	Menginisialisasi kelas SidewaysMove, atribut initial_cube dengan konfigurasi kubus pada awal pencarian, dan objective_history untuk menyimpan nilai <i>objective value</i> tiap iterasi
---------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Output	<i>Instance kelas SidewaysMove</i>
<pre>class SidewaysMove: def __init__(self, initial_cube, max_sideways): self.cube = initial_cube self.max_sideways = max_sideways self.objective_history = []</pre>	

2) Fungsi get_neighbor

Proses	Mengenumerasi semua <i>neighbor</i> kubus yang ada dan memilih <i>neighbor</i> dengan <i>objective value</i> terbaik.
Output	<i>Neighbor</i> terbaik dari kubus
<pre>def get_neighbor(self): best_neighbor = None _, best_error_count = self.cube.calculate_objective_value() for i1 in range(self.cube.n): for j1 in range(self.cube.n): for k1 in range(self.cube.n): for i2 in range(self.cube.n): for j2 in range(self.cube.n): for k2 in range(self.cube.n): if (i1, j1, k1) != (i2, j2, k2): neighbor = self.cube.copy() neighbor.swap(i1, j1, k1, i2, j2, k2) _, neighbor_error_count = neighbor.calculate_objective_value() if neighbor_error_count <=</pre>	

```

best_error_count:
    best_neighbor = neighbor
    best_error_count =
    neighbor_error_count

    return best_neighbor

```

3) Fungsi plot_results

Proses	Menampilkan <i>plot</i> fungsi objektif terhadap banyak iterasi serta menambahkan <i>label</i> dan legenda
Output	<i>Plot</i> fungsi objektif terhadap banyak iterasi
<pre> def plot_results(self): plt.figure(figsize=(12, 5)) plt.plot(self.objective_history, label='Objective Value (total_error)') plt.xlabel("Iterations") plt.ylabel("Objective Value") plt.title("Objective Function Over Iterations") plt.legend() plt.show() </pre>	

4) Fungsi hill_climb

Proses	Melakukan algoritma <i>hill climbing</i> dengan mengambil <i>neighbor</i> terbaik dan mencari <i>neighbor</i> terbaik dari <i>neighbor</i> tersebut hingga nilai <i>objective value</i> <i>neighbor</i> tidak lebih baik dari nilai <i>objective value</i> kubus saat ini.
Output	Konfigurasi akhir kubus setelah pencarian, jumlah selisih

magic number dengan setiap bagian kubus, jumlah bagian kubus yang masih salah, serta durasi pencarian algoritma.

```
def hill_climb(self):
    start_time = time.time()
    i = 0
    n_sideways = 0

    while True:
        best_cost, best_error_count =
self.cube.calculate_objective_value()
        self.objective_history.append(best_error_count)

        neighbor = self.get_neighbor()

        if not neighbor:
            break

        neighbor_cost, neighbor_error_count =
neighbor.calculate_objective_value()

        if neighbor_error_count > best_error_count:
            break
        elif neighbor_error_count == best_error_count:
            n_sideways += 1
            if n_sideways >= self.max_sideways:
                break
        else:
            n_sideways = 0

        self.cube = neighbor
        i += 1

    print(f"Jumlah iterasi: {i}")
```

```

        end_time = time.time()
        duration = end_time - start_time
        final_cost, final_error_count =
        self.cube.calculate_objective_value()
        self.plot_results()
        return self.cube, final_cost, final_error_count, duration
    
```

C. Stochastic Hill Climbing

Algoritma *hill climbing* ini bekerja dengan mengambil *neighbor* secara random dari konfigurasi saat ini dan menjadikan *neighbor* sebagai *current* hanya jika nilai *objective value* *neighbor* lebih baik.

- 1) Kelas Stochastic beserta konstruktornya

Proses	Menginisialisasi kelas Stochastic, atribut <i>initial_cube</i> dengan konfigurasi kubus pada awal pencarian, serta atribut <i>max_iteration</i> dengan jumlah iterasi maksimum
Output	<i>Instance</i> kelas Stochastic

```

class Stochastic:
    def __init__(self, initial_cube, max_iteration=50000):
        self.cube = initial_cube
        self.max_iteration = max_iteration
    
```

- 2) Fungsi *get_random_neighbor*

Proses	Mengambil dua posisi acak yang berbeda dari kubus dan mengembalikan <i>neighbor</i> yang merupakan hasil pertukaran posisi tersebut
---------------	-------------------------------------------------------------------------------------------------------------------------------------

Output	<i>Neighbor</i> acak kubus saat ini
<pre>def get_random_neighbor(self): i1, j1, k1 = random.randint(0, self.cube.n - 1), random.randint(0, self.cube.n - 1), random.randint(0, self.cube.n - 1) i2, j2, k2 = random.randint(0, self.cube.n - 1), random.randint(0, self.cube.n - 1), random.randint(0, self.cube.n - 1) while (i1, j1, k1) == (i2, j2, k2): i2, j2, k2 = random.randint(0, self.cube.n - 1), random.randint(0, self.cube.n - 1), random.randint(0, self.cube.n - 1) neighbor = self.cube.copy() neighbor.swap(i1, j1, k1, i2, j2, k2) return neighbor</pre>	

3) Fungsi plot_results

Proses	Melakukan <i>plotting</i> terhadap <i>objective value</i> terhadap jumlah iterasi saat ini
Output	Visualisasi <i>objective value</i> terhadap jumlah iterasi
<pre>def plot_results(self): plt.figure(figsize=(12, 5)) plt.plot(self.objective_history, label='Objective Value (total_error)') plt.xlabel("Iterations") plt.ylabel("Objective Value") plt.title("Objective Function Over Iterations") plt.legend()</pre>	

```
plt.show()
```

4) Fungsi hill_climb

Proses	<ul style="list-style-type: none">• Melakukan algoritma <i>stochastic hill climbing</i> dan melakukan <i>plotting</i> terhadap hasil akhir pencarian.• Selama iterasi kurang dari <code>max_iteration</code>, fungsi akan mengambil <i>neighbor</i> secara acak.• Jika <i>neighbor</i> tersebut memiliki <i>cost</i> yang lebih kecil daripada kubus saat ini, akan dilakukan penukaran.• Hal tersebut dilakukan hingga iterasi mencapai iterasi maksimum.• Algoritma juga akan mencatat durasi algoritma dijalankan
Output	Konfigurasi akhir kubus setelah pencarian, jumlah selisih <i>magic number</i> dengan setiap bagian kubus, jumlah bagian kubus yang masih salah, durasi pencarian algoritma, serta jumlah iterasi total.

```
def hill_climb(self):  
    start_time = time.time()  
    iteration = 0  
  
    while iteration < self.max_iteration:  
        current_cost = self.cube.calculate_objective_value()  
        self.objective_history.append(current_cost[1])  
        # print(f"Current Cost: {current_cost}, Iteration:  
        {iteration}")  
  
        neighbor = self.get_random_neighbor()  
        neighbor_cost = neighbor.calculate_objective_value()
```

```

        # print(f"Neighbor Cost: {neighbor_cost}")

        if neighbor_cost < current_cost:
            self(cube = neighbor

            iteration += 1

        print(f"Jumlah iterasi: {iteration}")

        end_time = time.time()
        duration = end_time - start_time
        self.plot_results()
        return self(cube, self(cube.calculate_objective_value()[0],
self(cube.calculate_objective_value()[1], duration, iteration

```

D. Random Restart Hill Climbing

Algoritma Random Restart Hill Climbing adalah algoritma Hill Climbing yang dilakukan berkali-kali. Algoritma Random Restart yang diimplementasikan pada kasus ini adalah Stochastic Hill Climbing karena Steepest Ascent Hill Climbing akan memakan waktu terlalu lama jika dilakukan berkali-kali.

- 1) Kelas RandomRestart beserta konstruktornya

Proses	Menginisialisasi kelas RandomRestart dengan atribut berupa jumlah <i>restart</i> dan jumlah iterasi maksimum serta objective_history untuk menyimpan daftar objective_value pada tiap iterasi
Output	Instansiasi kelas RandomRestart

```

class RandomRestart:
    def __init__(self, n_restarts=10, max_iteration=3000):
        self.n_restarts = n_restarts

```

```

        self.max_iteration = max_iteration
        self.objective_history = []

```

2) Fungsi plot_results

Proses	Melakukan <i>plotting</i> terhadap <i>objective value</i> terhadap jumlah iterasi saat ini
Output	Visualisasi <i>objective value</i> terhadap jumlah iterasi
<pre> def plot_results(self): plt.figure(figsize=(12, 5)) plt.plot(self.objective_history, label='Objective Value (total_error)') plt.xlabel("Iterations") plt.ylabel("Objective Value") plt.title("Objective Function Over Iterations") plt.legend() plt.show() </pre>	

3) Fungsi hill_climb

Proses	Melakukan algoritma <i>stochastic hill climbing</i> secara berkali-kali. Pada akhir setiap <i>restart</i> , konfigurasi kubus serta <i>objective value</i> yang terbaik akan diperbarui.
Output	Konfigurasi akhir kubus setelah pencarian, jumlah selisih <i>magic number</i> dengan setiap bagian kubus, jumlah bagian kubus yang masih salah, serta durasi pencarian algoritma.
<pre> def hill_climb(self): best_solution = None best_cost = float('inf') best_error_count = float('inf') </pre>	

```

        total_duration = 0

        for restart in range(self.n_restarts):
            initial_cube = MagicCube(5)
            shc = Stochastic(initial_cube,
max_iteration=self.max_iteration)

                solution, cost, error_count, duration, iteration =
shc.hill_climb()
                total_duration += duration

                if error_count < best_error_count or (error_count ==
best_error_count and cost < best_cost):
                    best_solution = solution
                    best_cost = cost
                    best_error_count = error_count

                self.objective_history.append(error_count)
                print(f"Jumlah iterasi restart {restart + 1}:
{iteration}")
                print(f"Jumlah restart: {self.n_restarts}")
                self.plot_results()
            return best_solution, best_cost, best_error_count,
total_duration

```

E. Simulated Annealing

Simulated Annealing (SA) adalah metode pencarian solusi yang terinspirasi oleh proses *annealing* dalam metalurgi, di mana logam dipanaskan hingga suhu tinggi dan kemudian didinginkan secara bertahap untuk mencapai struktur kristal yang stabil dan meminimalkan energi. Dalam konteks optimasi, metode ini digunakan untuk menemukan solusi yang optimal dengan menjelajahi ruang solusi secara bertahap dan menghindari jebakan pada solusi lokal.

Dalam penerapannya untuk masalah seperti *Diagonal Magic Cube*, *Simulated Annealing* bertujuan untuk menemukan susunan angka dalam struktur kubus 3D yang memenuhi syarat tertentu, misalnya jumlah setiap baris, kolom, layer, dan diagonalnya harus sama dengan *magic number*. Langkah-langkah Utama dalam *Simulated Annealing*:

1. Inisialisasi: Proses dimulai dengan memilih solusi awal secara acak (contohnya, susunan awal angka dalam kubus). Selain itu, parameter penting seperti temperatur awal, *cooling rate* (laju penurunan suhu), dan *stopping temperature* (suhu akhir) ditentukan terlebih dahulu.
2. Evaluasi Solusi (*Cost Function*): Setiap solusi memiliki nilai objective atau cost yang dihitung berdasarkan seberapa jauh solusi tersebut dari kondisi ideal. Dalam kasus *Diagonal Magic Cube*, cost dapat diukur berdasarkan jumlah penyimpangan setiap baris, kolom, layer, dan diagonal dari *magic number* yang diinginkan. Solusi dengan cost yang lebih rendah lebih dekat ke solusi optimal.
3. Pertimbangan Solusi Tetangga (*Neighbor Solution*): Pada setiap langkah, solusi tetangga dipilih dengan melakukan sedikit perubahan pada solusi saat ini, misalnya dengan menukar dua angka dalam kubus (operasi *swap*). Solusi tetangga ini kemudian dihitung cost-nya.
4. Penerimaan Solusi: Jika solusi tetangga memiliki cost lebih rendah daripada solusi saat ini, solusi tersebut diterima sebagai solusi baru. Namun, jika cost solusi tetangga lebih tinggi, solusi tersebut masih memiliki peluang untuk diterima, yang ditentukan oleh fungsi probabilitas penerimaan:

$$P = e^{\frac{-\Delta E}{T}}$$

Di mana, ΔE adalah selisih cost antara solusi saat ini dan solusi tetangga dan T adalah suhu saat ini. Pada suhu tinggi, peluang untuk menerima solusi dengan cost lebih tinggi relatif besar, yang memungkinkan eksplorasi ruang solusi secara luas. Ketika suhu menurun, peluang ini menjadi semakin kecil, mendorong algoritma menuju konvergensi.

5. Pendinginan (*Cooling*): Setelah setiap iterasi, suhu diturunkan secara bertahap sesuai *cooling rate*. Ini mengurangi kemungkinan menerima solusi dengan *cost* yang lebih tinggi seiring waktu, sehingga algoritma semakin fokus pada pencarian solusi dengan *cost* lebih rendah.
6. Penghentian (*Stopping Condition*): Proses berlanjut hingga suhu mencapai *stopping temperature* atau tidak ditemukan perbaikan pada solusi dalam beberapa iterasi berturut-turut.

Source Codenya adalah sebagai berikut:

- 1) Kelas Simulated Annealing beserta konstruktornya

Proses	<p>Menginisialisasi algoritma dengan berbagai parameter:</p> <ol style="list-style-type: none"> 1. Parameter Suhu: current_temperature, cooling_rate, dan stopping_temperature: Atur suhu awal, laju pendinginan, dan suhu akhir yang menentukan kapan algoritma berhenti. 2. Penyimpanan Riwayat: objective_history dan probability_history: Menyimpan nilai objektif dan probabilitas penerimaan untuk setiap iterasi, membantu melacak perkembangan solusi. 3. Status Awal dan Iterasi: <ul style="list-style-type: none"> • initial_state dan initial_objective_value: Menyimpan solusi awal dan nilai objektif awal untuk dibandingkan dengan solusi akhir. • iteration dan stuck_count: Melacak jumlah iterasi dan frekuensi terjebaknya algoritma disolusi lokal.
Output	Instance algoritma simulated annealing dengan parameter yang telah diset

```

class SimulatedAnnealing:
    def __init__(self, initial_cube, initial_temperature, cooling_rate,
stopping_temperature, tolerance):
        self(cube = initial_cube
        self.current_temperature = initial_temperature
        self.cooling_rate = cooling_rate
        self.stopping_temperature = stopping_temperature
        self.tolerance = tolerance
        self.objective_history = []
        self.probability_history = []
        self.stuck_count = 0
        self.initial_state = np.copy(initial_cube)
        self.initial_objective_value =
initial_cube.calculate_objective_value()
        self.iteration = 0

```

2) Fungsi acceptance_probability

Proses	<ul style="list-style-type: none"> Menghitung probabilitas penerimaan solusi tetangga (neighbor) berdasarkan perbedaan nilai <i>cost</i> antara solusi saat ini dan solusi tetangga. Jika <i>cost</i> tetangga lebih rendah dari solusi saat ini, solusi diterima dengan probabilitas 1.0. Jika <i>cost</i> tetangga lebih tinggi, solusi mungkin diterima berdasarkan probabilitas eksponensial: $P = e^{\frac{neighbor\ cost - current\ cost}{T}}$
Output	Mengembalikan probabilitas penerimaan solusi tetangga, sebuah nilai antara 0 dan 1.

```

def acceptance_probability(self, old_cost, new_cost):
    old_total_error = old_cost[0]
    new_total_error = new_cost[0]

    if new_total_error < old_total_error:
        return 1.0

```

```

        else:
            return math.exp((old_total_error - new_total_error) /
self.current_temperature)

```

3) Fungsi get_neighbor

Proses	<ul style="list-style-type: none"> • Memilih dua posisi acak dalam kubus (i_1, j_1, k_1 dan i_2, j_2, k_2) untuk melakukan operasi <i>swap</i>. • Menghasilkan solusi tetangga dengan menukar dua elemen dalam kubus pada posisi tersebut.
Output	Mengembalikan objek neighbor sebagai kandidat solusi baru yang diperoleh melalui perubahan (swap) kecil pada solusi saat ini.
<pre> def get_neighbor(self): i1, j1, k1 = random.randint(0, self.cube.n - 1), random.randint(0, self.cube.n - 1), random.randint(0, self.cube.n - 1) i2, j2, k2 = random.randint(0, self.cube.n - 1), random.randint(0, self.cube.n - 1), random.randint(0, self.cube.n - 1) neighbor = self.cube.copy() neighbor.swap(i1, j1, k1, i2, j2, k2) return neighbor </pre>	

4) Fungsi anneal

Proses	<p>Menjalankan algoritma utama simulated annealing dengan:</p> <ol style="list-style-type: none"> 1. Inisialisasi Solusi: Menyimpan solusi awal sebagai <code>current_solution</code> dan menghitung nilai <code>cost</code> awal (<code>current_cost</code>). Menetapkan solusi ini juga sebagai solusi terbaik (<code>best_solution</code> dan <code>best_cost</code>). 2. Loop Utama (Sampai Suhu Mencapai Batas): <ul style="list-style-type: none"> • Menghasilkan solusi tetangga (<code>neighbor</code>) menggunakan <code>get_neighbor</code>.
---------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

	<ul style="list-style-type: none"> • Menghitung nilai cost dari tetangga (new_cost). • Menggunakan acceptance_probability untuk menentukan probabilitas menerima solusi tetangga. • Memutuskan untuk menerima atau menolak solusi tetangga berdasarkan probabilitas tersebut. • Memperbarui solusi terbaik (best_solution) jika tetangga memiliki cost lebih rendah. • Mendinginkan suhu dengan mengalikan current_temperature dengan cooling_rate. <p>3. Menghentikan Loop:</p> <p>Loop berakhir jika suhu mencapai stopping_temperature atau jika solusi terbaik telah mencapai batas toleransi (tolerance).</p> <p>4. Menyimpan Hasil Akhir:</p> <p>Menyimpan solusi akhir (final_state), mencatat durasi waktu eksekusi, dan menghitung nilai cost akhir.</p>
Output	<ul style="list-style-type: none"> • Mencetak final objective value dari solusi terbaik, durasi waktu, dan jumlah stuck count. • Mengembalikan solusi terbaik (best_solution), nilai cost terbaik (best_cost), dan durasi proses.
<pre>def anneal(self): start_time = time.time() current_solution = self.cube current_cost = current_solution.calculate_objective_value() best_solution = current_solution best_cost = current_cost while self.current_temperature > self.stopping_temperature: neighbor = self.get_neighbor() new_cost = neighbor.calculate_objective_value()</pre>	

```

        acceptance_prob = self.acceptance_probability(current_cost,
new_cost)

        # Only record data every 100 iterations for plotting
        if self.iteration % 100 == 0:
            self.objective_history.append(current_cost[1])
            self.probability_history.append(acceptance_prob)

        if acceptance_prob == 1 or acceptance_prob >
random.uniform(0, 1):
            current_solution = neighbor
            current_cost = new_cost
        else:
            self.stuck_count += 1

        if new_cost[1] < best_cost[1] or (new_cost[1] ==
best_cost[1] and new_cost[0] < best_cost[0]):
            best_solution = neighbor
            current_cost = new_cost
            best_cost = new_cost

        self.current_temperature *= self.cooling_rate
        self.iteration += 1

        if best_cost[1] <= self.tolerance:
            break

    end_time = time.time()
    duration = end_time - start_time
    self.final_state = np.copy(best_solution.cube)
    final_objective_value =
best_solution.calculate_objective_value()

    show_cube(best_solution)
    print(f"\nFinal Objective Function Value (total_error,
error_count): {final_objective_value}")
    print(f"Duration: {duration:.2f} seconds")
    print(f"Frequency of being 'stuck' at local optima:
{self.stuck_count}")

```

```

        self.plot_results(duration, best_cost[1])

    return best_solution, best_cost, duration

```

5) Fungsi plot_results

Proses	<ul style="list-style-type: none"> • Membuat grafik Objective Value yang menampilkan perubahan nilai objektif setiap 100 iterasi. • Membuat grafik Acceptance Probability yang menampilkan probabilitas penerimaan solusi setiap 100 iterasi.
Output	Dua grafik yang menunjukkan perkembangan nilai objektif dan probabilitas penerimaan solusi tetangga selama proses pencarian solusi.

```

def plot_results(self, duration, best_cost):
    plt.figure(figsize=(12, 5))
    plt.plot(range(0, self.iteration, 100), self.objective_history,
label='Objective Value (total_error)')
    plt.xlabel("Iterations (every 100)")
    plt.ylabel("Objective Value")
    plt.title("Objective Function Over Iterations (every 100)")
    plt.legend()
    plt.show()

    plt.figure(figsize=(12, 5))
    plt.plot(range(0, self.iteration, 100),
self.probability_history, label='Acceptance Probability (e^ΔE/T)')
    plt.xlabel("Iterations (every 100)")
    plt.ylabel("Acceptance Probability")
    plt.title("Acceptance Probability Over Iterations (every 100)")
    plt.legend()
    plt.show()

```

F. Genetic Algorithm

Genetic Algorithm (Algoritma genetika) merupakan metode pencarian berbasis populasi yang terinspirasi dari proses evolusi biologis. Dalam konteks pencarian solusi *Diagonal Magic Cube*, algoritma ini digunakan untuk menemukan susunan angka dalam struktur kubus 3D sedemikian rupa sehingga jumlah setiap baris, kolom, layer, dan diagonal sama dengan magic constant tertentu.

Implementasi algoritma genetika untuk magic cube dimulai dengan membangkitkan populasi awal yang terdiri dari beberapa kandidat solusi. Setiap kandidat solusi merepresentasikan sebuah kubus dengan ukuran $n \times n \times n$ yang berisi permutasi angka dari 1 hingga n^3 . Untuk memastikan solusi yang valid, setiap angka harus muncul tepat satu kali dalam kubus.

Proses evolusi dilakukan melalui serangkaian operasi genetika yang meliputi seleksi, crossover, dan mutasi. Seleksi dilakukan berdasarkan nilai *fitness* yang dihitung dari total penyimpangan terhadap magic constant. Semakin kecil penyimpangannya, semakin baik *fitness*-nya. Individu dengan *fitness* terbaik memiliki peluang lebih besar untuk terpilih sebagai parent.

Setelah proses seleksi, algoritma melakukan crossover antara dua parent terpilih untuk menghasilkan offspring baru. Proses ini dilakukan dengan probabilitas tertentu untuk setiap posisi dalam kubus. Selanjutnya, *offspring* yang dihasilkan mengalami mutasi melalui operasi swap untuk mempertahankan keragaman populasi dan menghindari konvergensi prematur. Langkah-langkah dalam algoritma *genetic* ini lebih jelas sebagai berikut:

1. Inisialisasi Populasi: Dibuat beberapa individu secara acak yang akan menjadi populasi awal.
2. Seleksi: Individu-individu dalam populasi dievaluasi berdasarkan nilai *fitness*-nya. Individu dengan nilai *fitness* terbaik dipilih untuk melanjutkan ke generasi berikutnya.

3. Crossover: Dua individu dipilih untuk disilangkan dengan metode yang memadukan elemen-elemen mereka guna membentuk individu baru (*child*) yang mungkin lebih optimal.
4. Mutasi: Individu yang dihasilkan dari proses persilangan diubah secara acak pada satu atau lebih elemennya. Mutasi ini bertujuan untuk menghindari *local optima* dan menjaga keberagaman dalam populasi.
5. Evaluasi dan Pengulangan: Algoritma terus dievaluasi hingga mencapai jumlah iterasi maksimum atau ketika ditemukan solusi yang memenuhi kriteria optimal.

Source Codenya adalah sebagai berikut:

1) Fungsi fitness

Proses	Mengembalikan nilai <i>fitness</i> sebagai kebalikan dari total error. Nilai <i>fitness</i> yang lebih tinggi menunjukkan individu yang lebih optimal.
Output	Nilai <i>fitness value</i>
<pre>def fitness(self): total_error, _ = self.calculate_objective_value() return 1 / (1 + total_error)</pre>	

2) Fungsi __init__

Proses	<ul style="list-style-type: none"> • Menginisialisasi parameter algoritma (ukuran kubus, ukuran populasi, maksimum iterasi) • Membangkitkan populasi awal dengan membuat sejumlah MagicCube • Menyiapkan log untuk mencatat nilai objektif
Output	Instance algoritma genetika dengan populasi awal dan parameter yang telah diset
<pre>def __init__(self, cube_size, population_size, max_iterations): self.cube_size = cube_size</pre>	

```

        self.population_size = population_size
        self.max_iterations = max_iterations
        self.population = [MagicCube(cube_size) for _ in
range(population_size)]
        self.objective_log = []

```

3) Fungsi run

Proses	<ul style="list-style-type: none"> Menjalankan evolusi untuk sejumlah iterasi yang ditentukan Pada setiap iterasi: <ol style="list-style-type: none"> Menghitung nilai objektif setiap individu Mencatat statistik (nilai maksimum dan rata-rata) Melakukan seleksi Melakukan crossover dan mutasi
Output	Solusi terbaik yang ditemukan, nilai objektif akhir, waktu komputasi, dan grafik konvergensi

```

def run(self):
    start_time = time.time()
    initial_cube = self.population[0].cube.copy()
    for iteration in range(self.max_iterations):
        fitness_values = [cube.fitness() for cube in
self.population]
        max_fitness = max(fitness_values)
        avg_fitness = sum(fitness_values) / len(fitness_values)

        self.objective_log.append((iteration, max_fitness,
avg_fitness))
        self.population = self.selection(fitness_values)
        self.population = self.crossover_and_mutate(self.population)

        end_time = time.time()
        final_cube = self.population[0].cube
        final_objective_value =
self.population[0].calculate_objective_value()[0]

```

```

        print("Initial Cube State:\n", initial_cube)
        print("Final Cube State:\n", final_cube)
        print("Final Objective Value:", final_objective_value)
        print("Population Size:", self.population_size)
        print("Iterations:", self.max_iterations)
        print("Duration:", end_time - start_time, "seconds")

        self.plot_objective_values()
    
```

4) Fungsi selection

Proses	Seleksi yang digunakan pada genetic algorithm untuk permasalahan ini adalah seleksi berbasis peringkat . Seleksi ini dilakukan dengan cara mengurutkan populasi berdasarkan nilai fitness (semakin kecil semakin baik) dan memilih setengah populasi terbaik.
Output	Subpopulasi terbaik yang akan menjadi parent untuk generasi berikutnya
<pre>def selection(self, fitness_values): selected_population = sorted(self.population, key=lambda cube: cube.fitness(), reverse=True) return selected_population[:self.population_size // 2]</pre>	

5) Fungsi crossover

Proses	Membuat salinan dari parent1 dan untuk setiap posisi dalam kubus: <ul style="list-style-type: none"> • Dengan probabilitas 0.5, mengambil nilai dari parent2 • Dengan probabilitas 0.5, mempertahankan nilai dari parent1
Output	Offspring baru yang mewarisi sifat dari kedua parent

```

def crossover(self, parent1, parent2):
    child_cube = np.copy(parent1.cube)
    for i in range(self.cube_size):
        for j in range(self.cube_size):
            for k in range(self.cube_size):
                if random.random() < 0.5:
                    child_cube[i, j, k] = parent2.cube[i, j, k]
    return MagicCube(self.cube_size, child_cube)

```

6) Fungsi mutate

Proses	Memilih dua posisi acak dalam kubus dan menukar nilai pada kedua posisi tersebut
Output	Kubus yang telah mengalami mutasi
<pre> def mutate(self, cube): i1, j1, k1 = random.randint(0, self.cube_size - 1), random.randint(0, self.cube_size - 1), random.randint(0, self.cube_size - 1) i2, j2, k2 = random.randint(0, self.cube_size - 1), random.randint(0, self.cube_size - 1), random.randint(0, self.cube_size - 1) cube.swap(i1, j1, k1, i2, j2, k2) </pre>	

7) Fungsi crossover_and_mutate

Proses	Untuk setiap individu baru yang akan dibuat: <ol style="list-style-type: none"> 1. Memilih dua parent secara acak 2. Melakukan crossover untuk menghasilkan child 3. Melakukan mutasi pada child 4. Menambahkan child ke populasi baru
Output	Populasi baru hasil crossover dan mutasi
<pre> def crossover_and_mutate(self, selected_population): new_population = [] </pre>	

```

        for _ in range(self.population_size):
            parent1, parent2 = random.sample(selected_population, 2)
            child = self.crossover(parent1, parent2)
            self.mutate(child)
            new_population.append(child)
        return new_population
    
```

8) Fungsi plot_objective_values

Proses	Memisahkan data log menjadi komponen iterasi, nilai maksimum, dan nilai rata-rata serta membuat plot dengan matplotlib
Output	Grafik yang menunjukkan perkembangan nilai objektif maksimum dan rata-rata selama proses evolusi

```

def plot_objective_values(self):
    iterations, max_values, avg_values = zip(*self.objective_log)

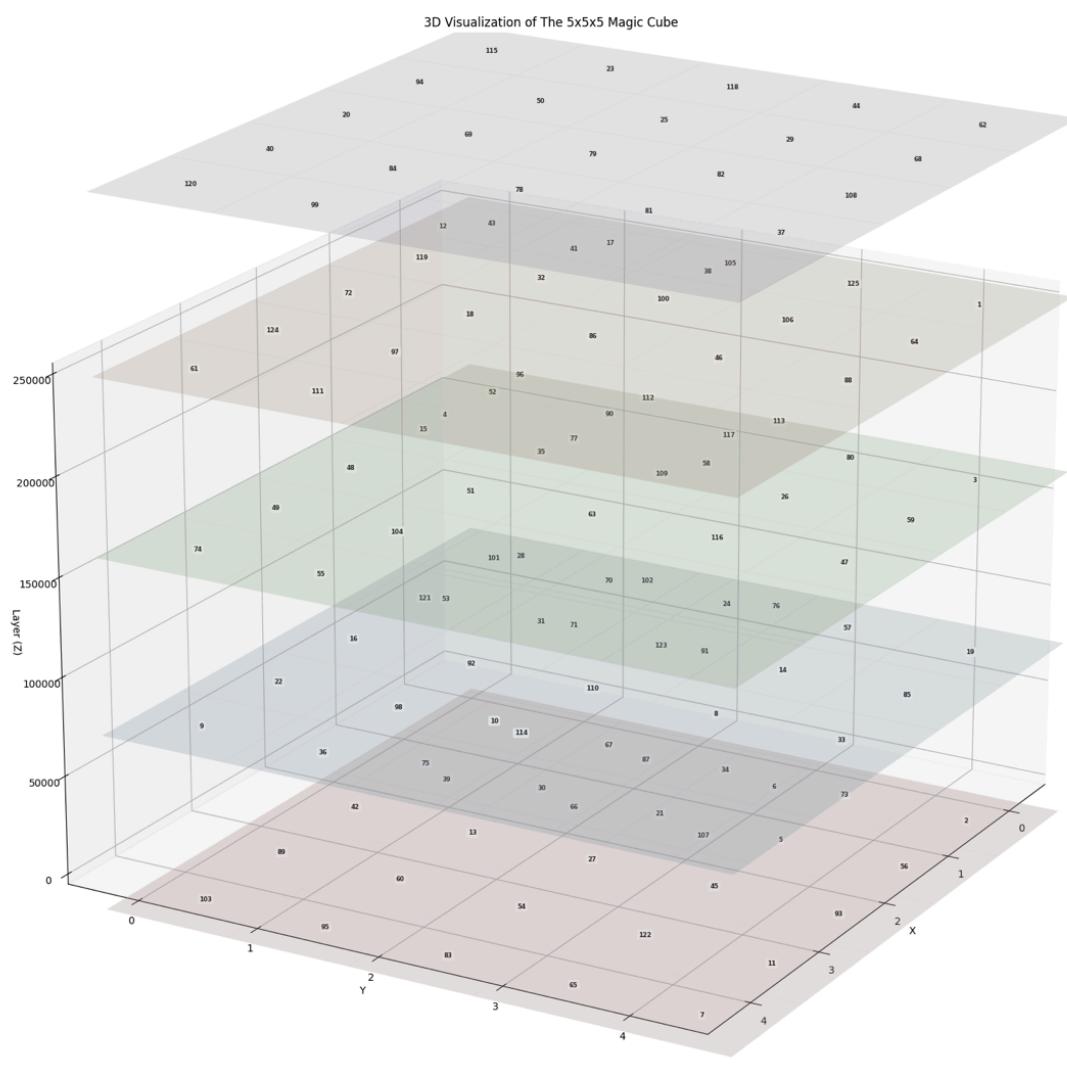
    plt.figure(figsize=(10, 5))
    plt.plot(iterations, max_values, label='Max Fitness Value',
marker='o')
    plt.plot(iterations, avg_values, label='Average Fitness Value',
marker='x')
    plt.title('Fitness Values Over Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Fitness Value')
    plt.legend()
    plt.grid(True)
    plt.show()
    
```

3. Hasil Eksperimen dan Analisis

Hasil Eksperimen

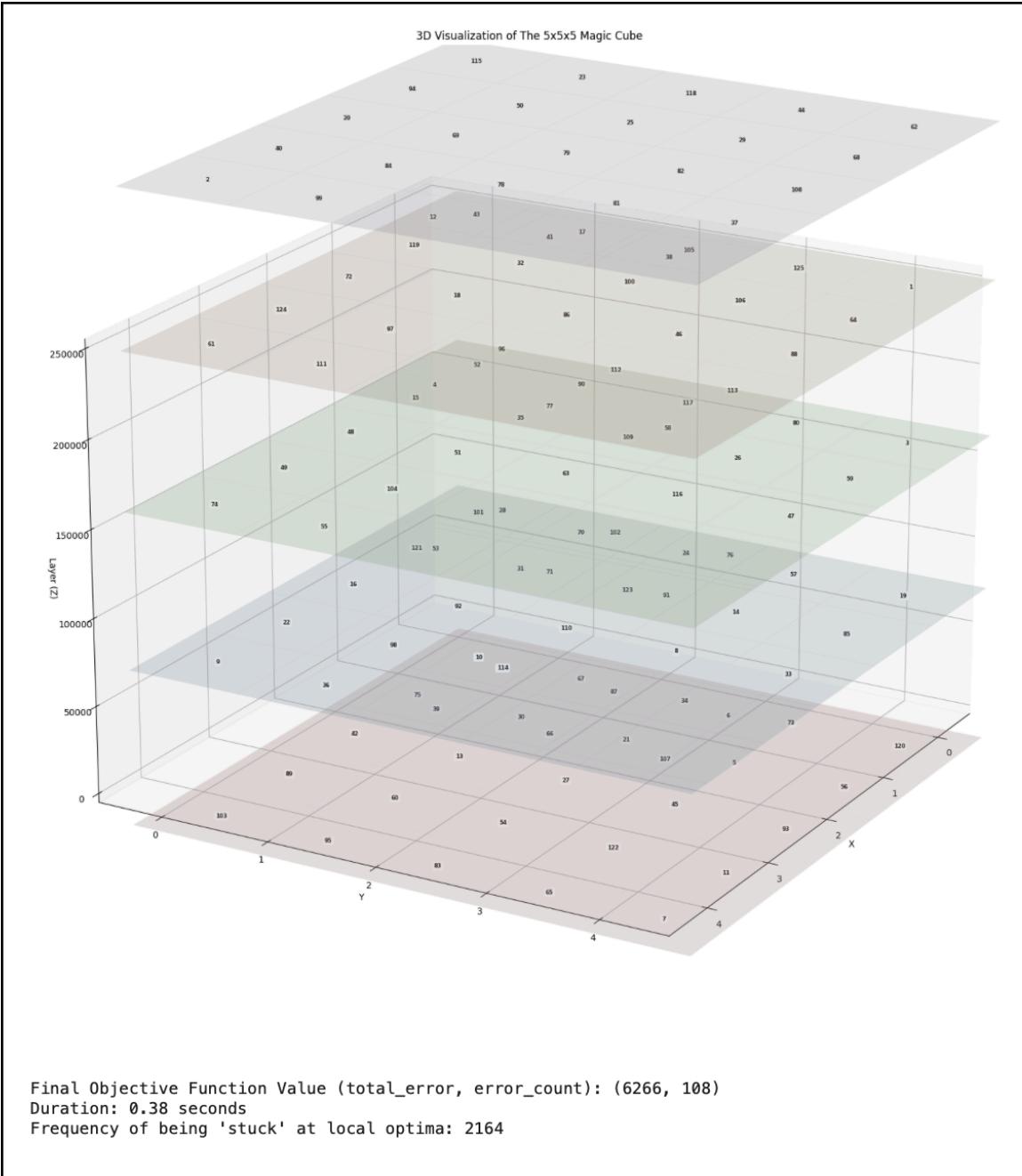
Test Case 1

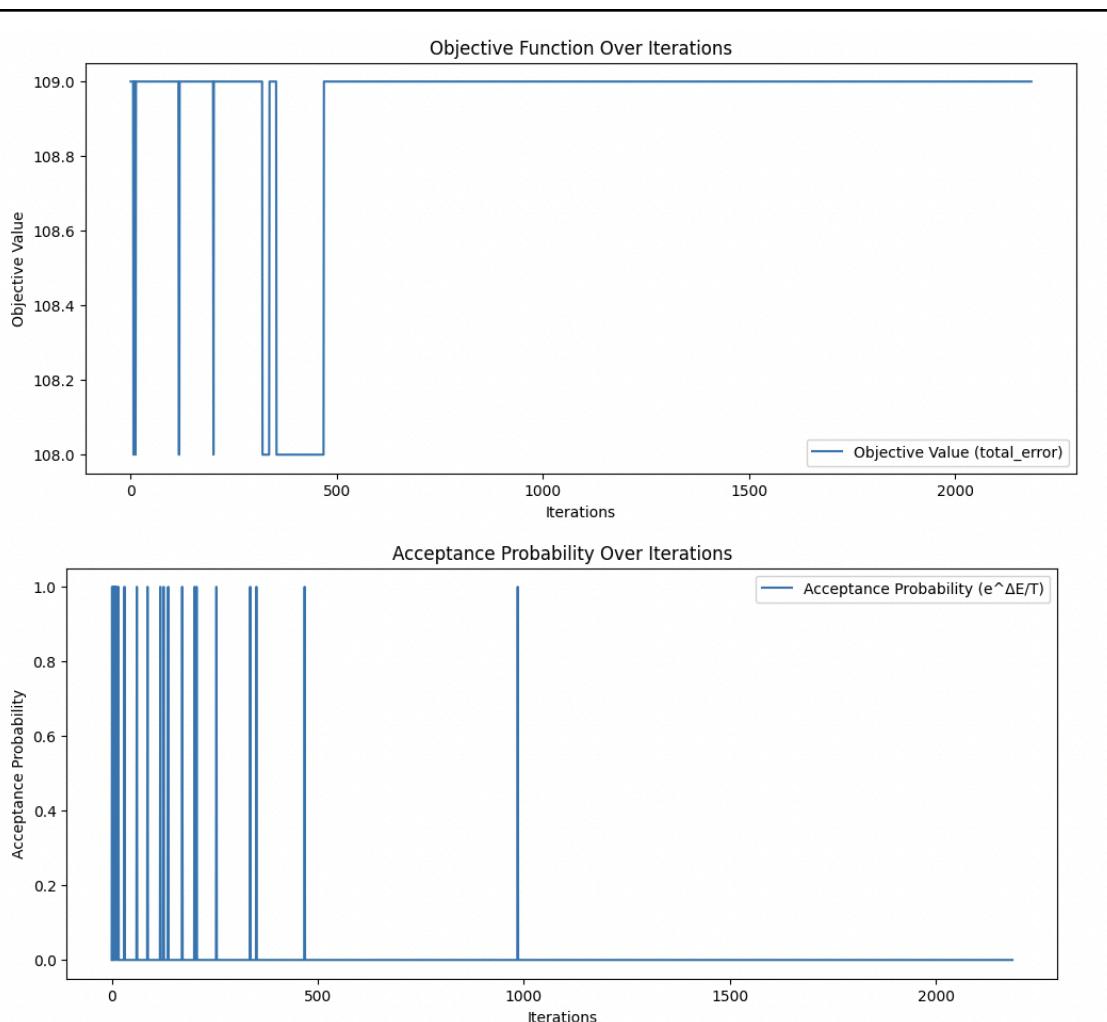
Initial Cube



Initial Objective Function Value (total_error, error_count): (6694, 109)

Hasil Simulated Annealing

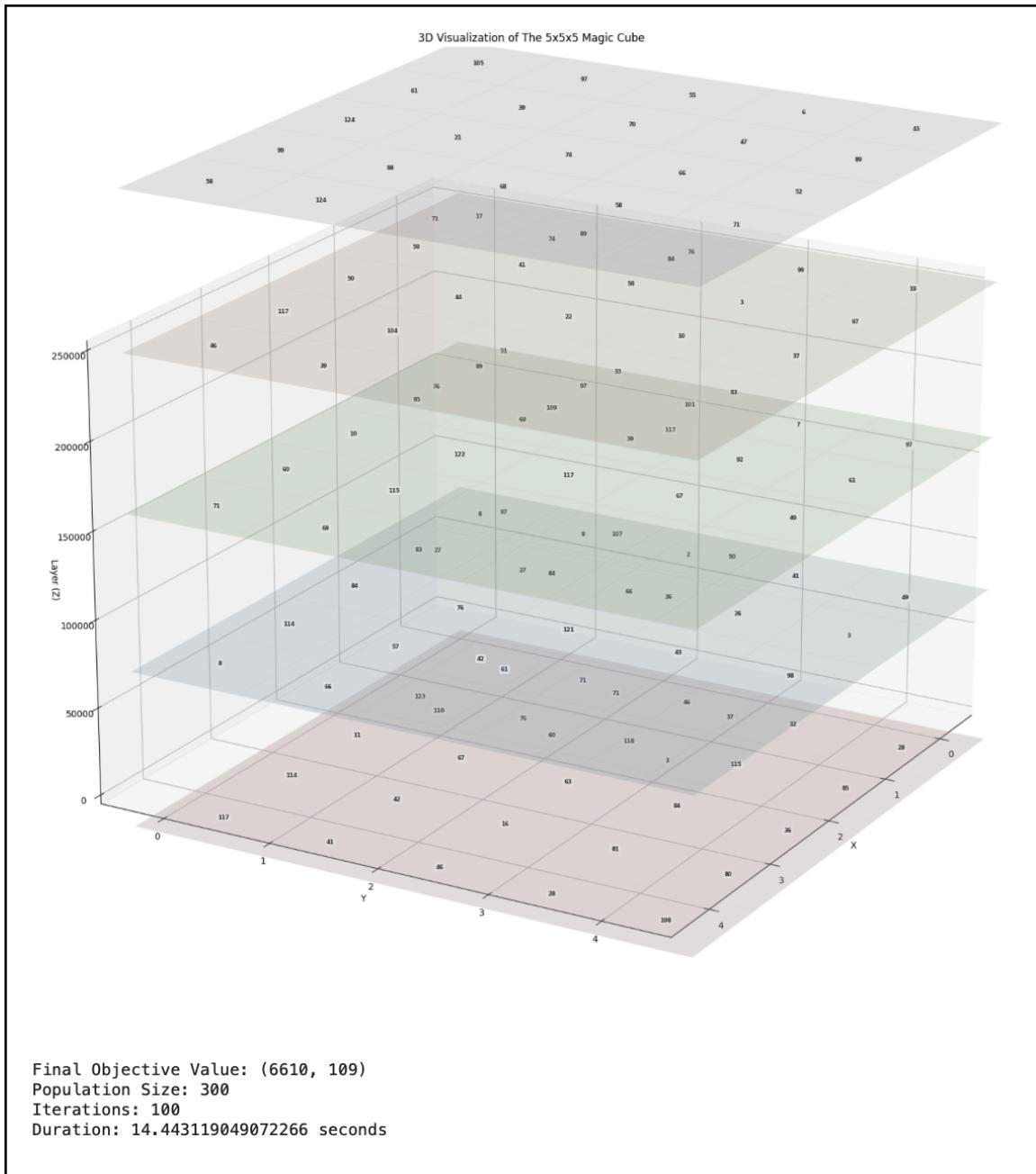


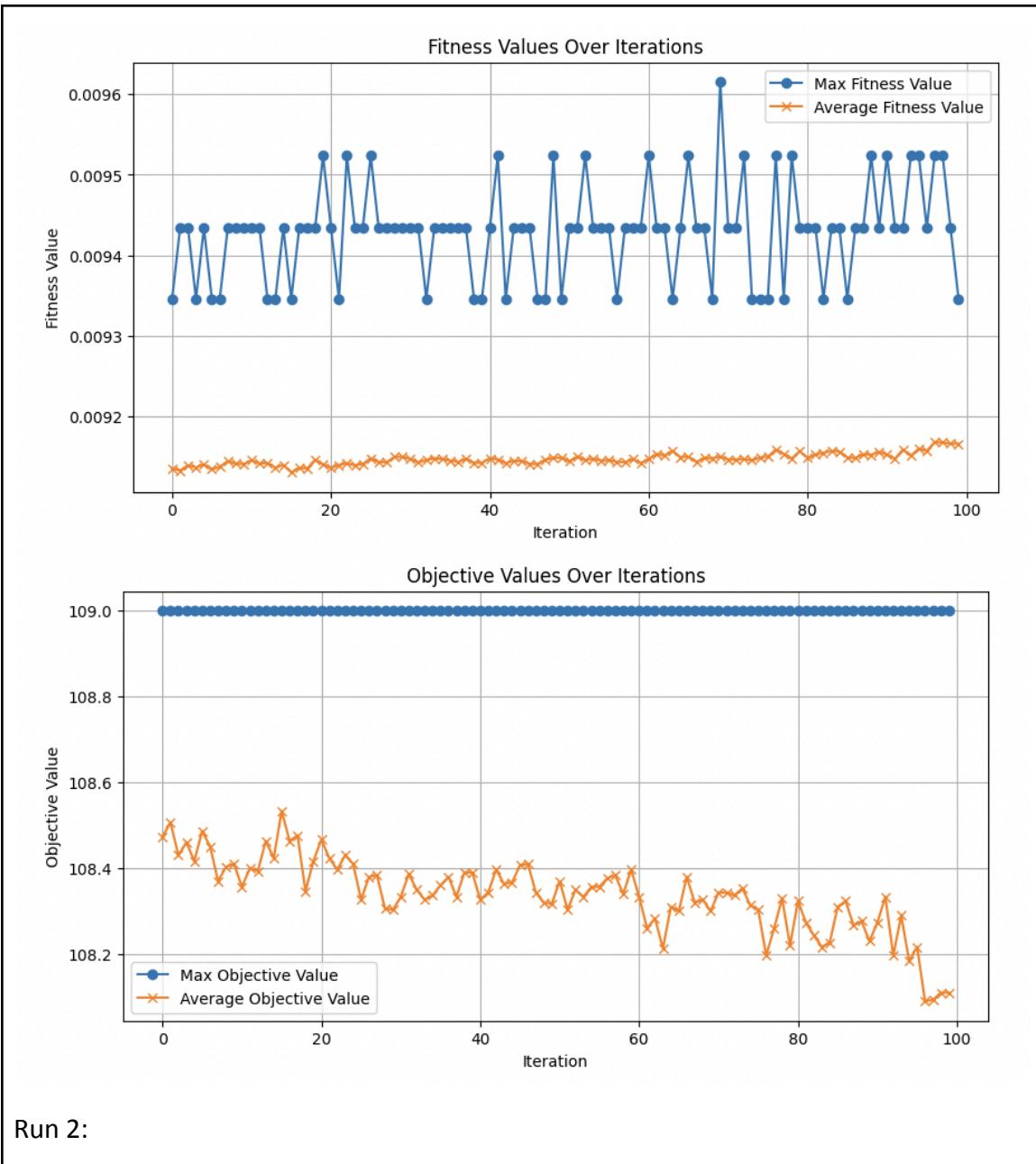


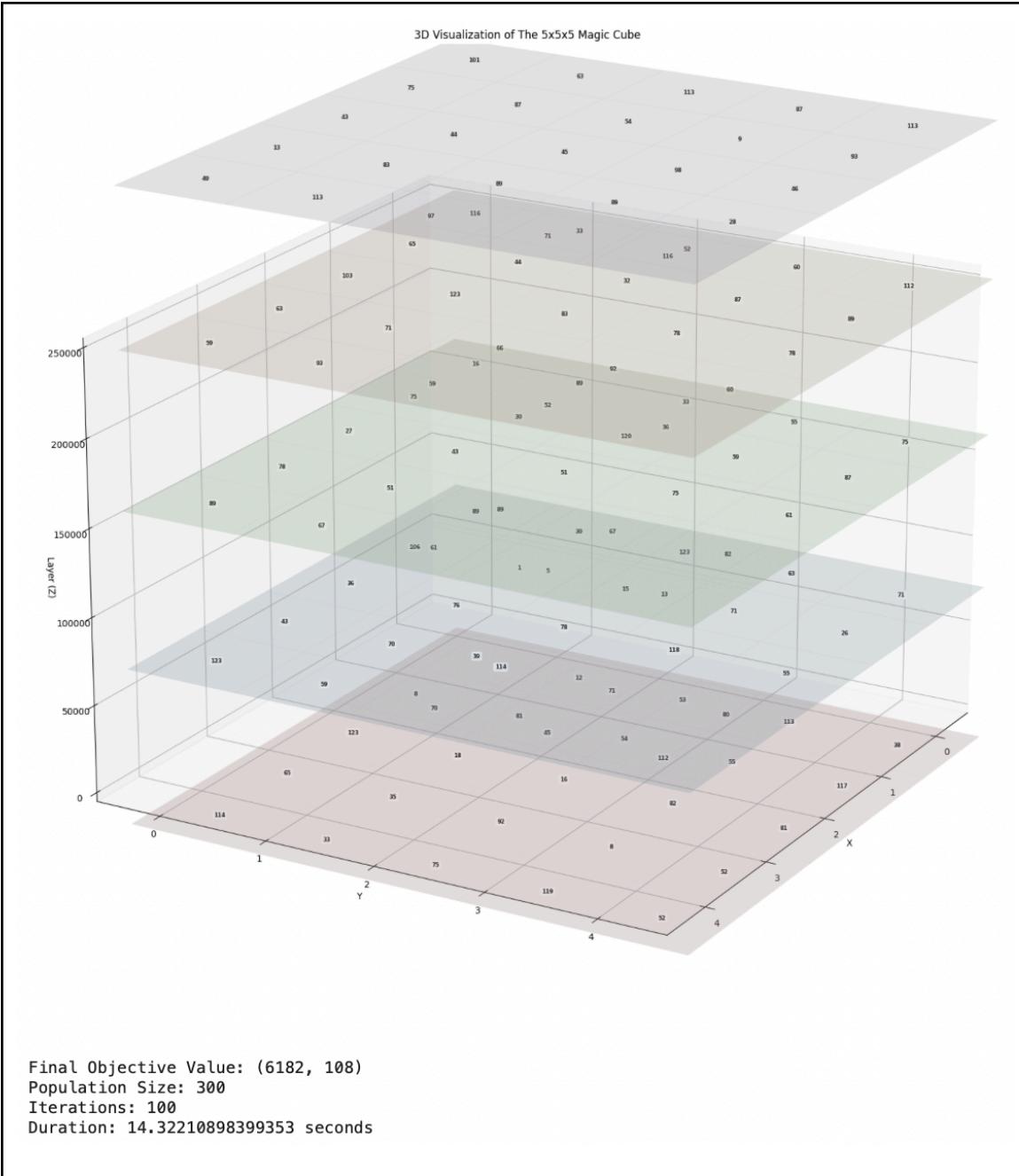
Populasi sebagai kontrol,

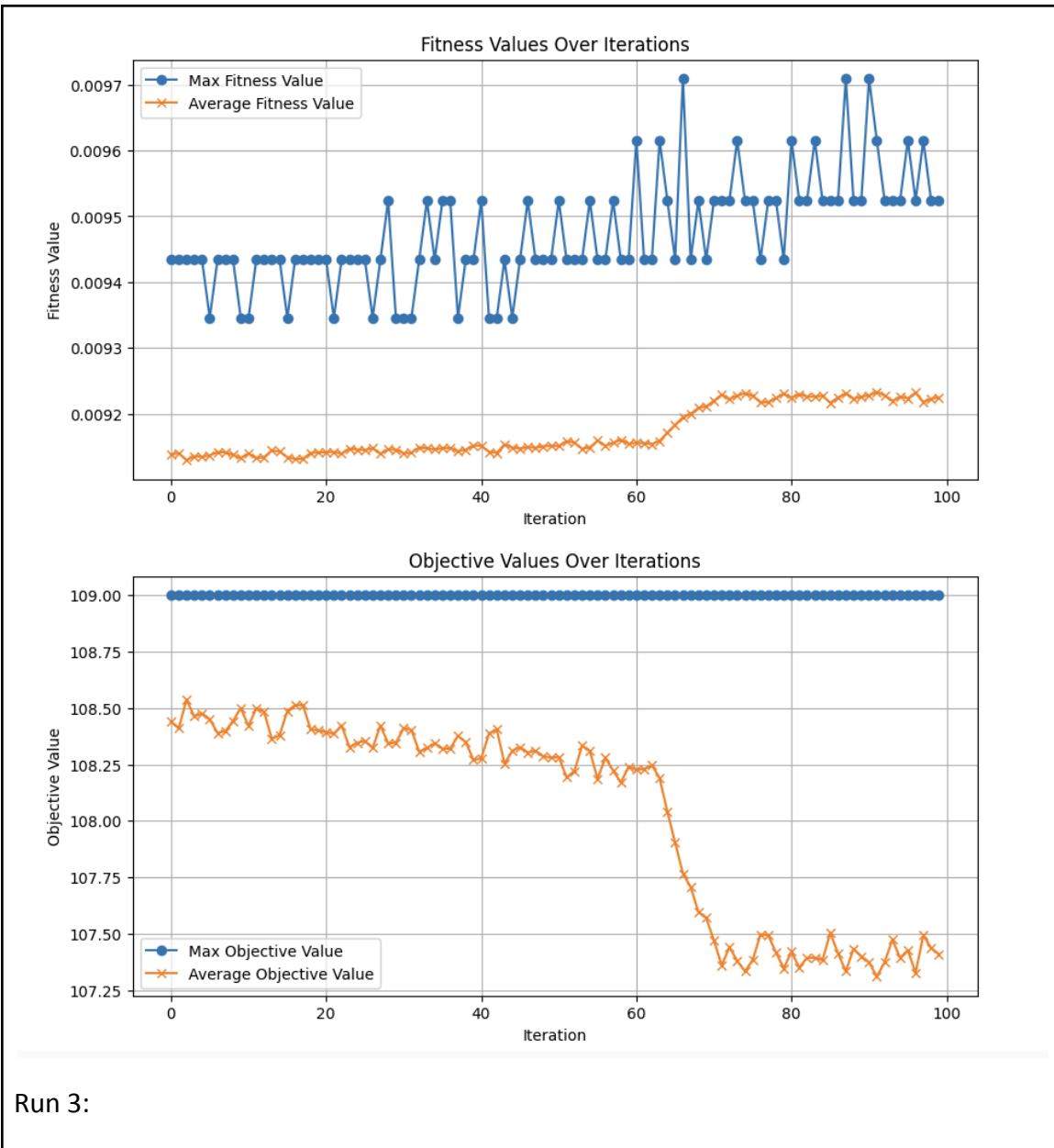
Variasi 1:

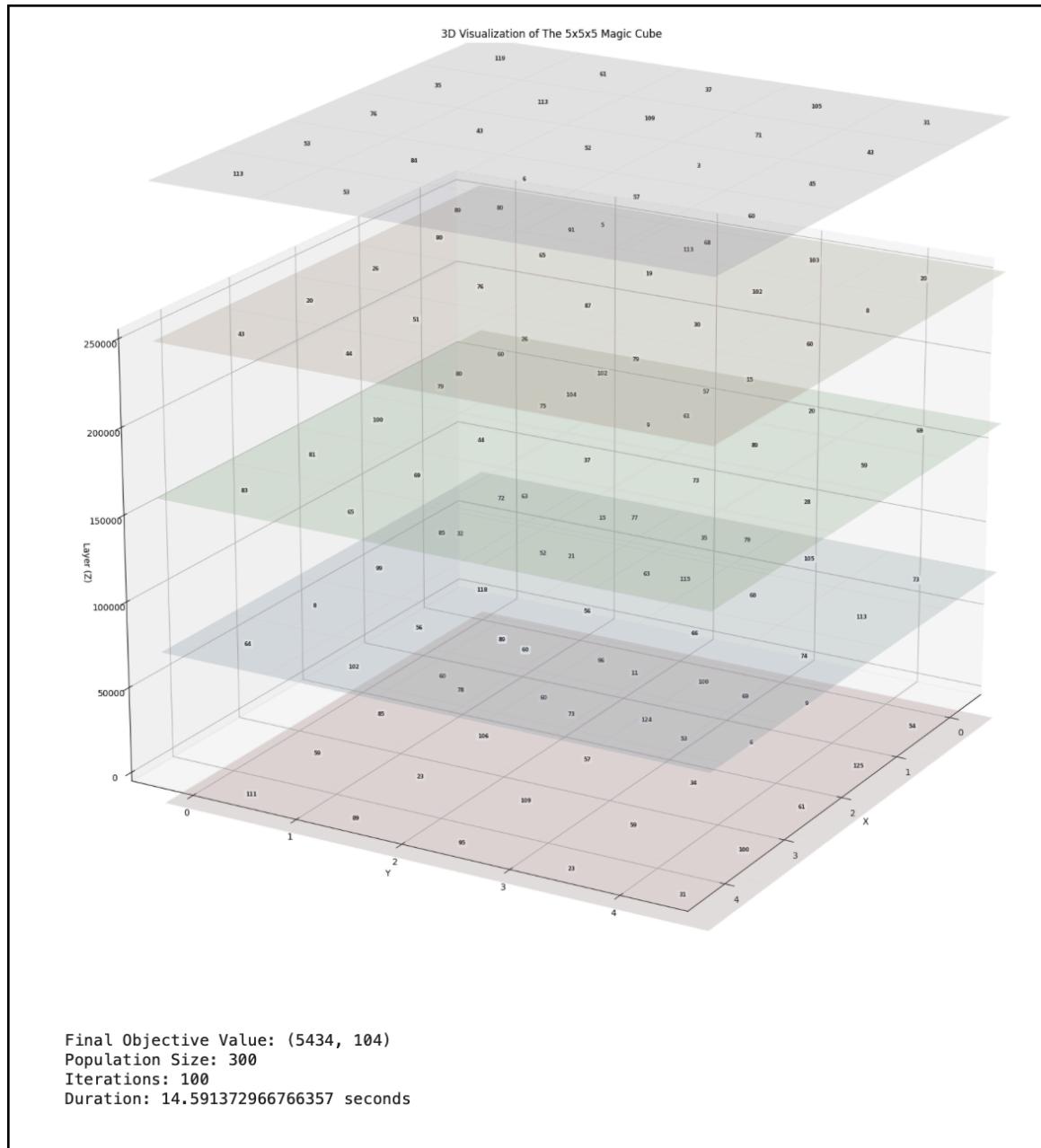
Run 1:

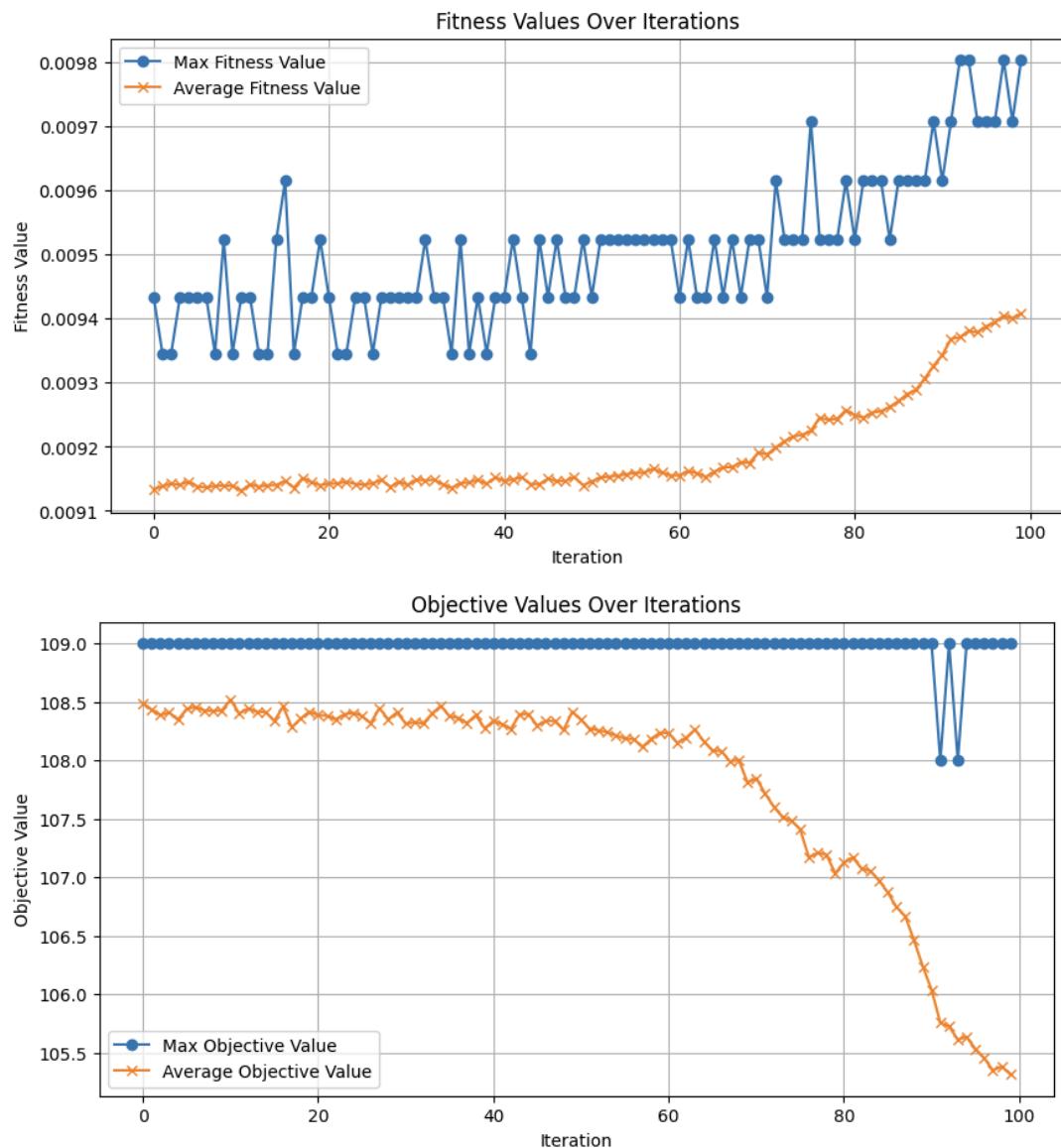






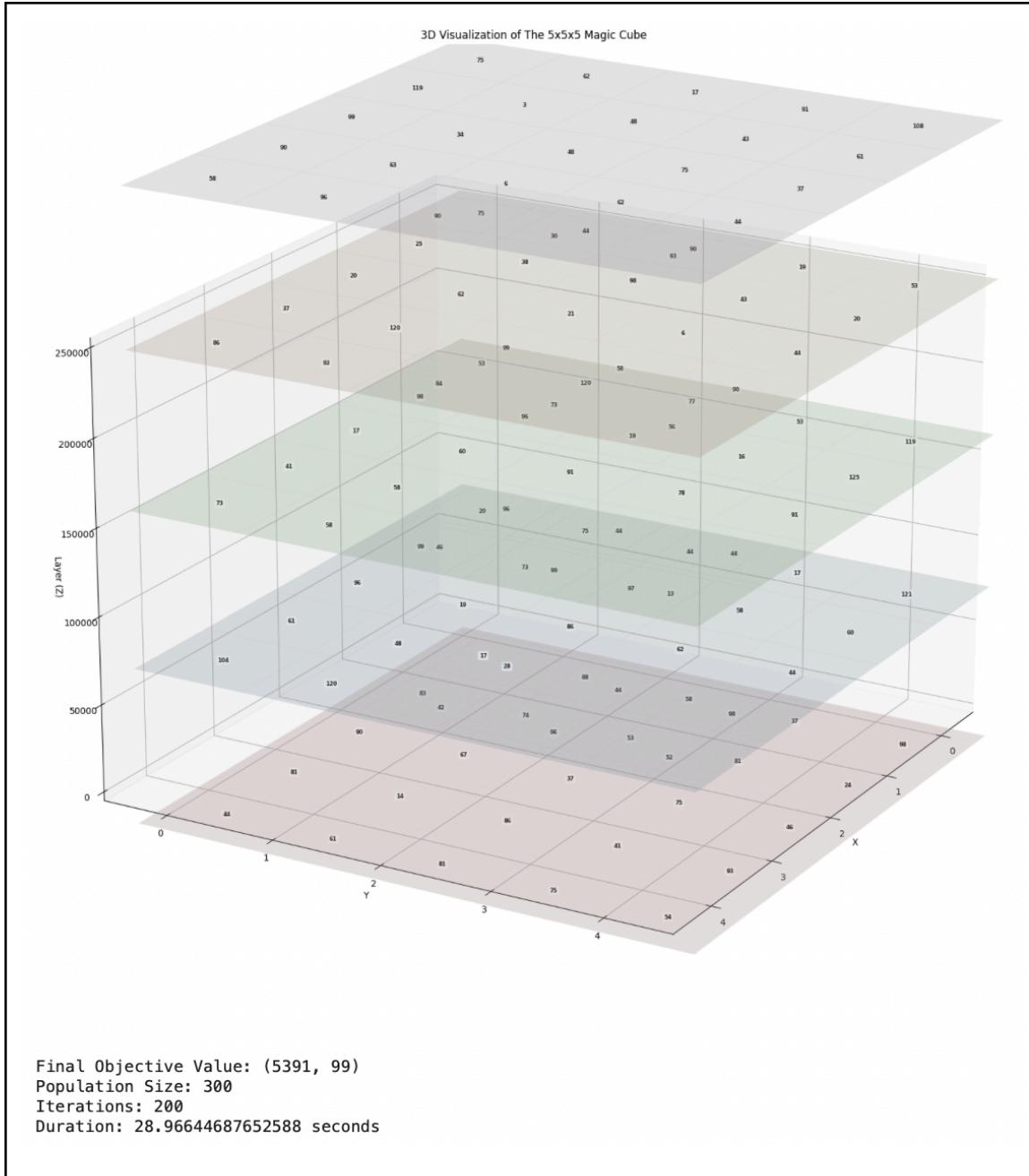


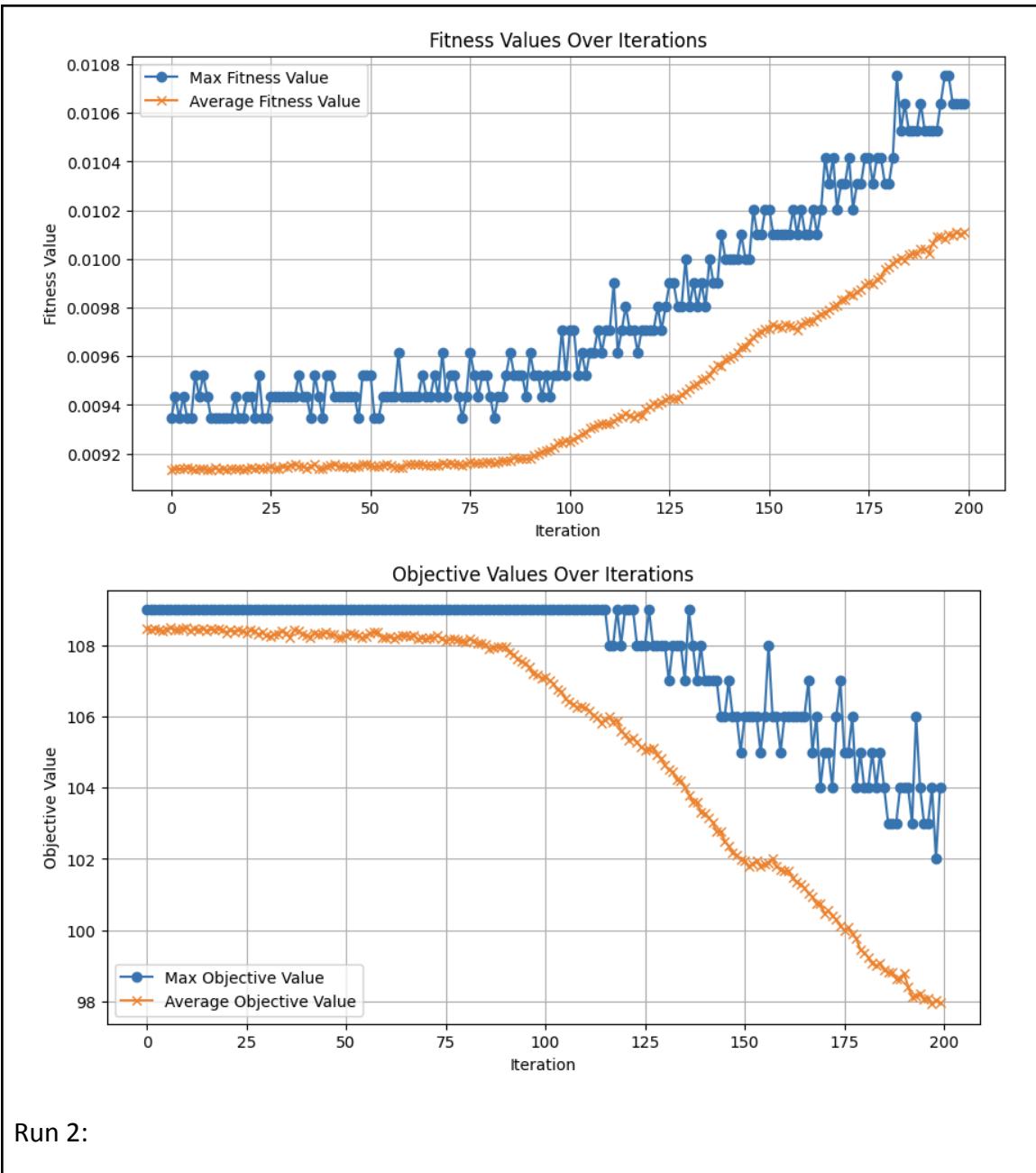


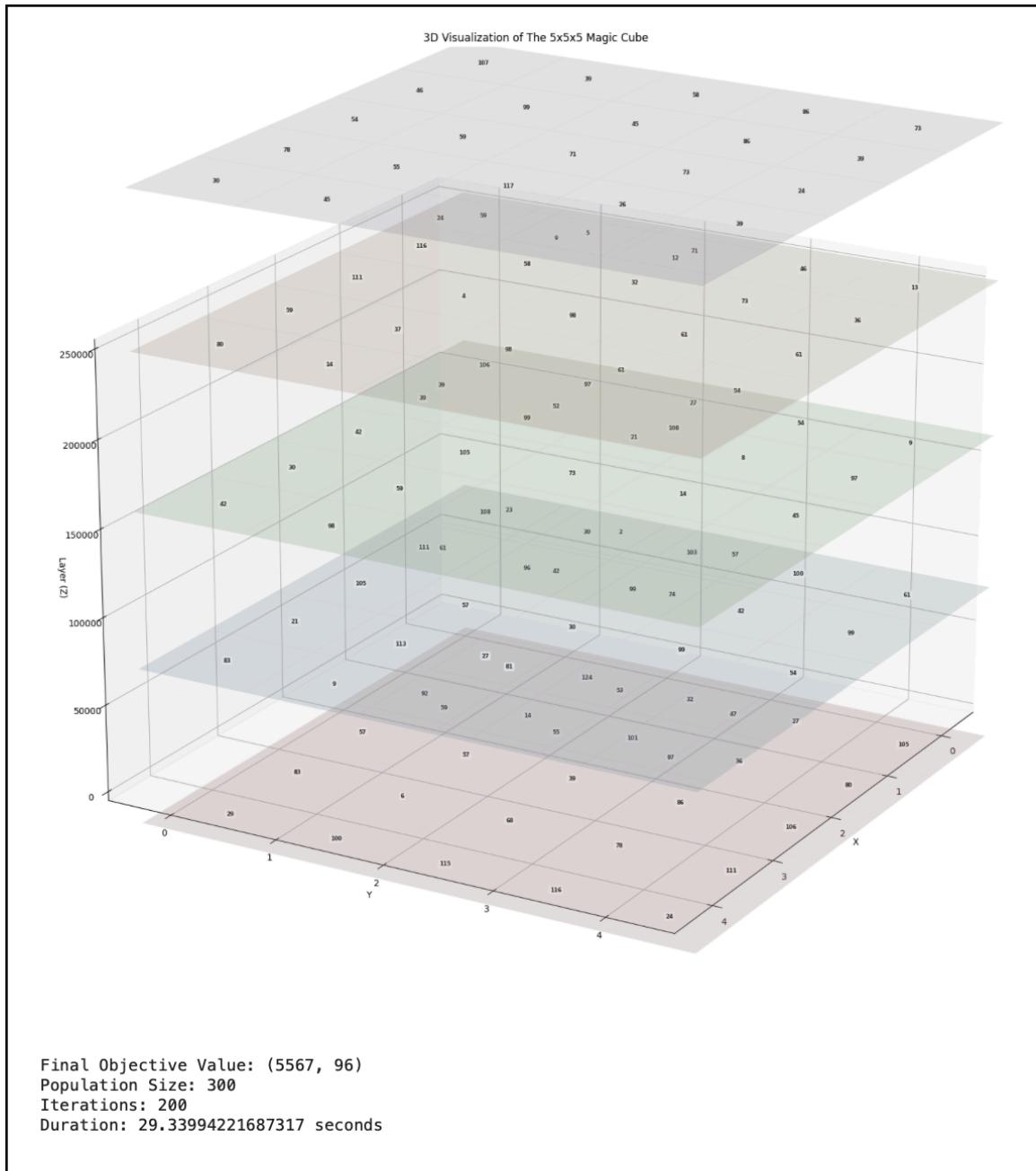


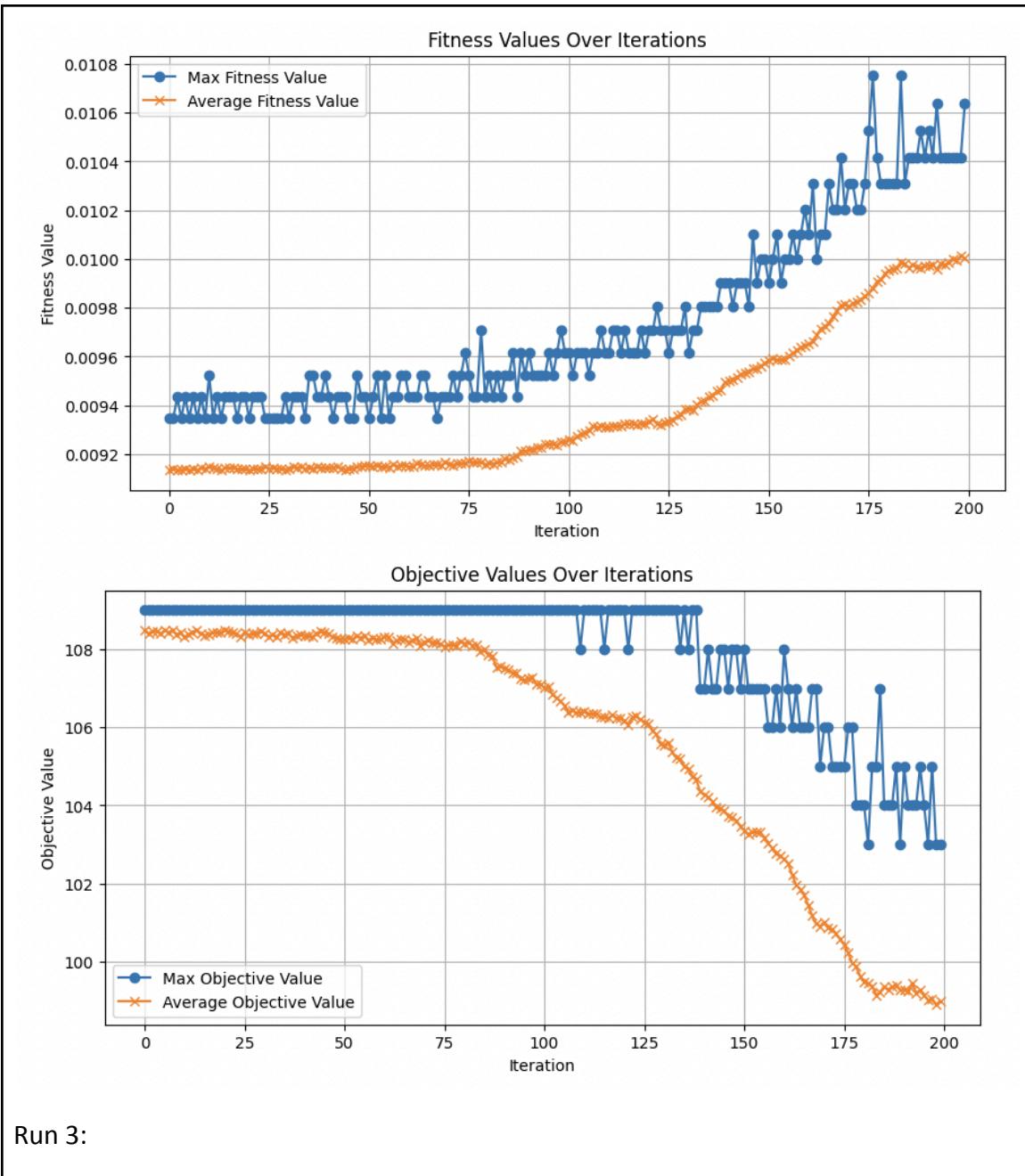
Variasi 2:

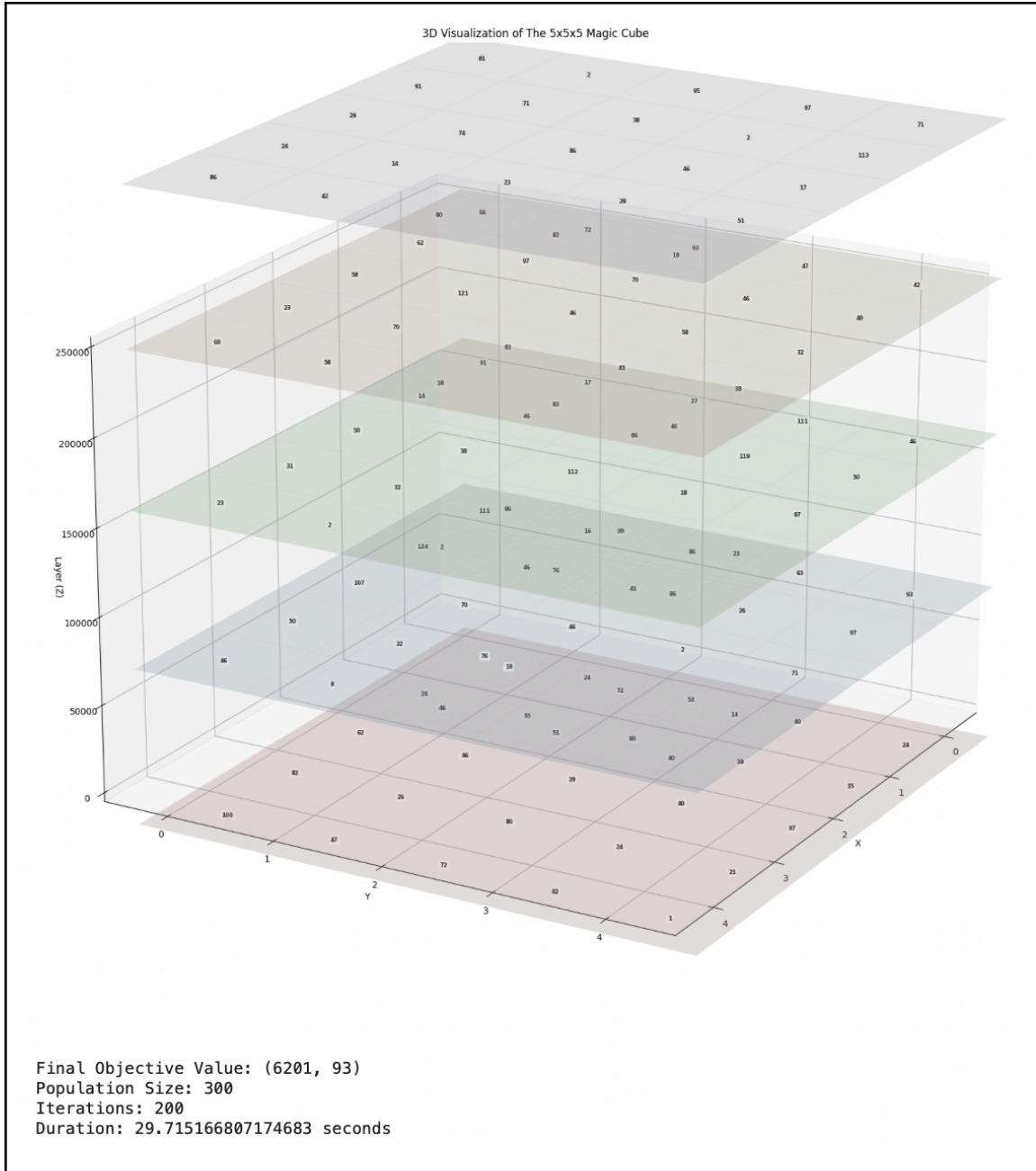
Run 1:

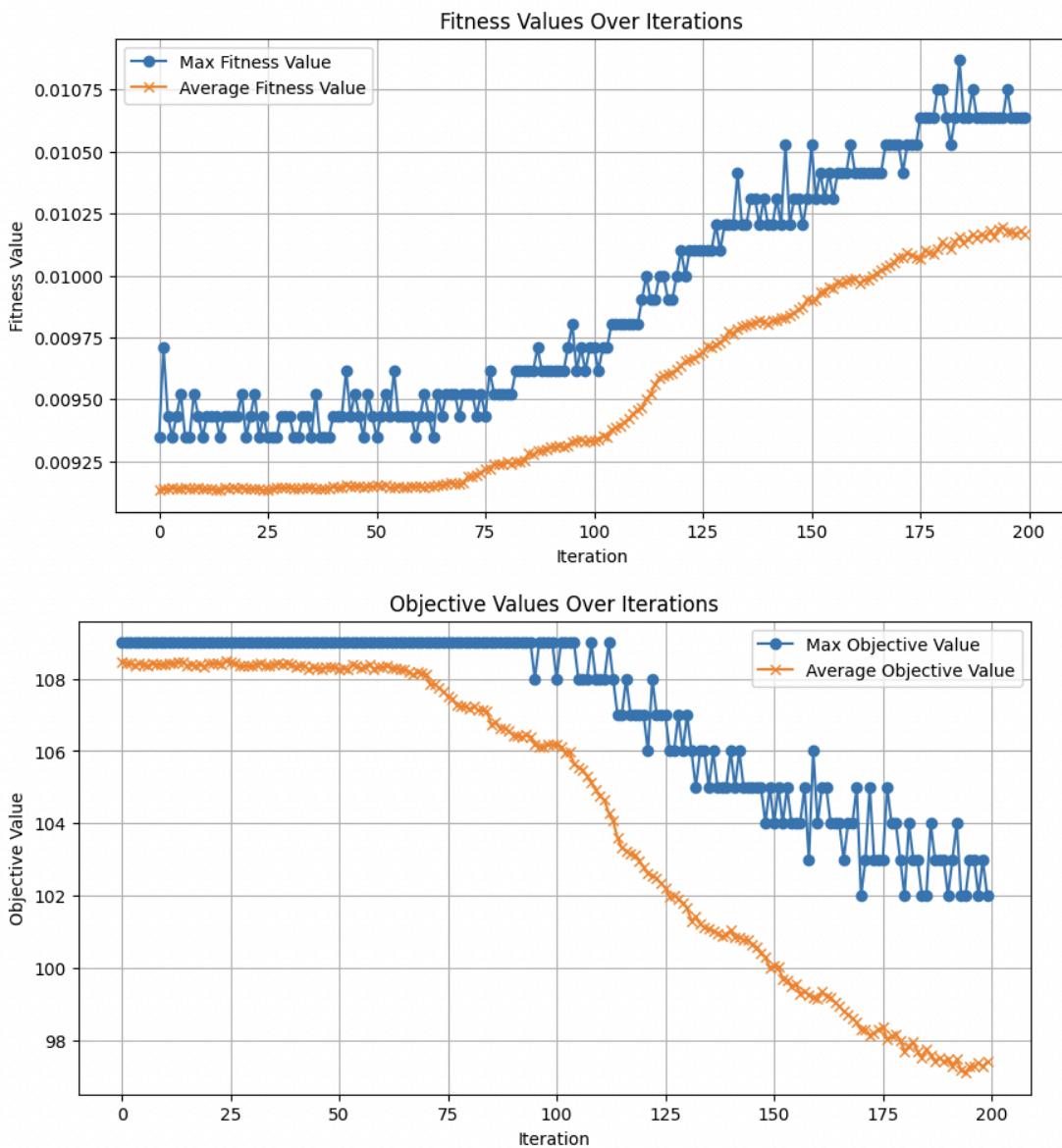






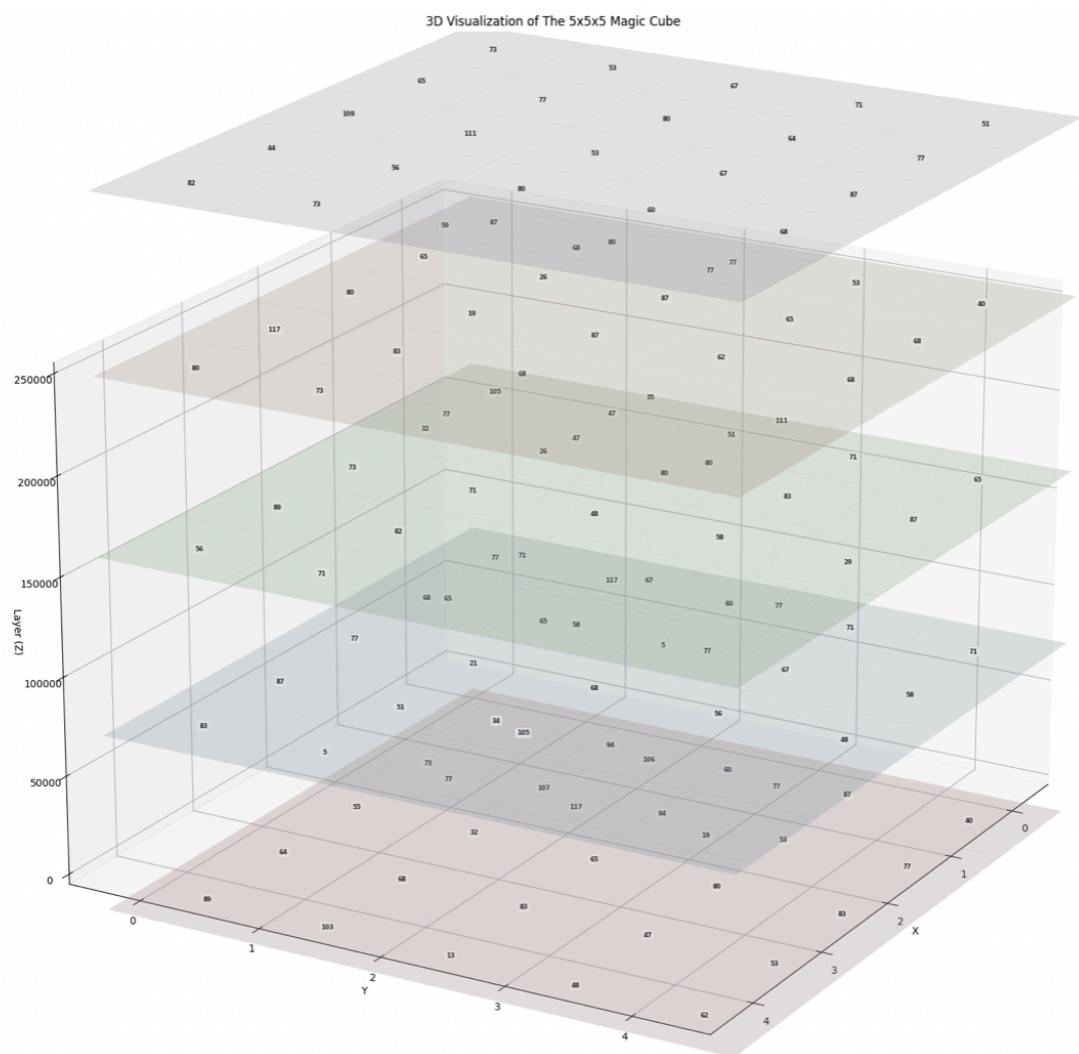




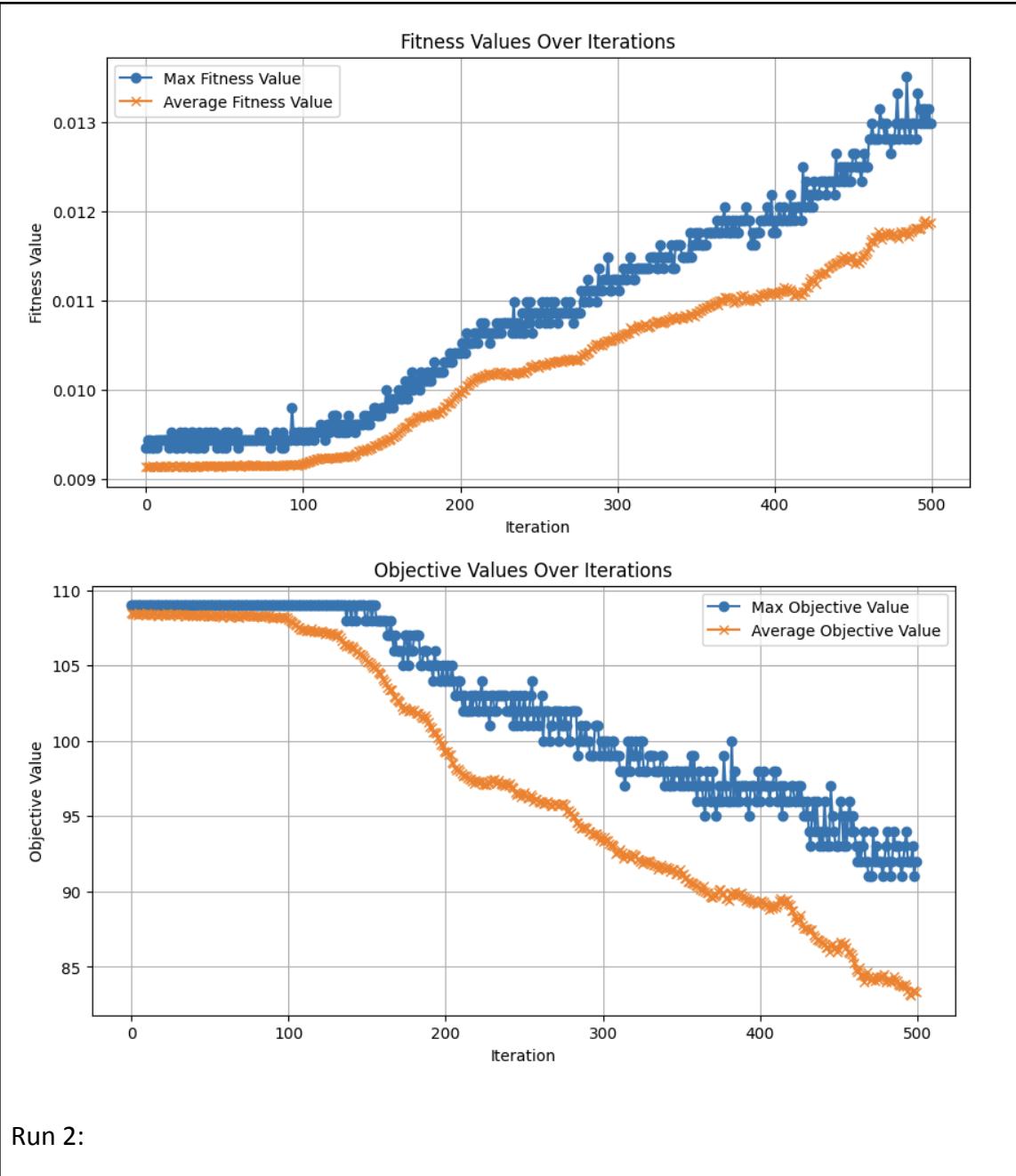


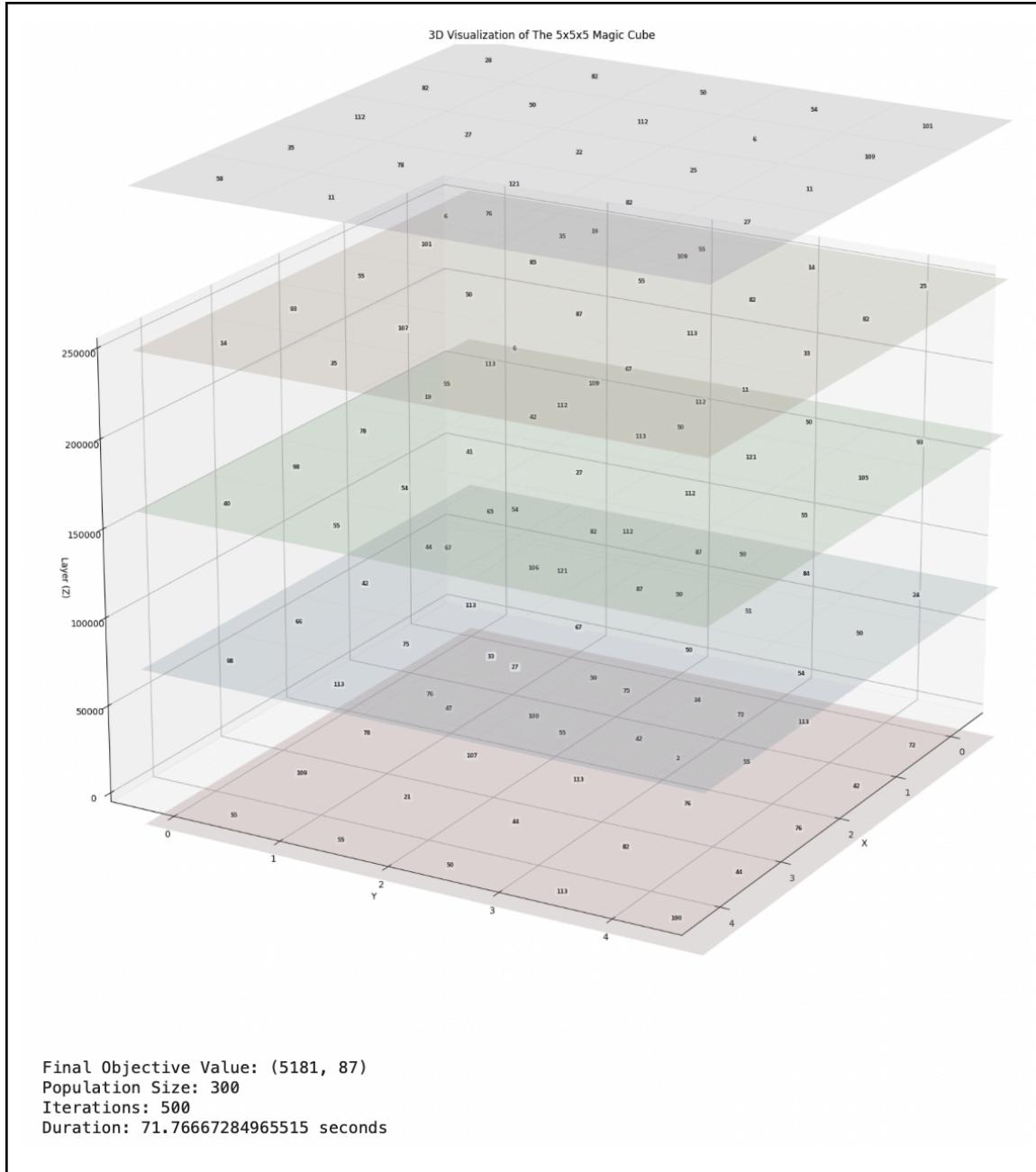
Variasi 3:

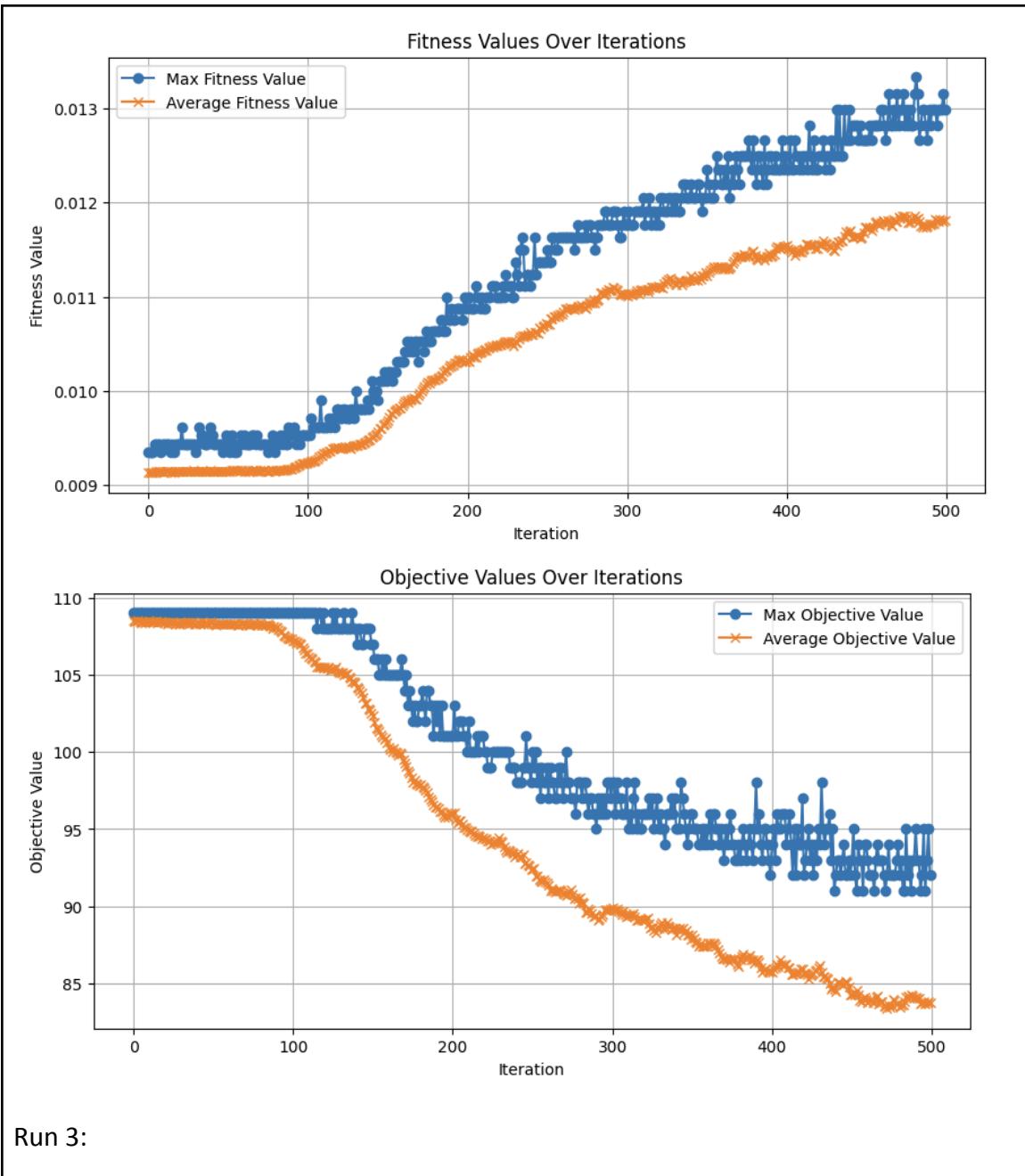
Run 1:

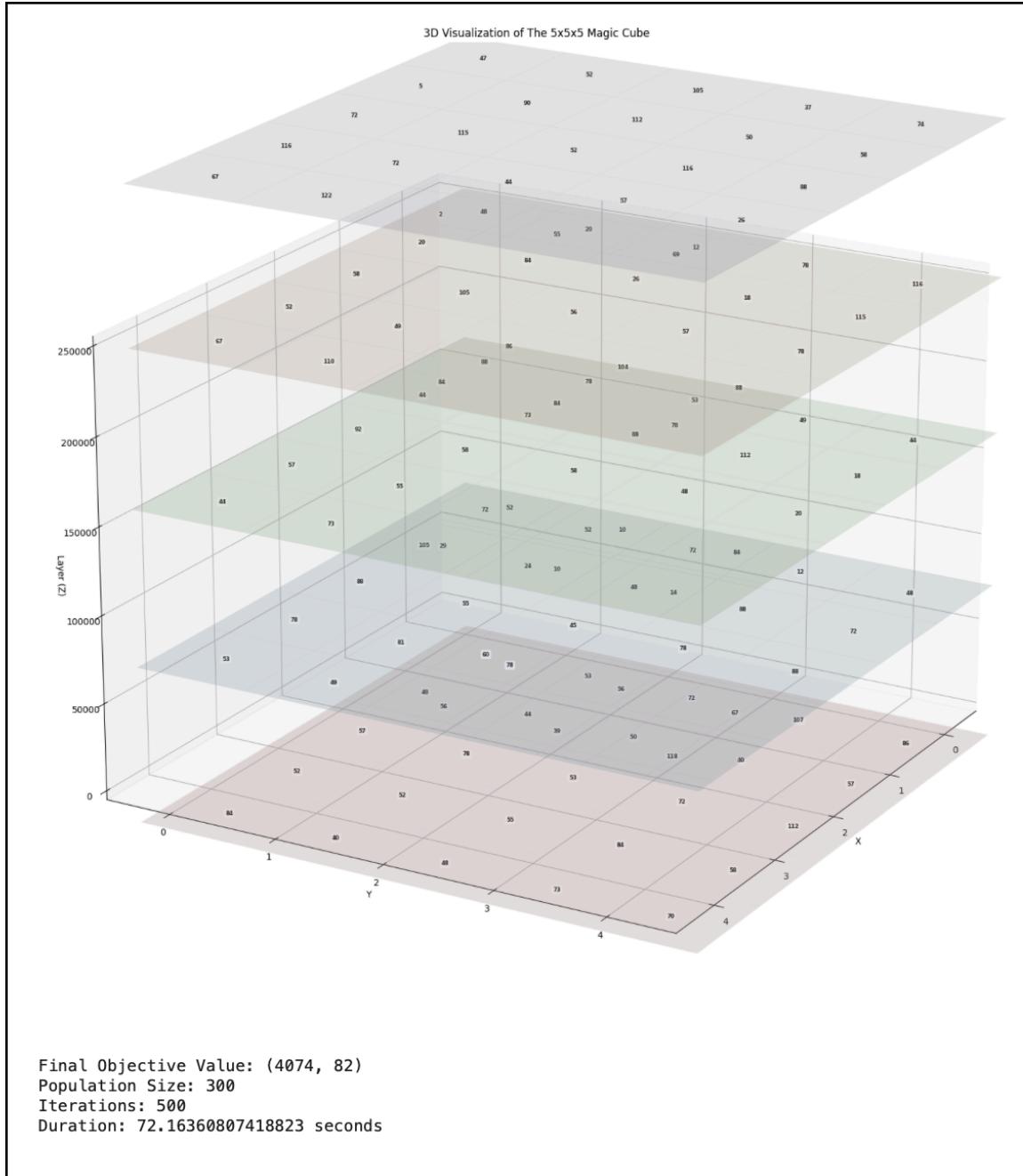


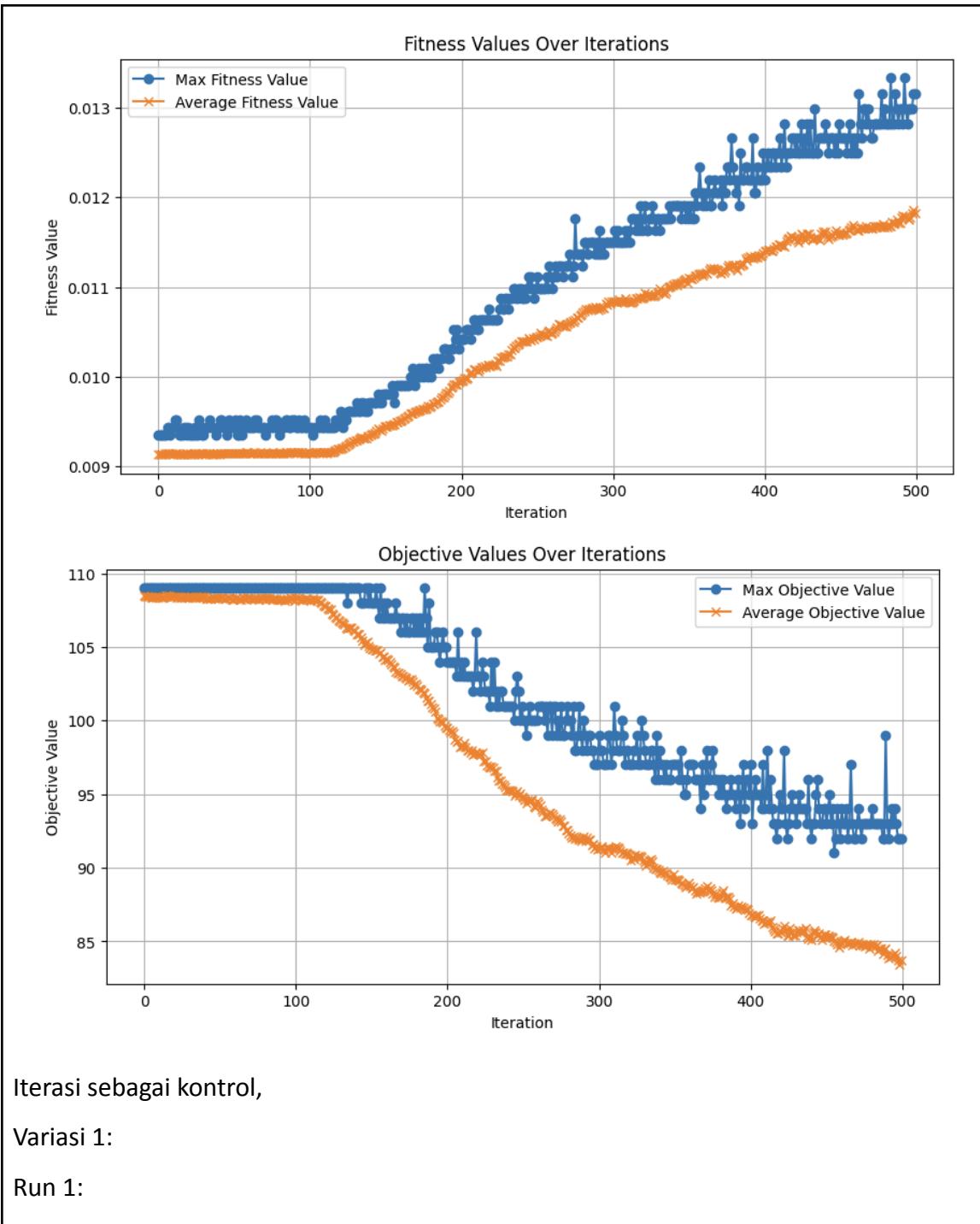
```
Final Objective Value: (3732, 85)
Population Size: 300
Iterations: 500
Duration: 73.61034393310547 seconds
```

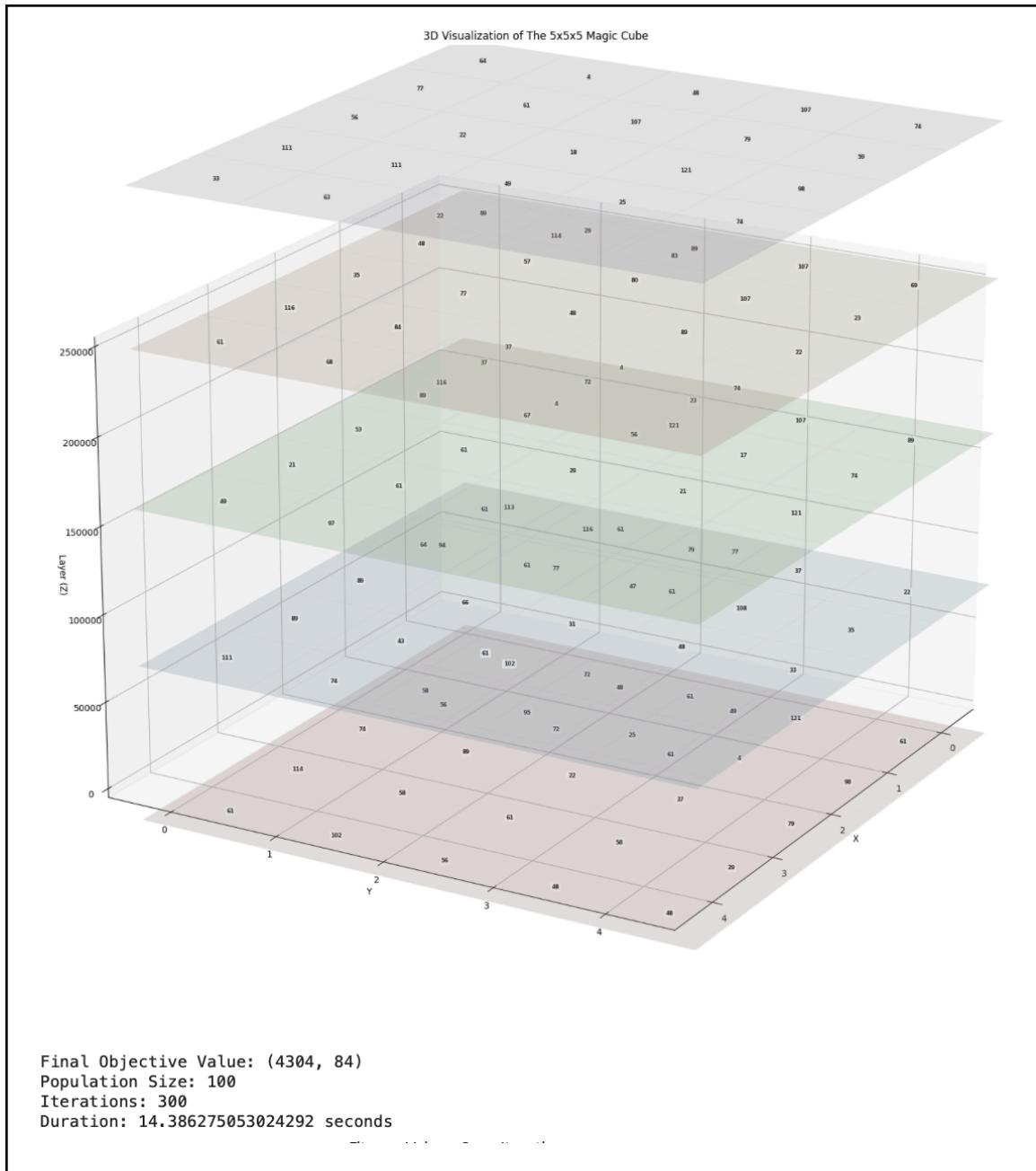


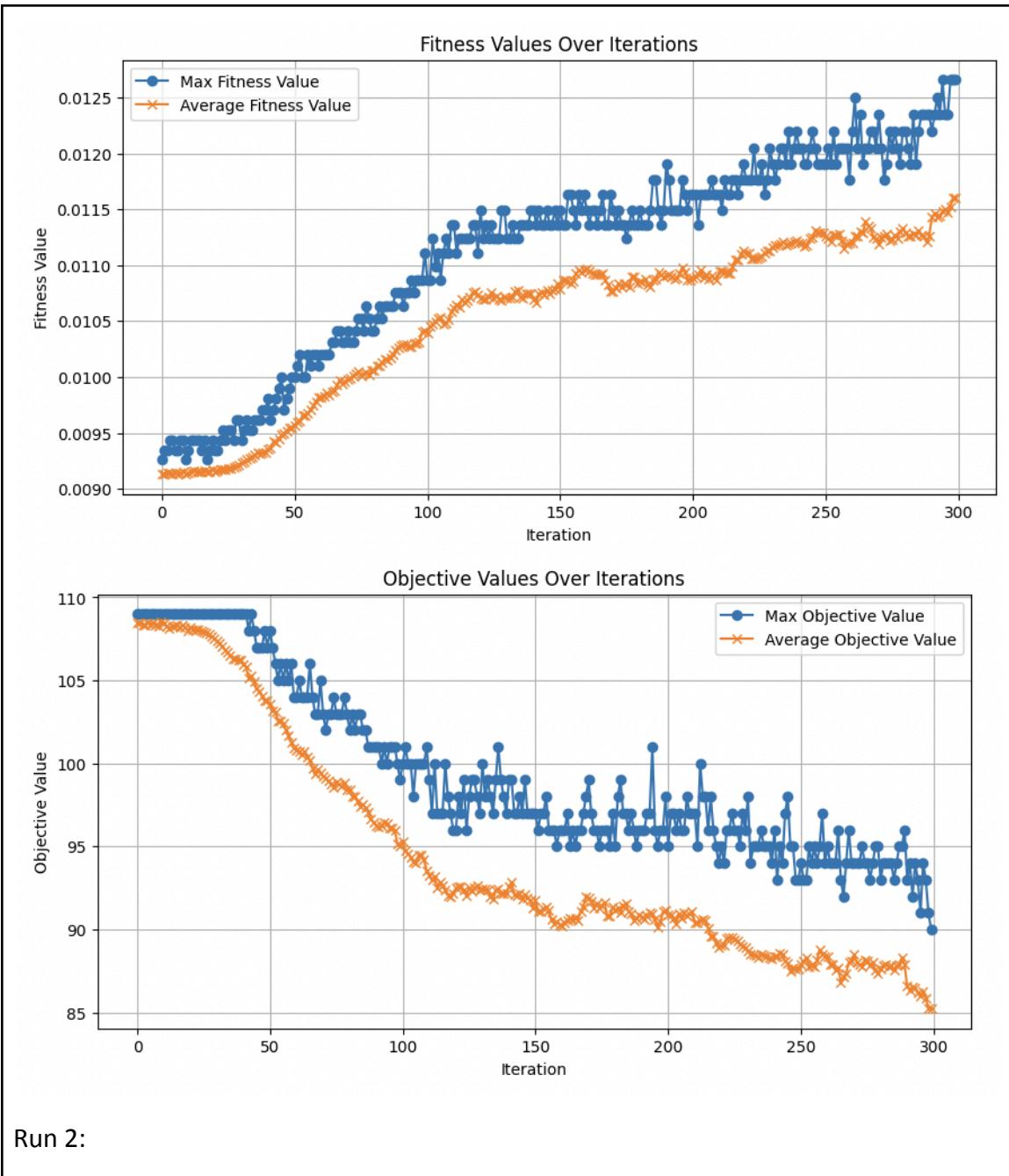


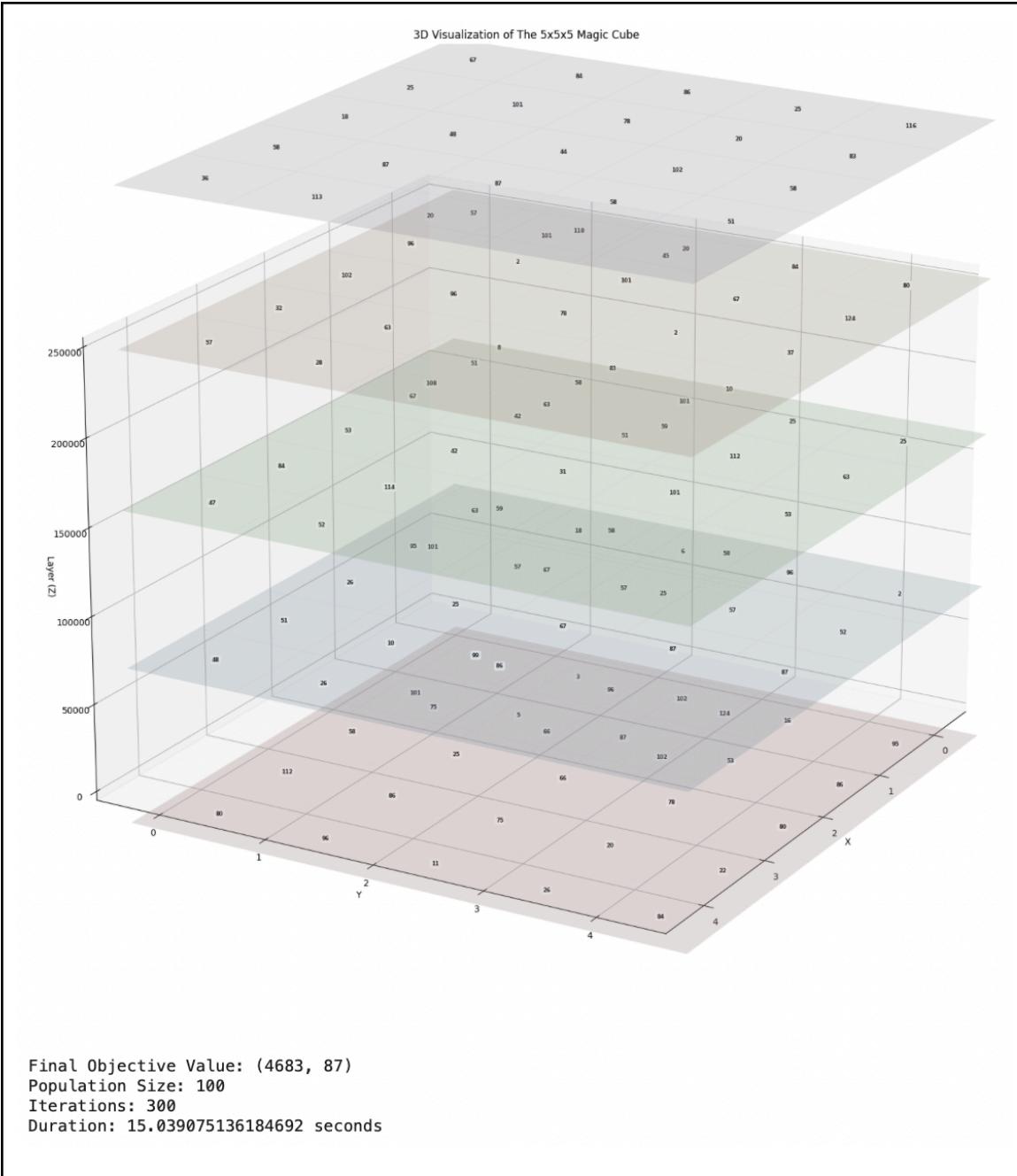


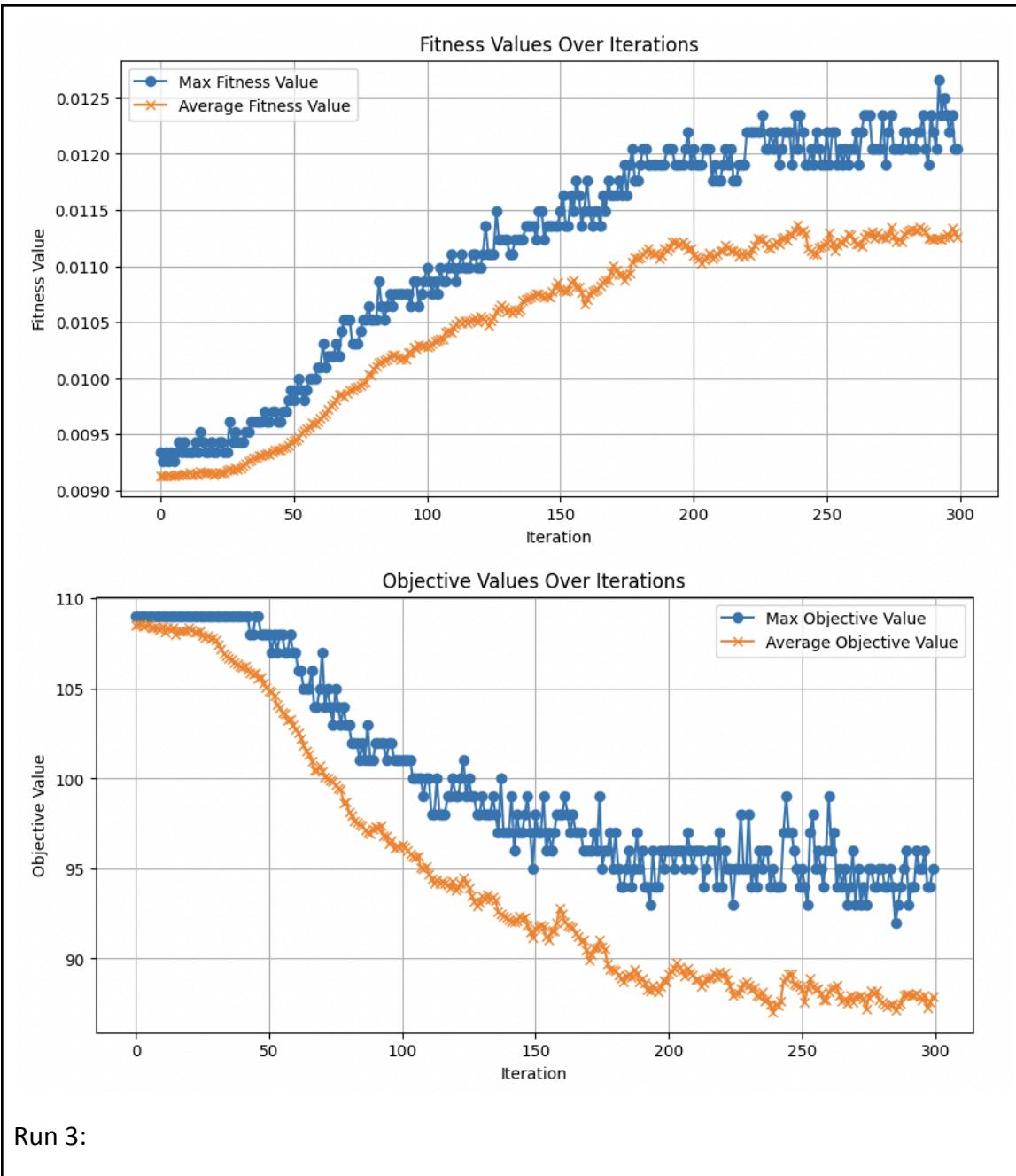


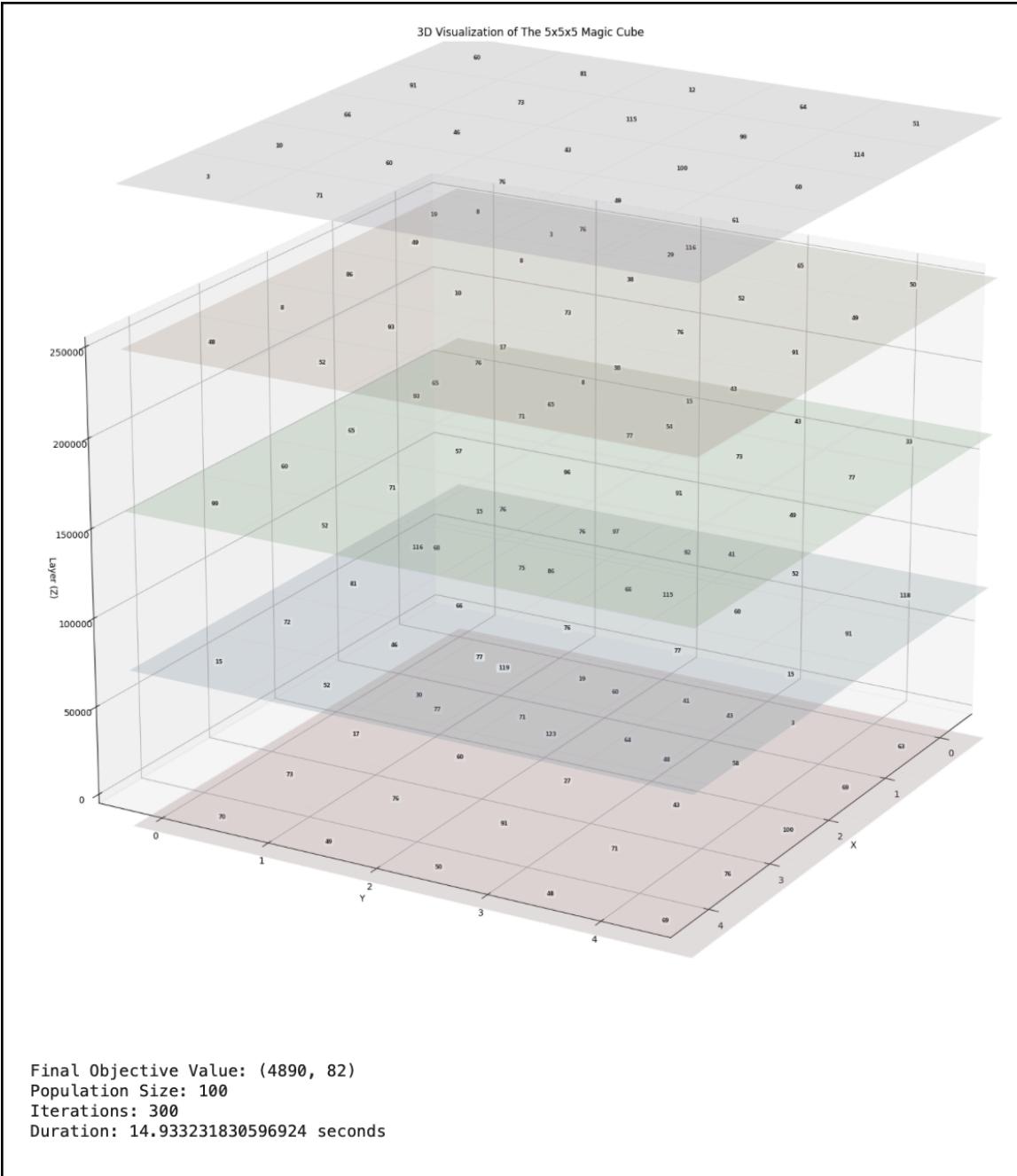


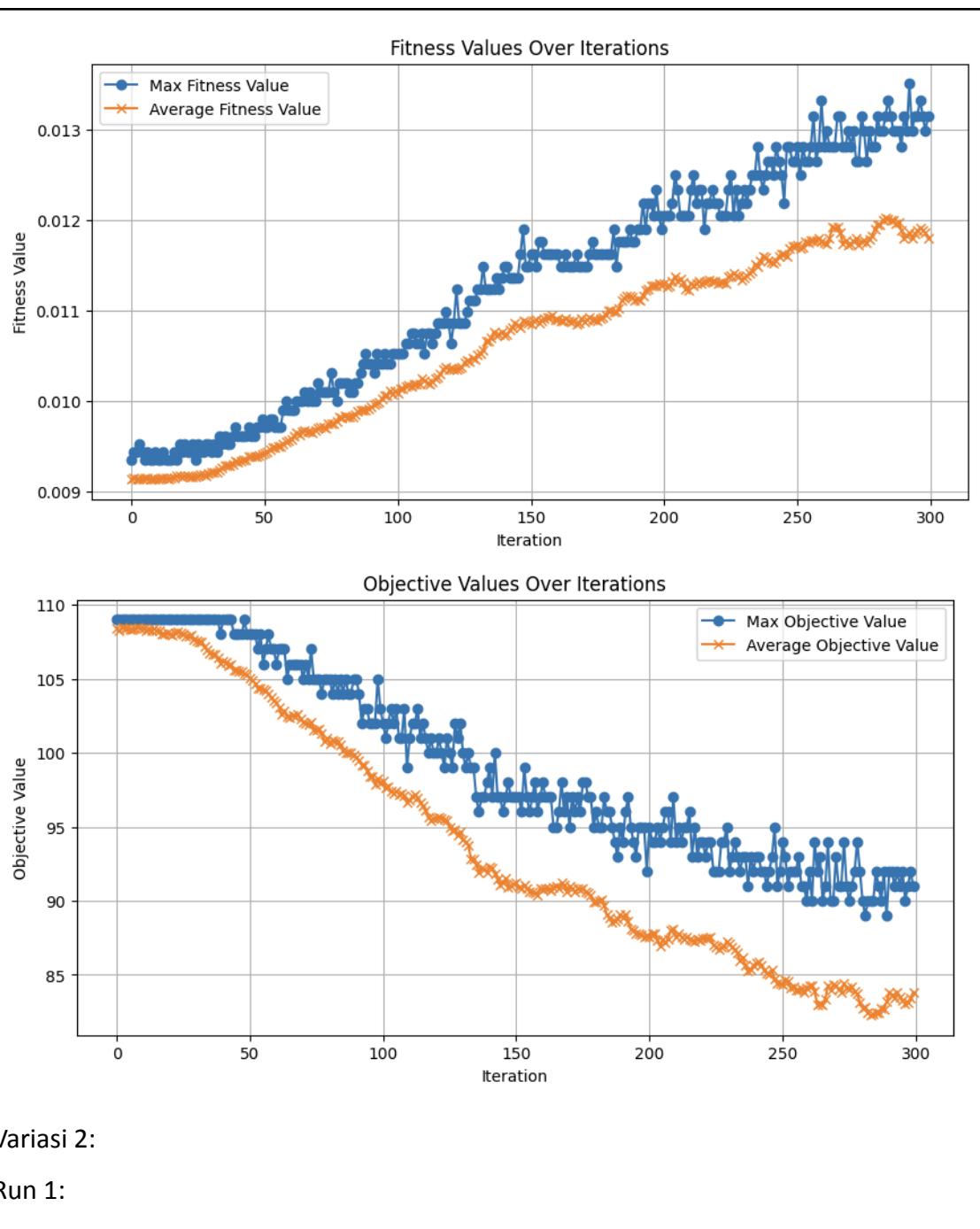


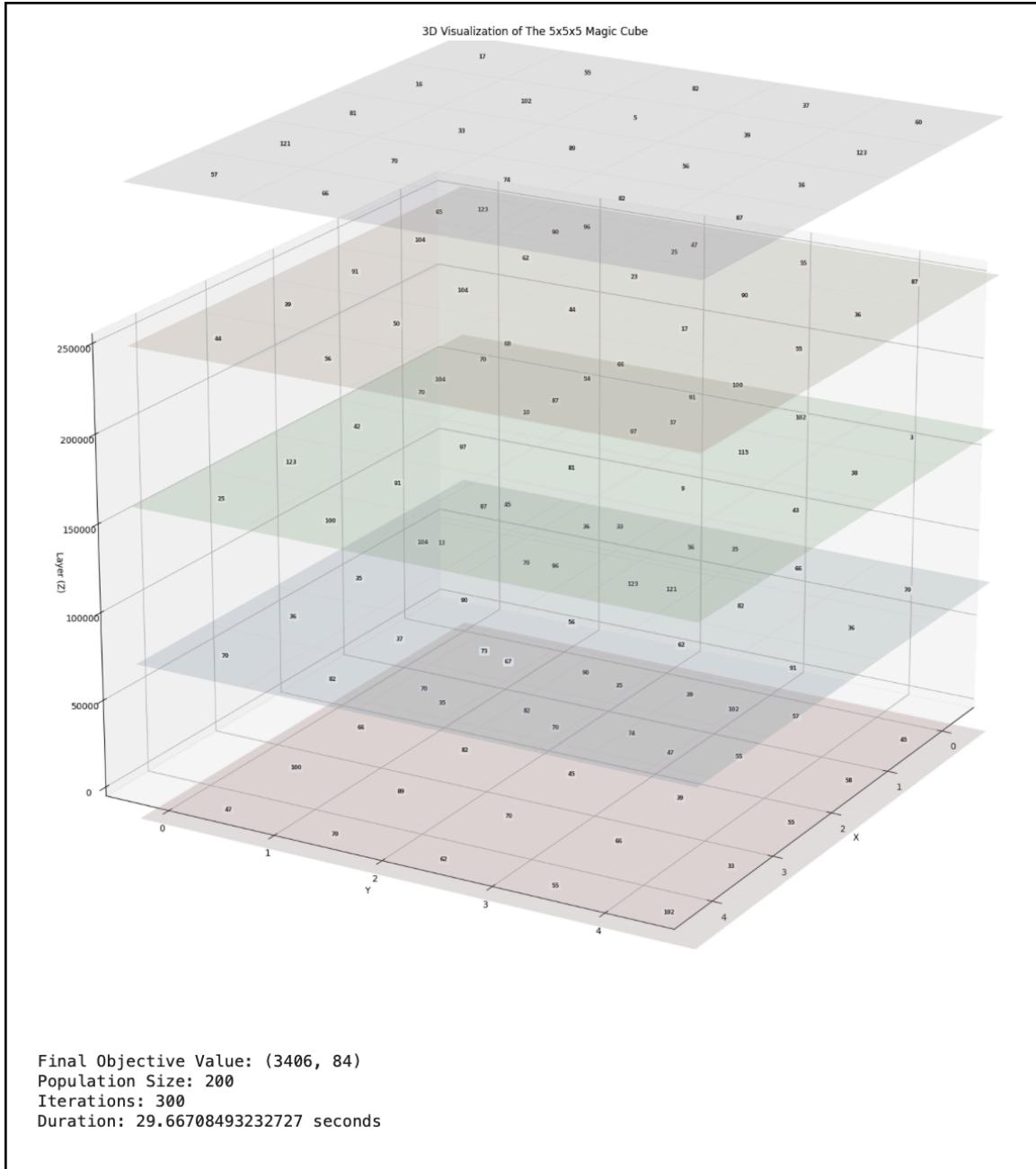


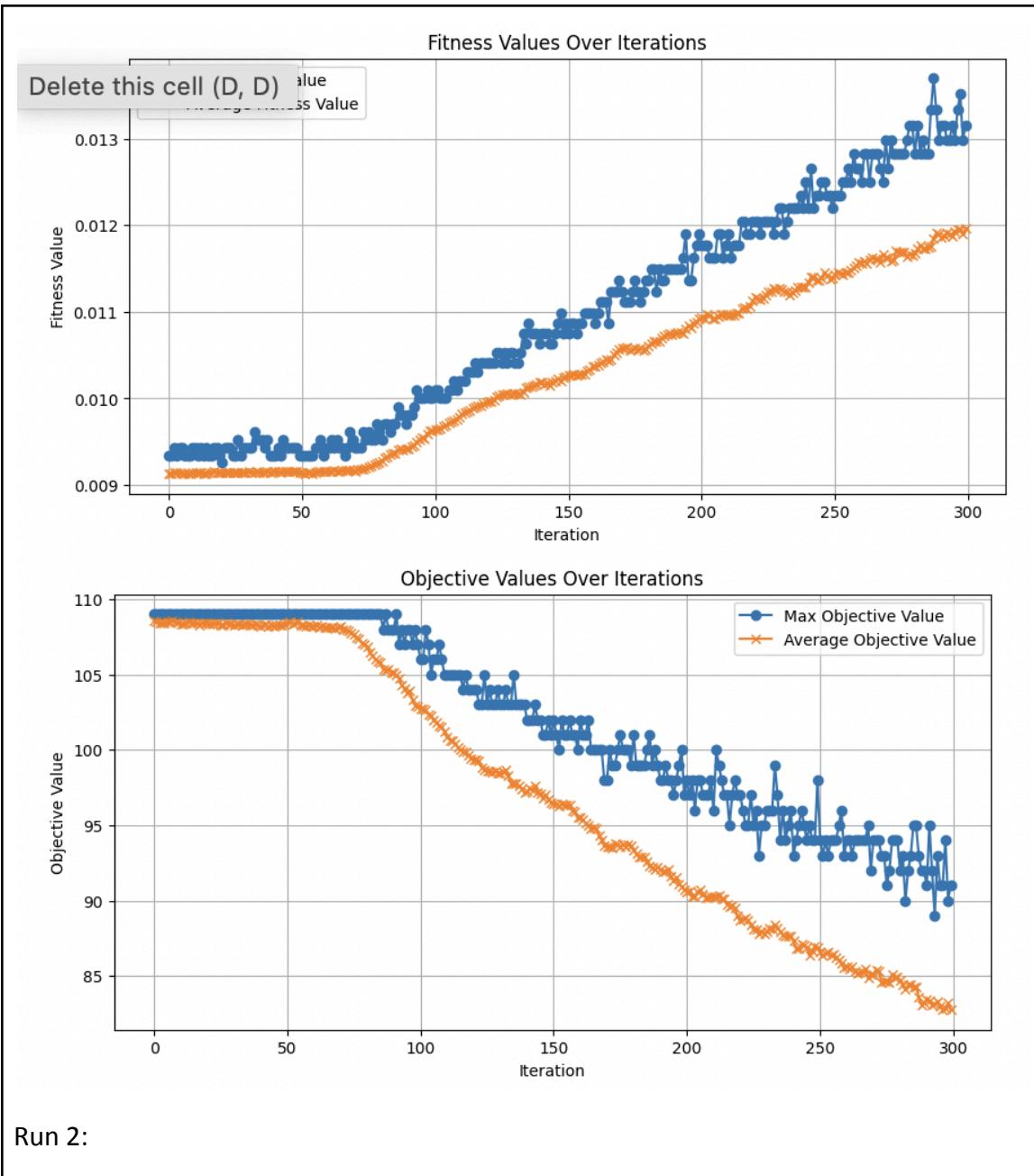


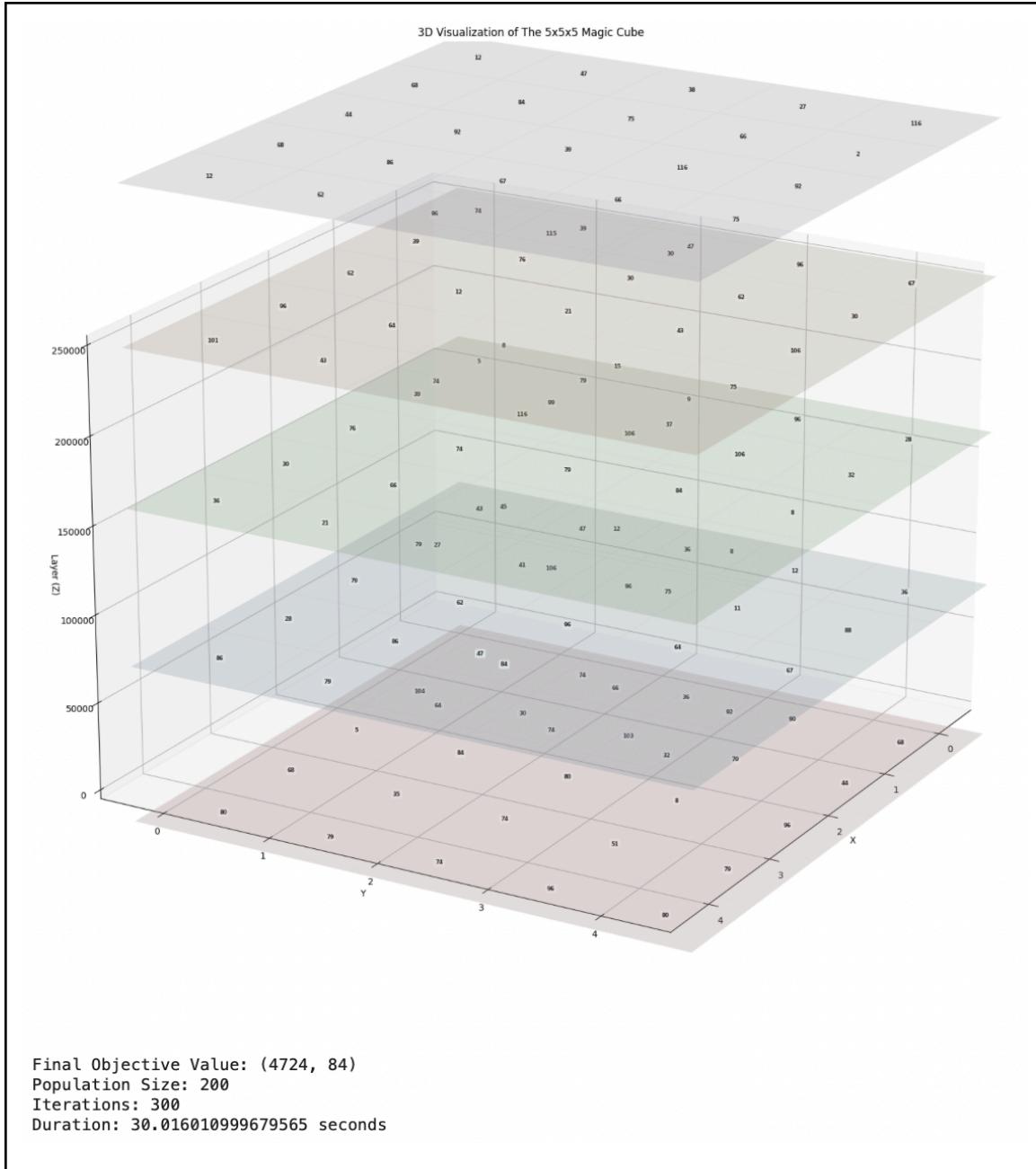


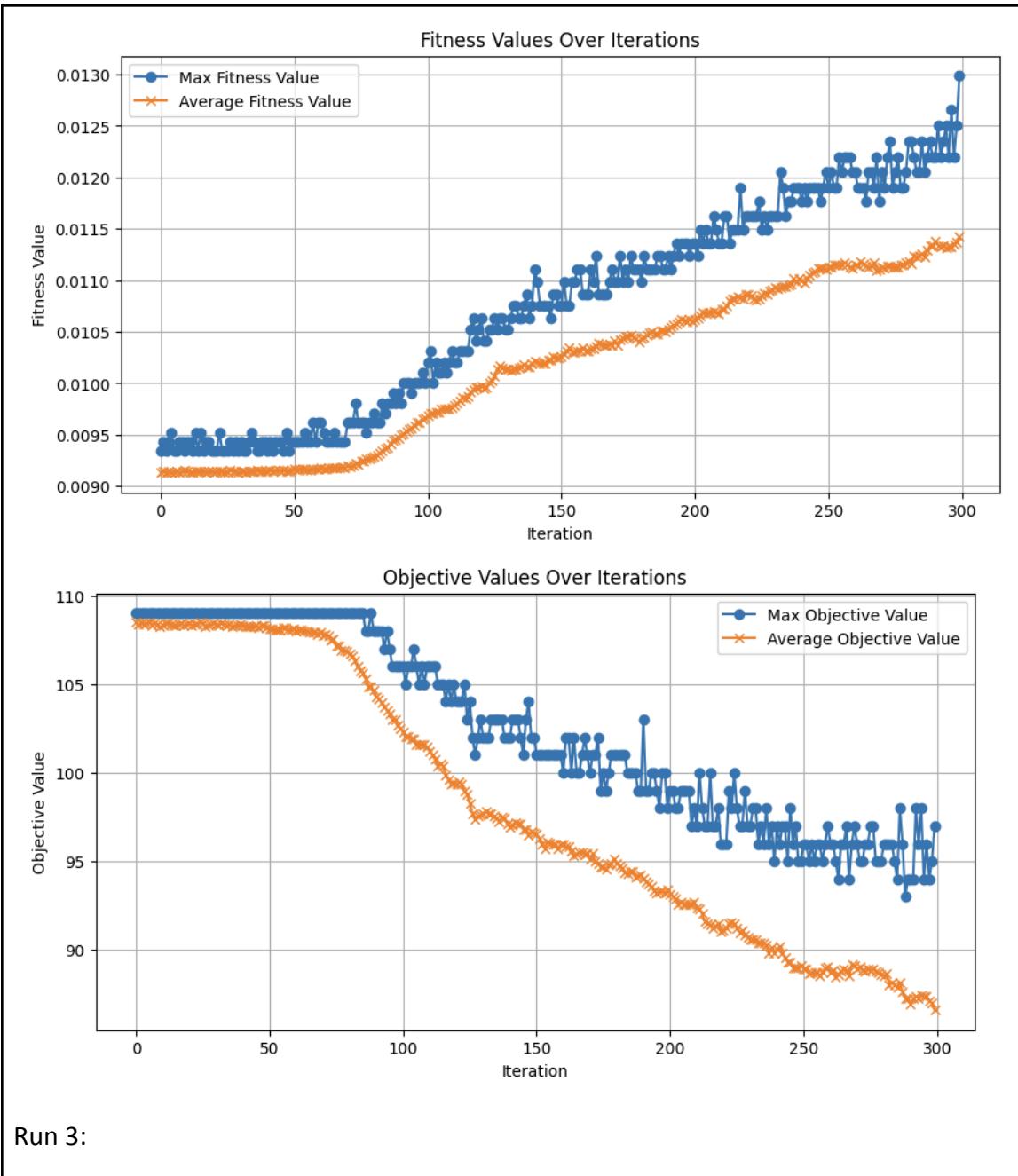


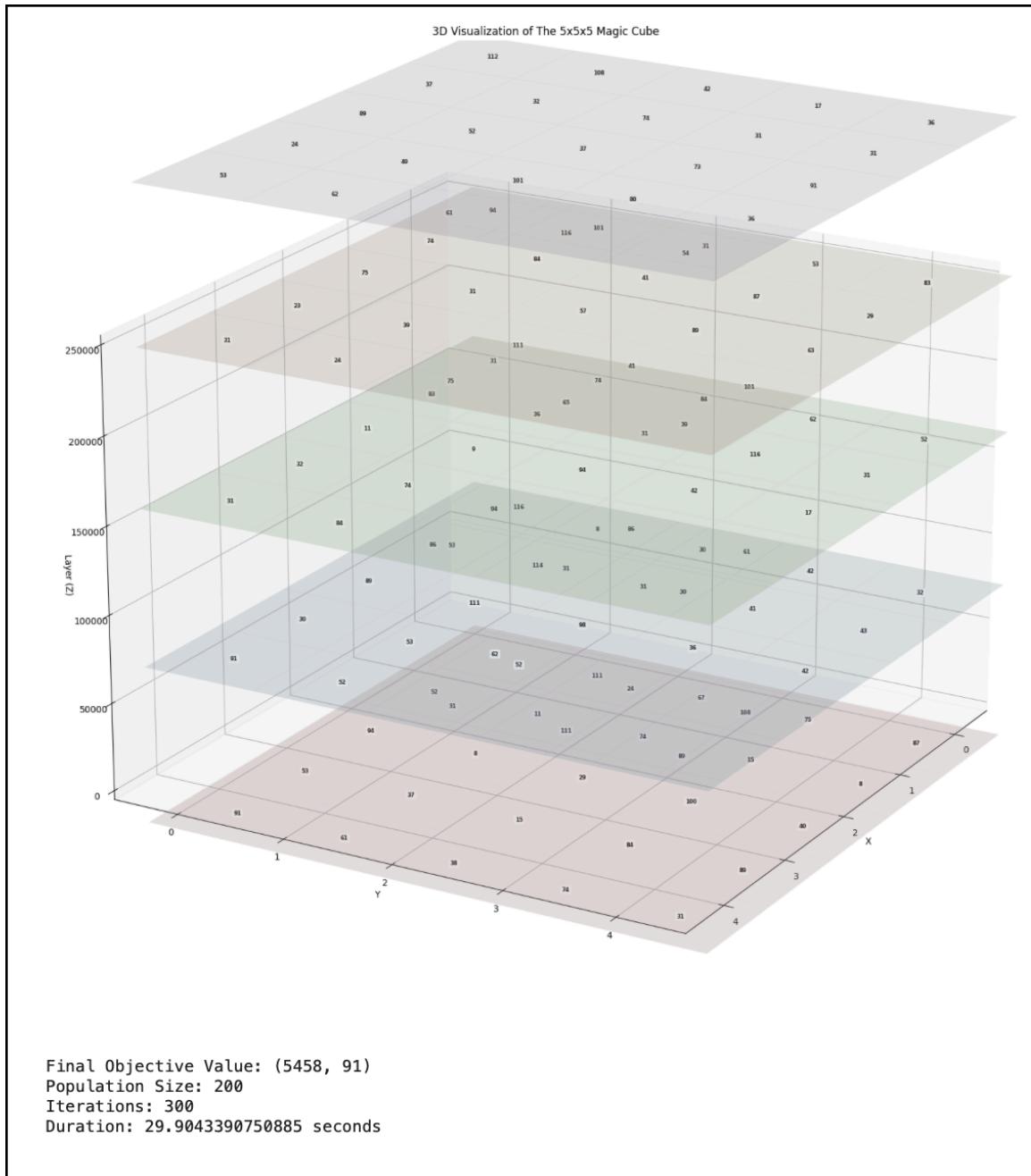


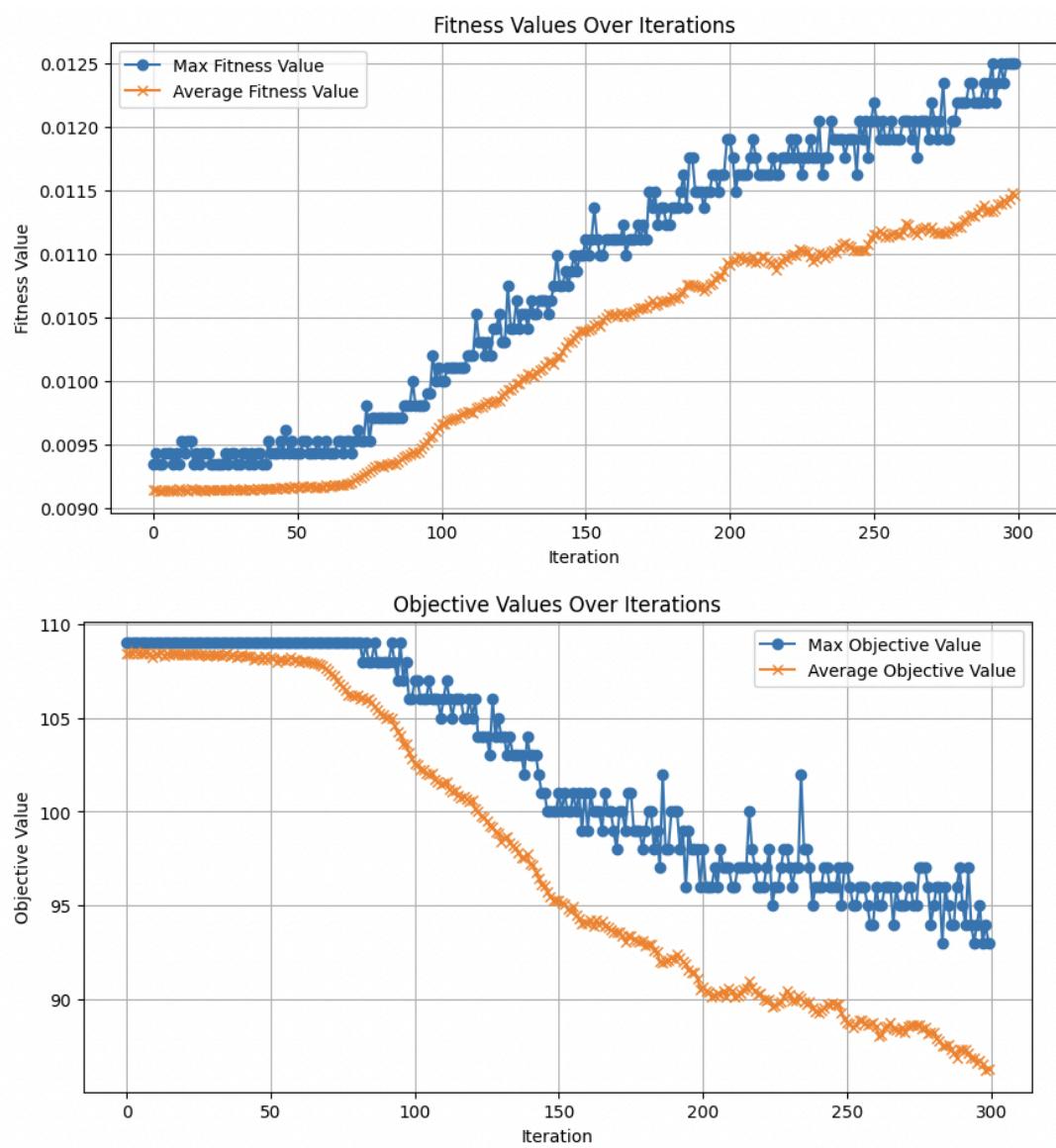






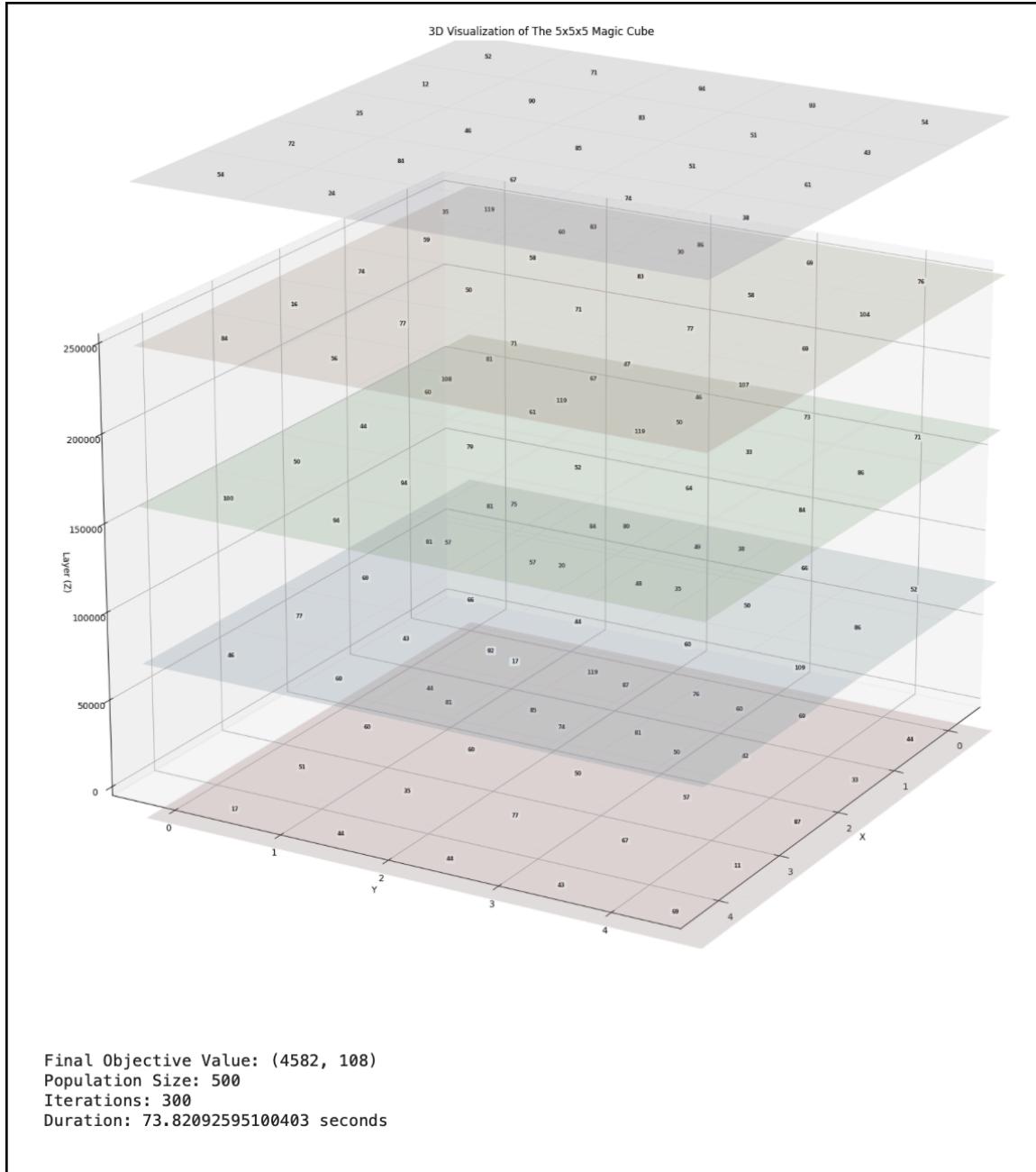


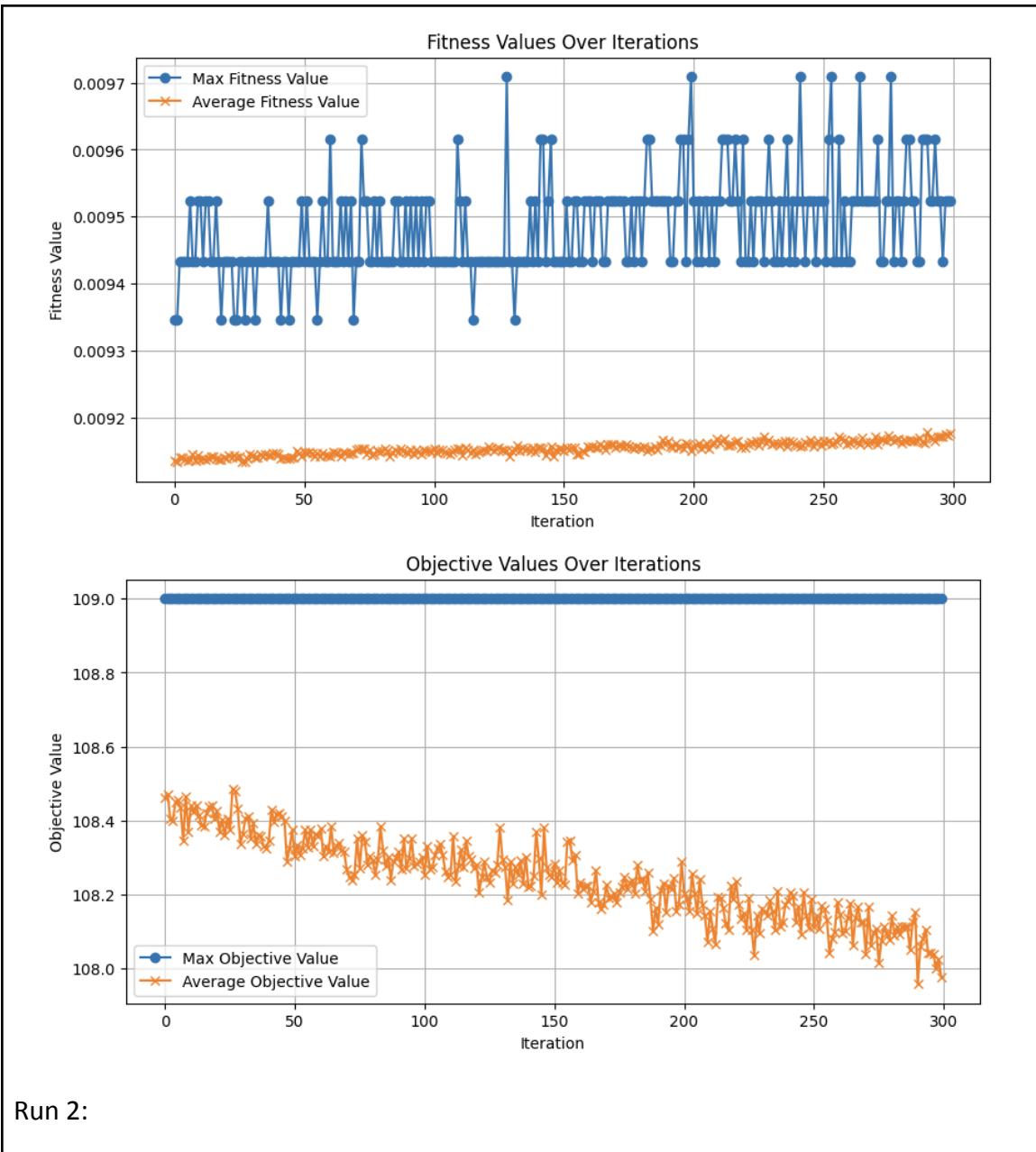


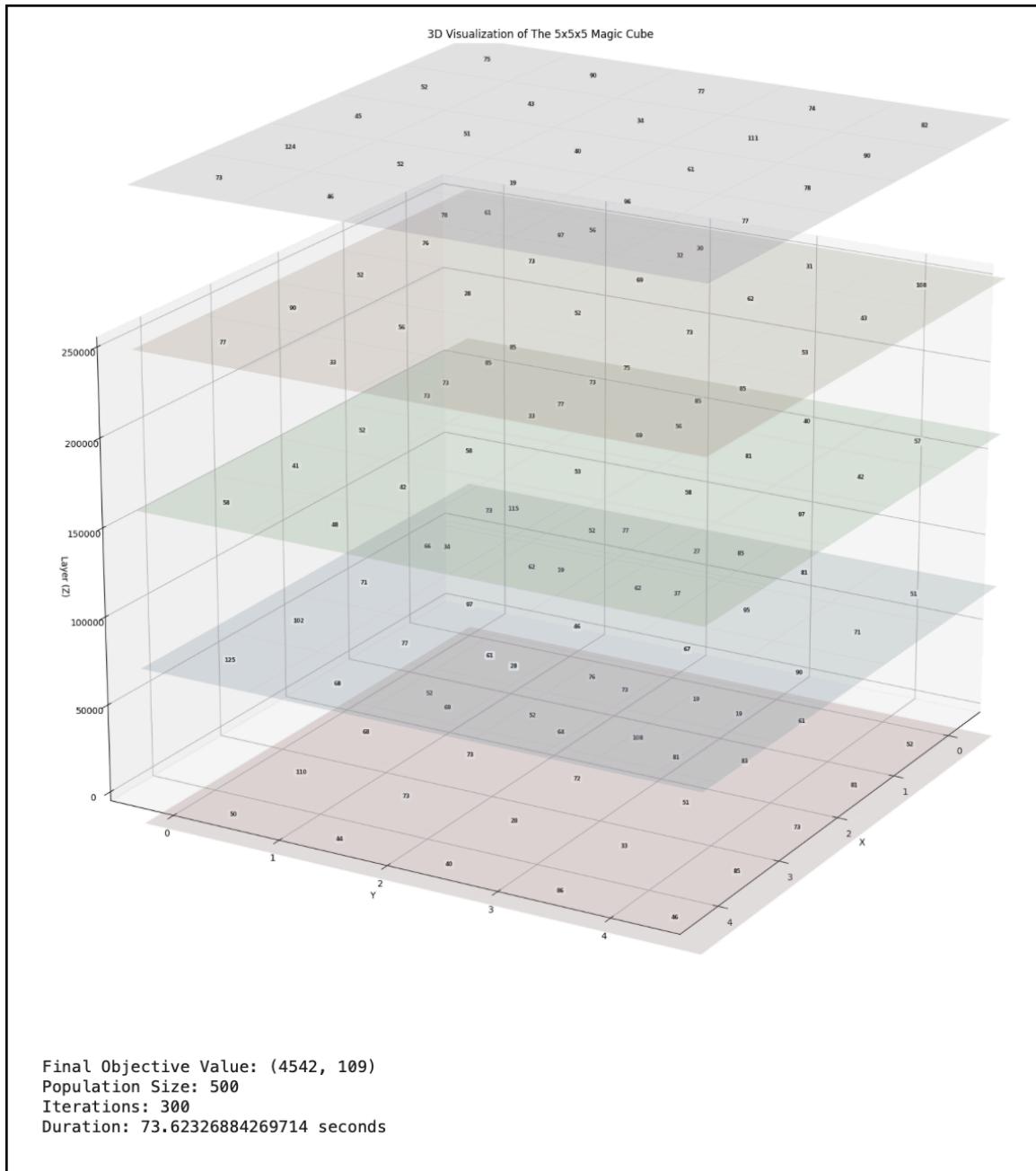


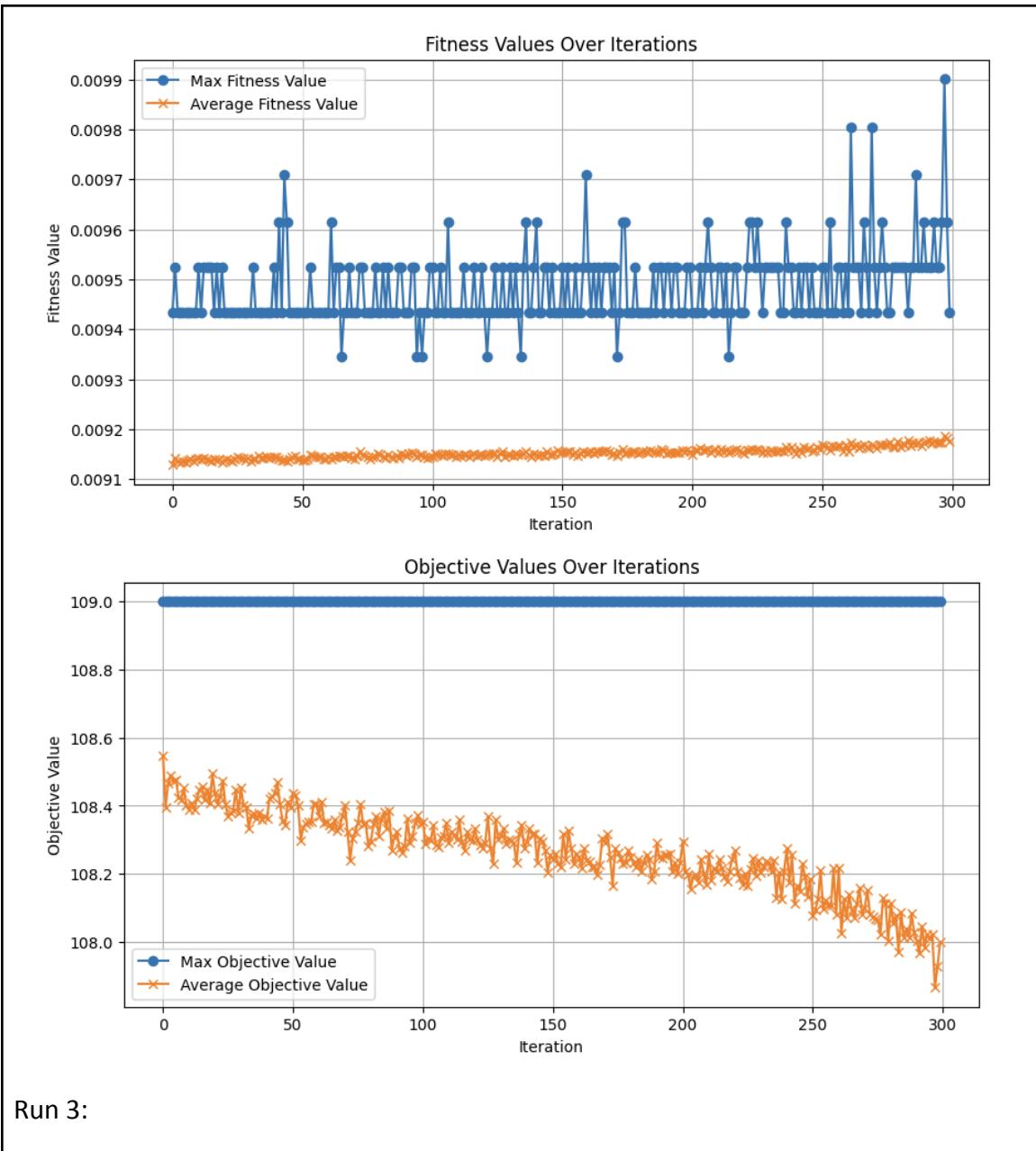
Variasi 3:

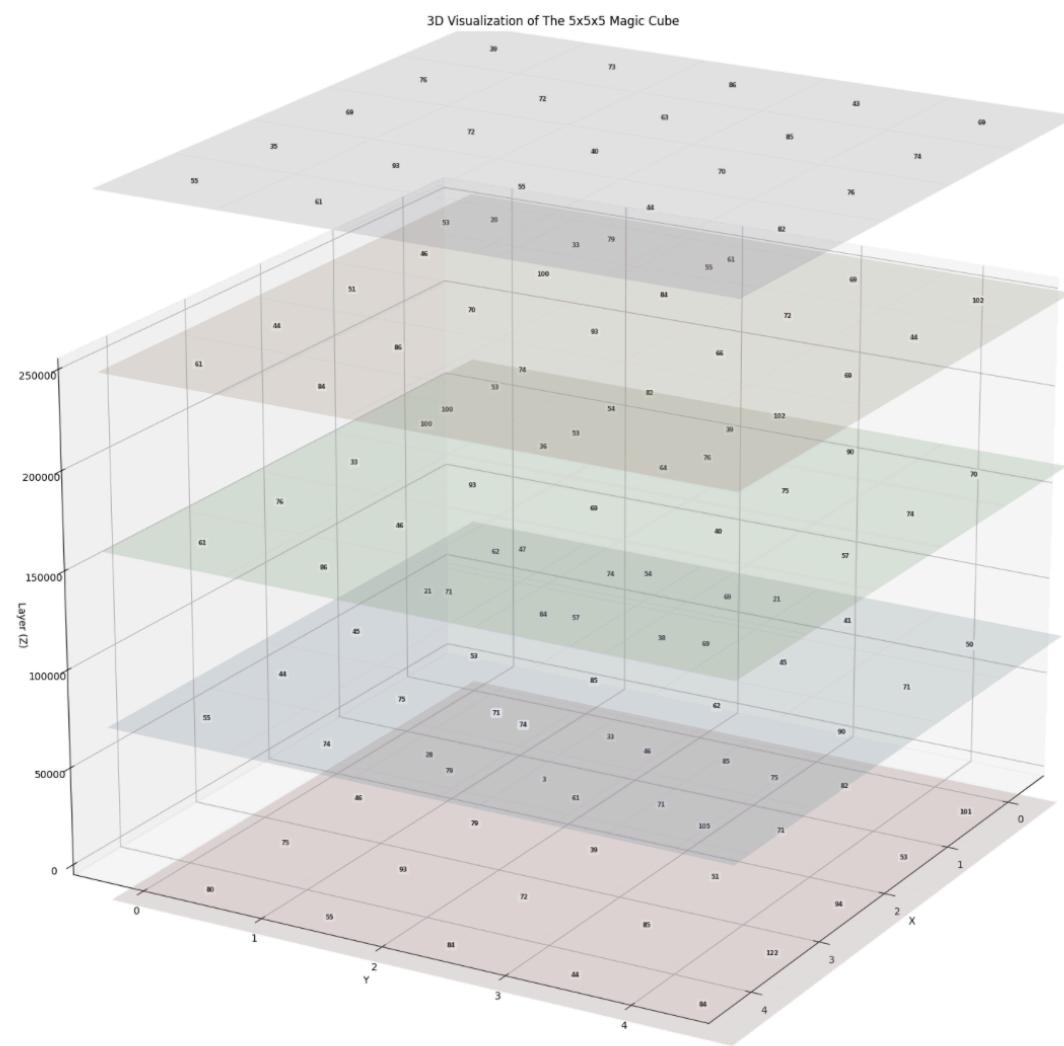
Run 1:



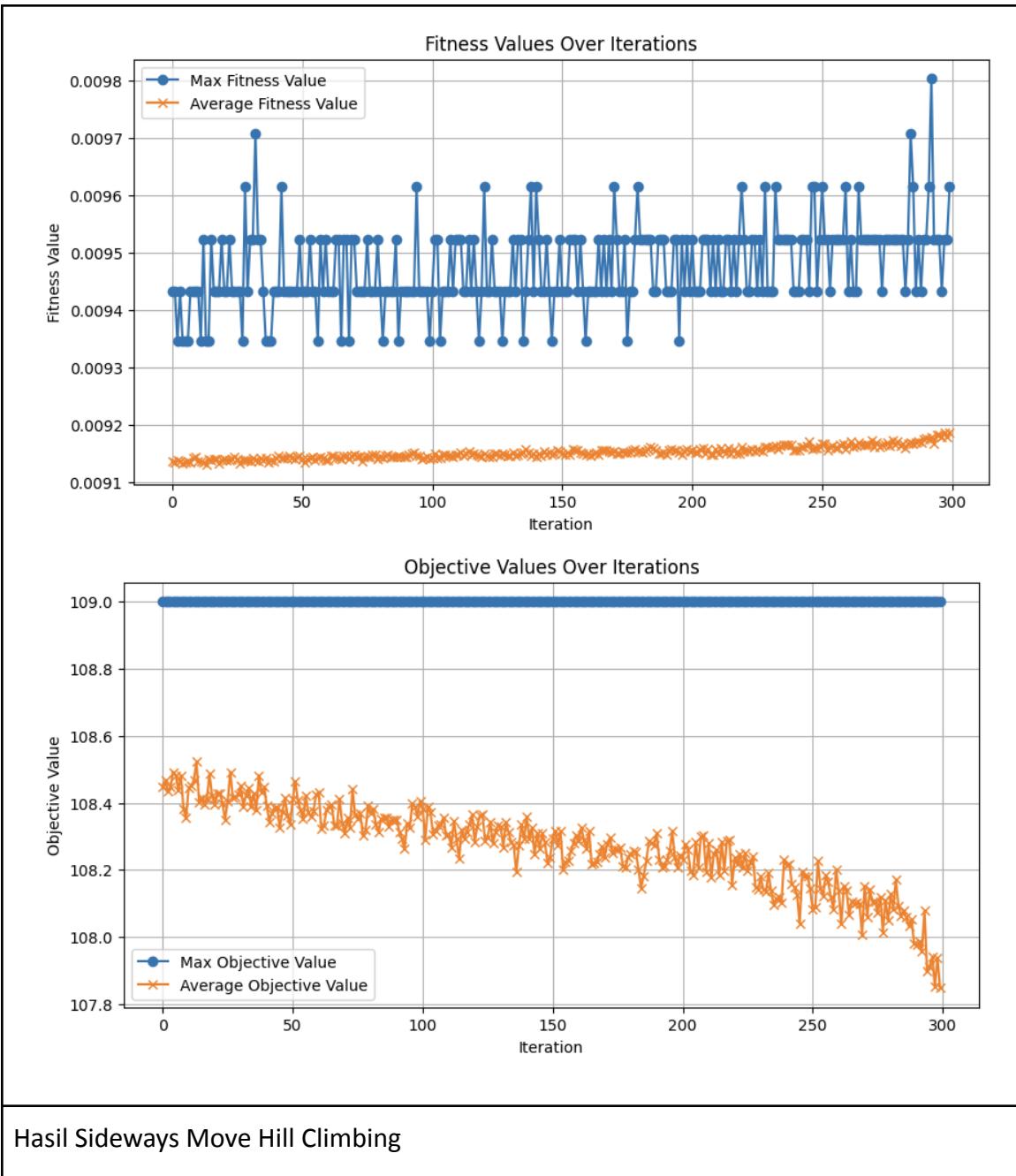






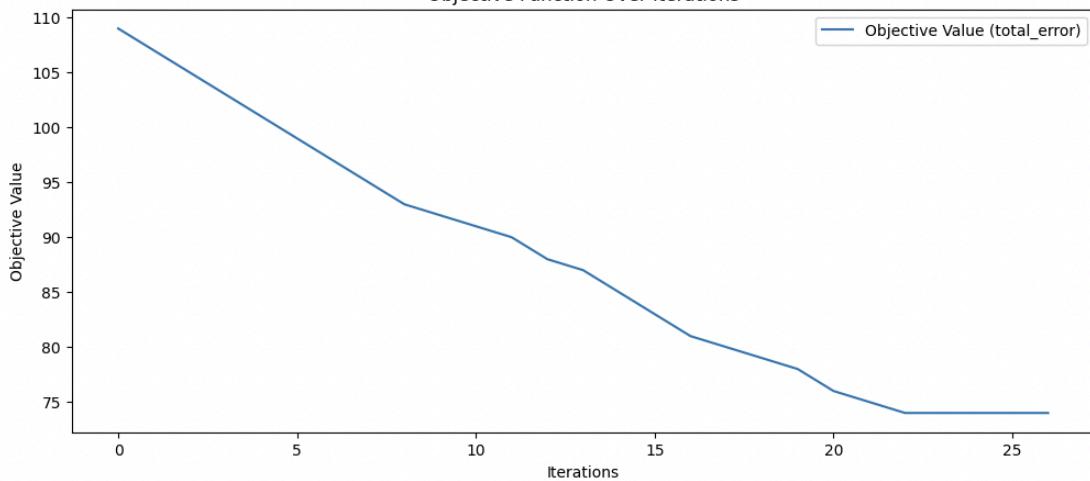


```
Final Objective Value: (4660, 108)
Population Size: 500
Iterations: 300
Duration: 74.50124216079712 seconds
```

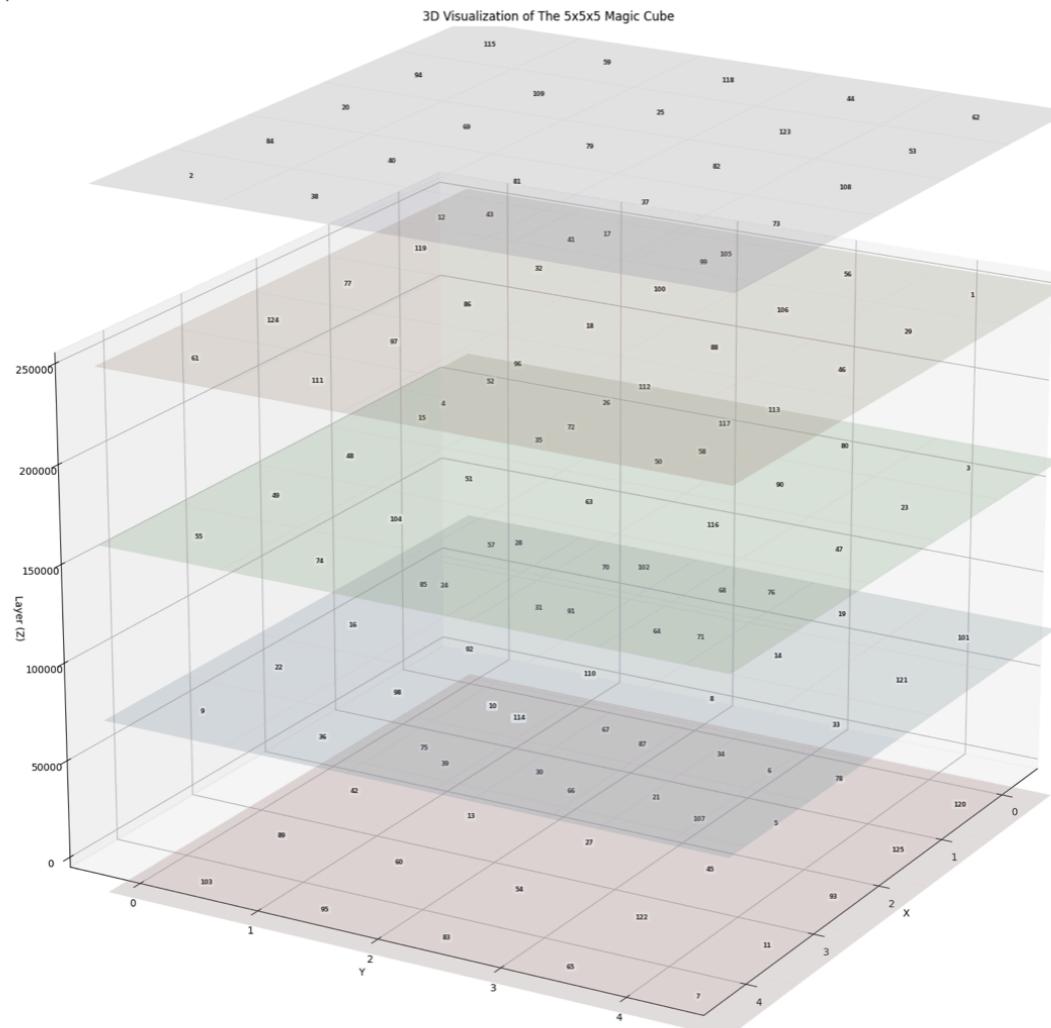


Jumlah iterasi: 26

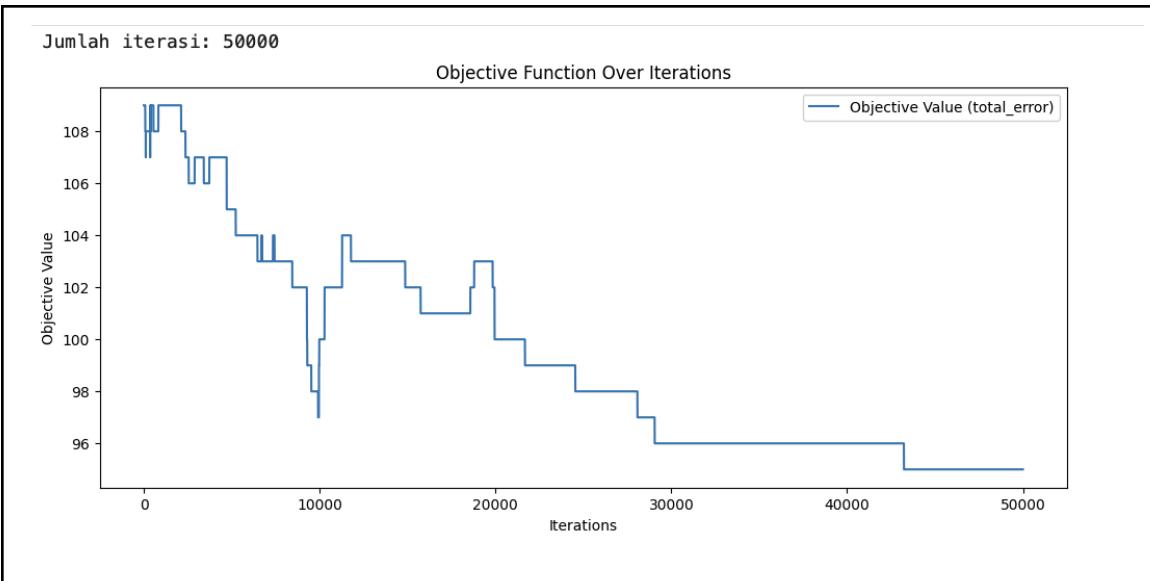
Objective Function Over Iterations



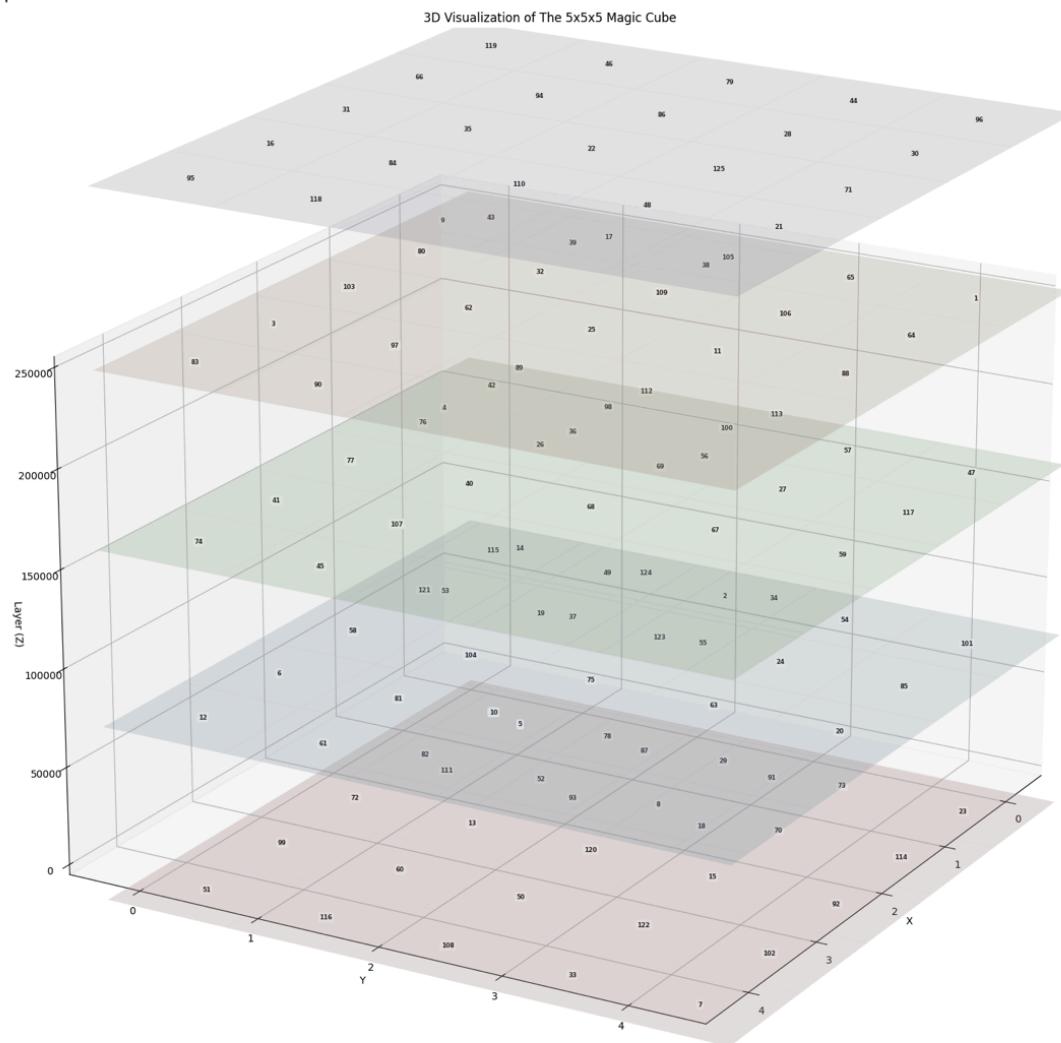
Optimized Cube:



Hasil Stochastic Hill Climbing



Optimized Cube:

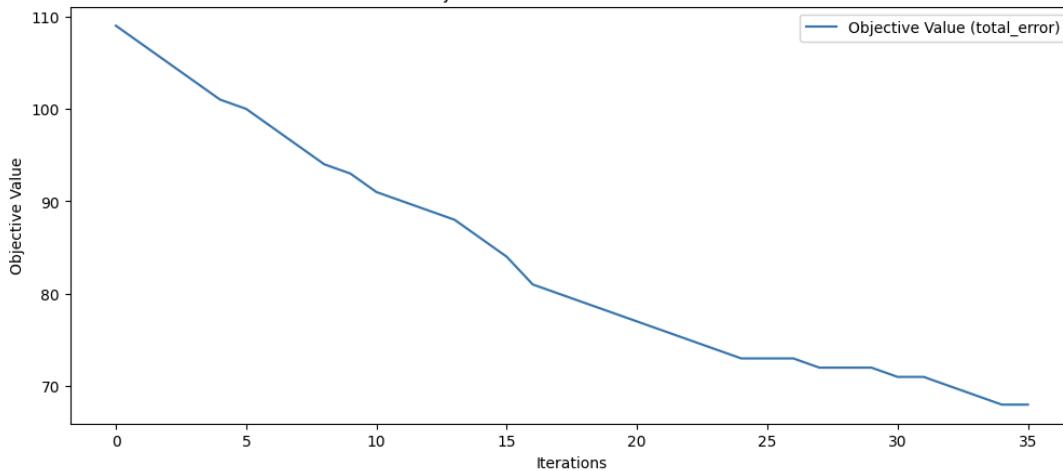


Best Cost: (1919, 95)
Duration: 16.22 seconds

Hasil Steepest Ascent Hill Climbing

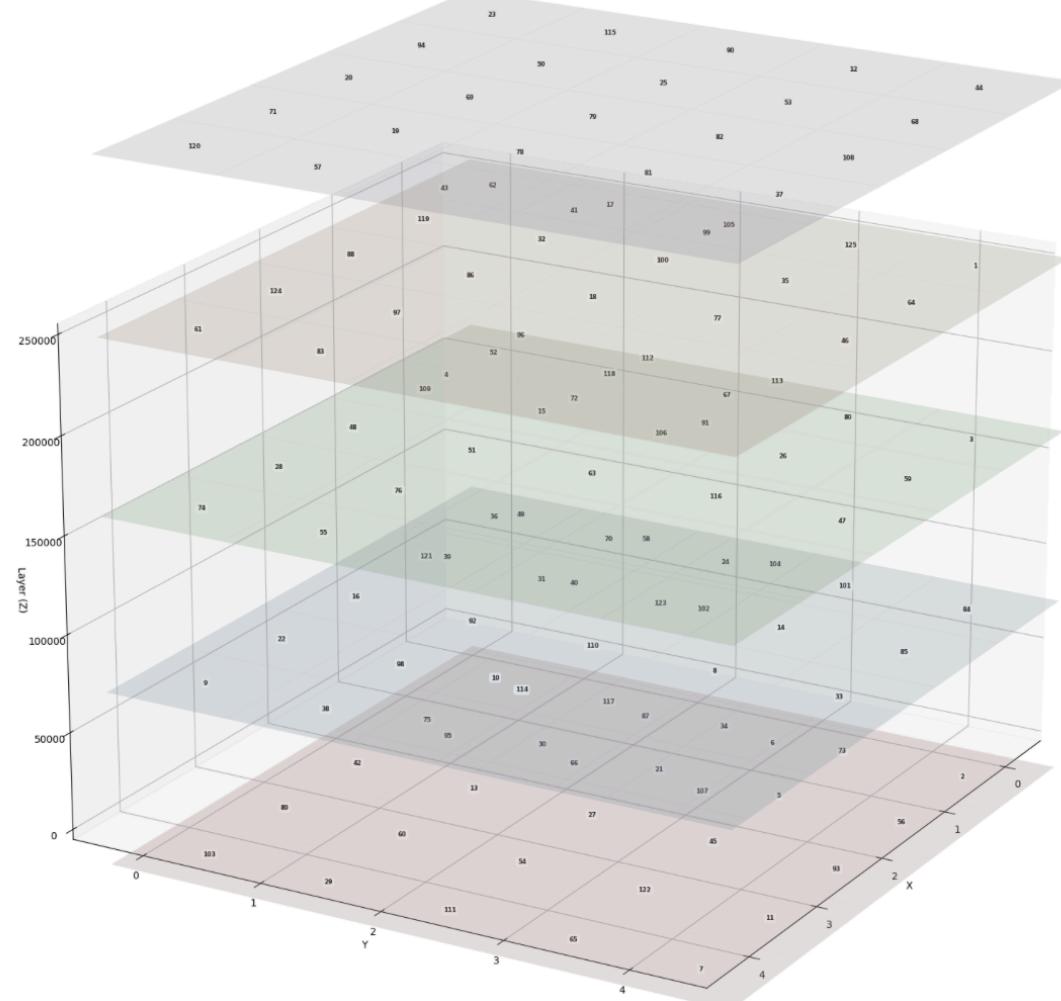
Jumlah iterasi: 35

Objective Function Over Iterations



Optimized Cube:

3D Visualization of The 5x5 Magic Cube

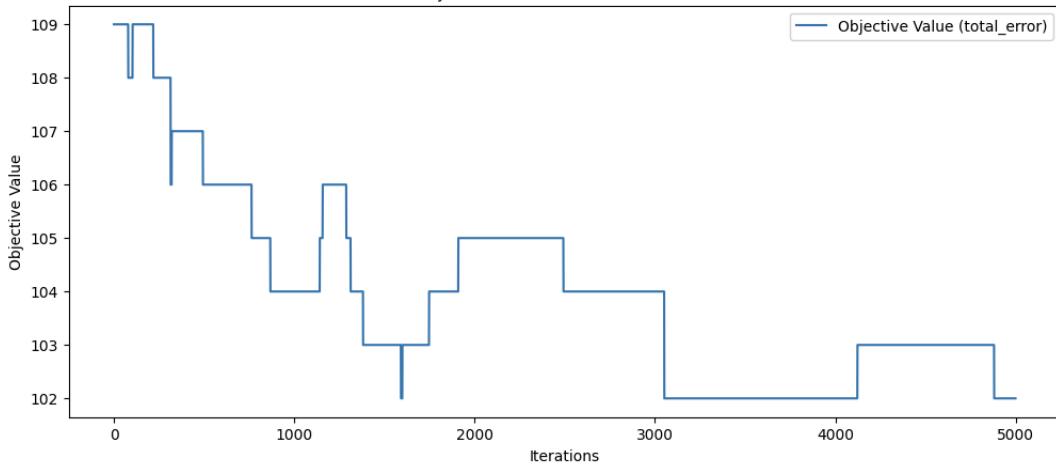


Best Cost: (4290, 68)
Duration: 96.17 seconds

Hasil Random Restart Hill Climbing

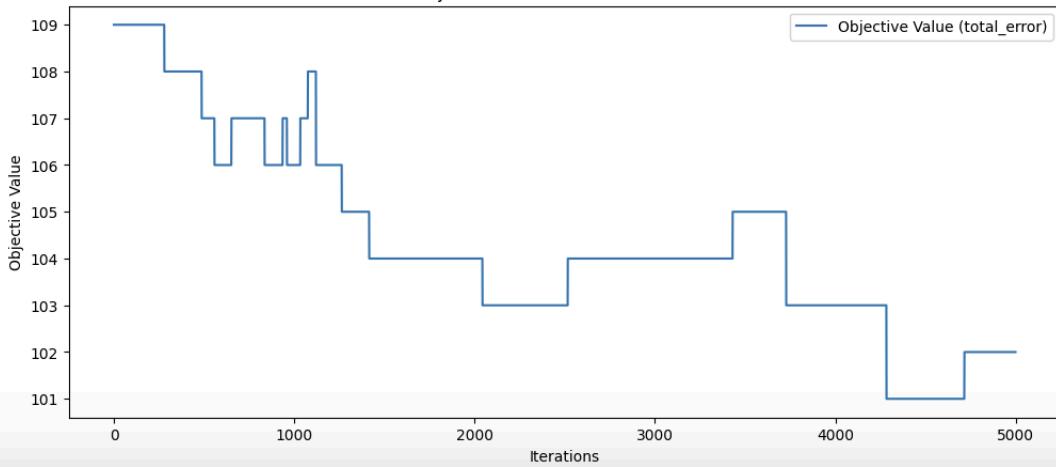
Restart 1:
Jumlah iterasi: 5000

Objective Function Over Iterations



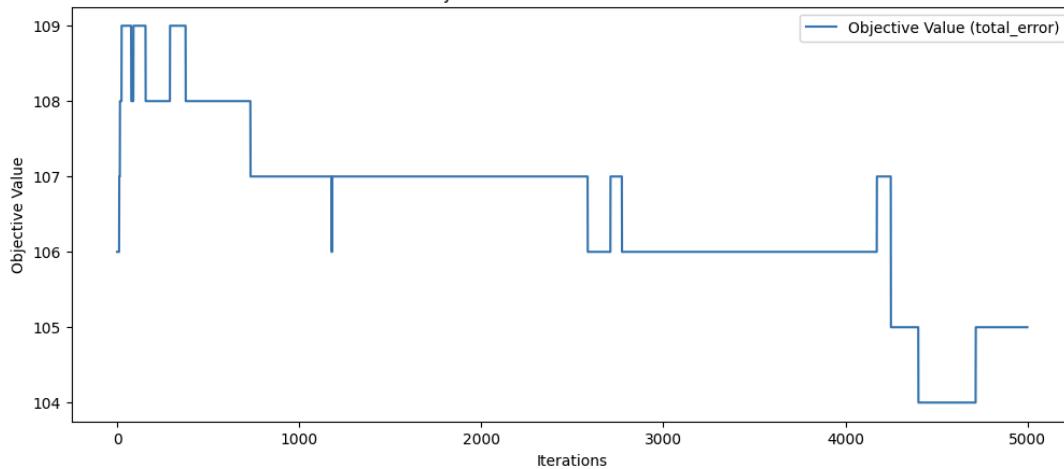
Restart 2:
Jumlah iterasi: 5000

Objective Function Over Iterations



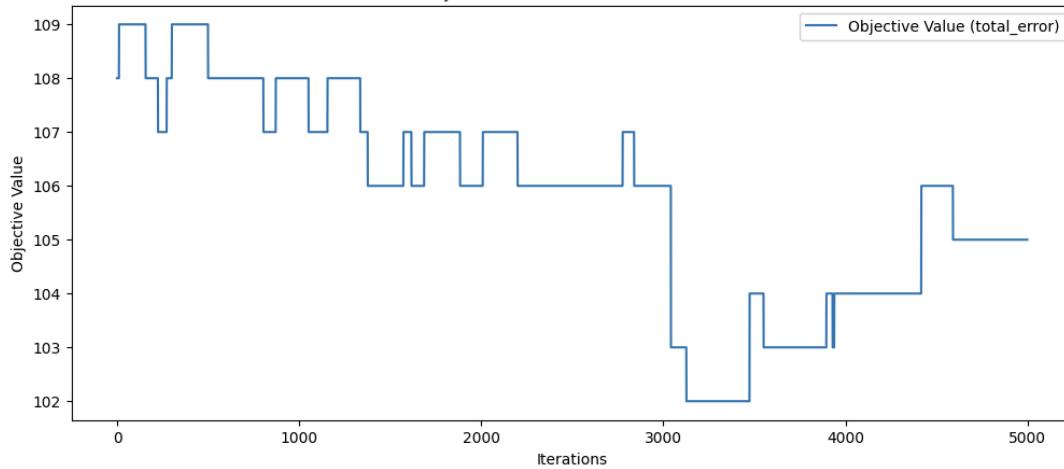
Restart 3:
Jumlah iterasi: 5000

Objective Function Over Iterations



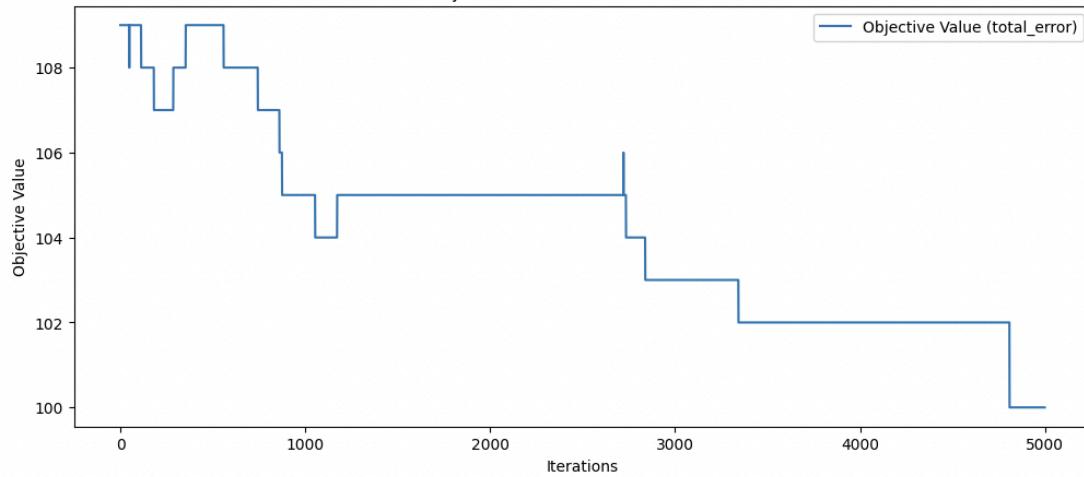
Restart 4:
Jumlah iterasi: 5000

Objective Function Over Iterations



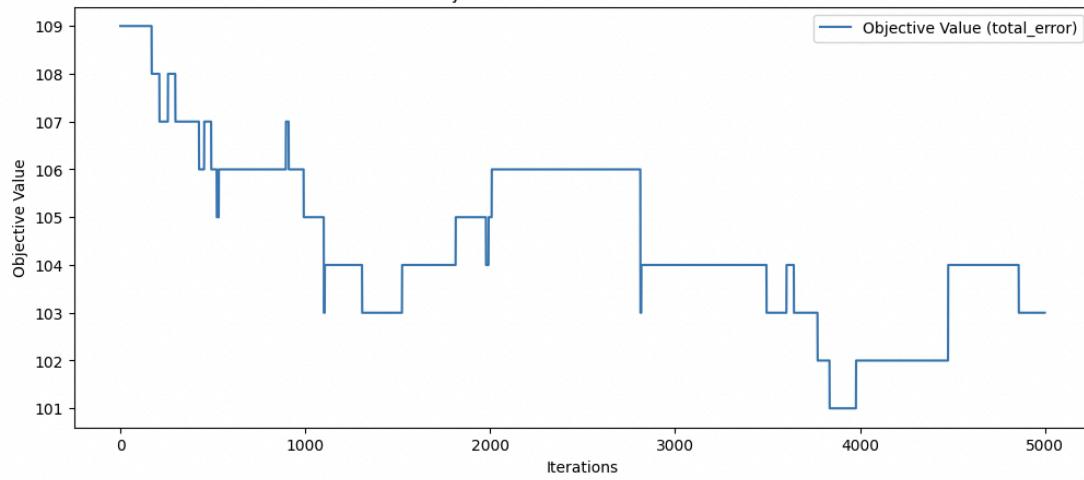
Restart 5:
Jumlah iterasi: 5000

Objective Function Over Iterations

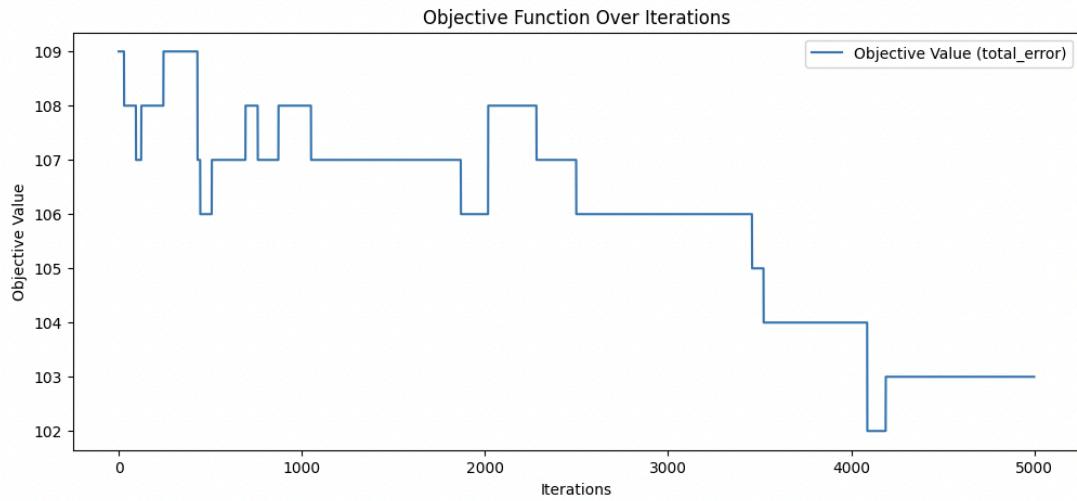


Restart 6:
Jumlah iterasi: 5000

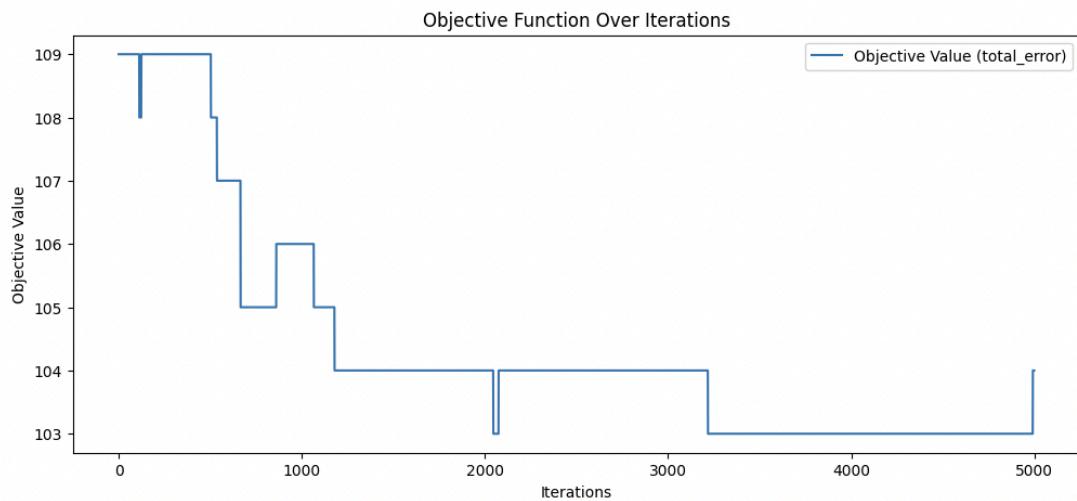
Objective Function Over Iterations



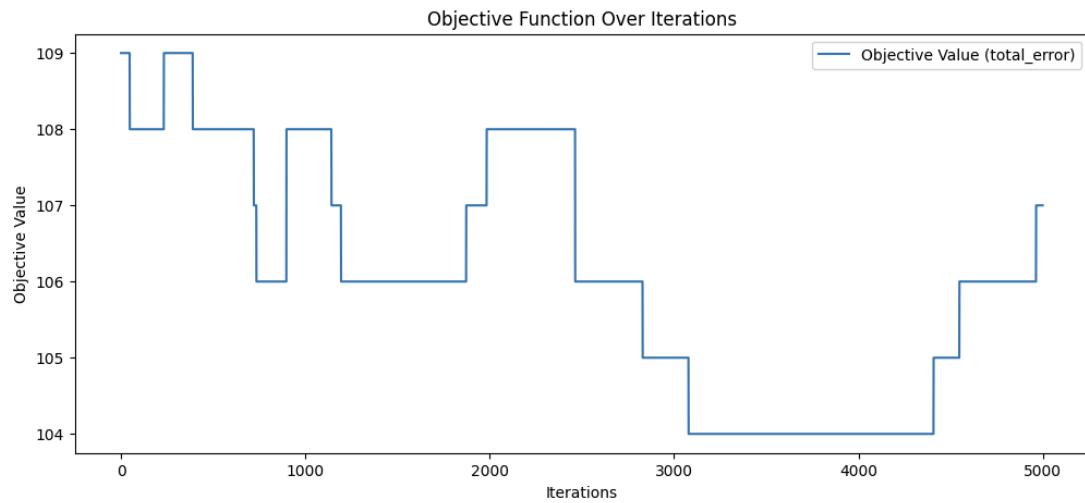
Restart 7:
Jumlah iterasi: 5000



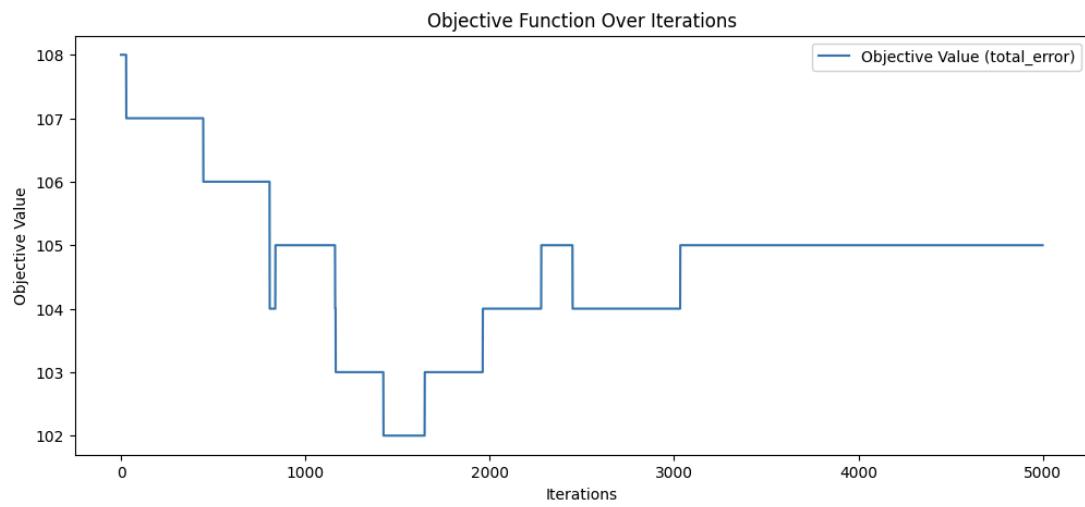
Restart 8:
Jumlah iterasi: 5000



Restart 9:
Jumlah iterasi: 5000

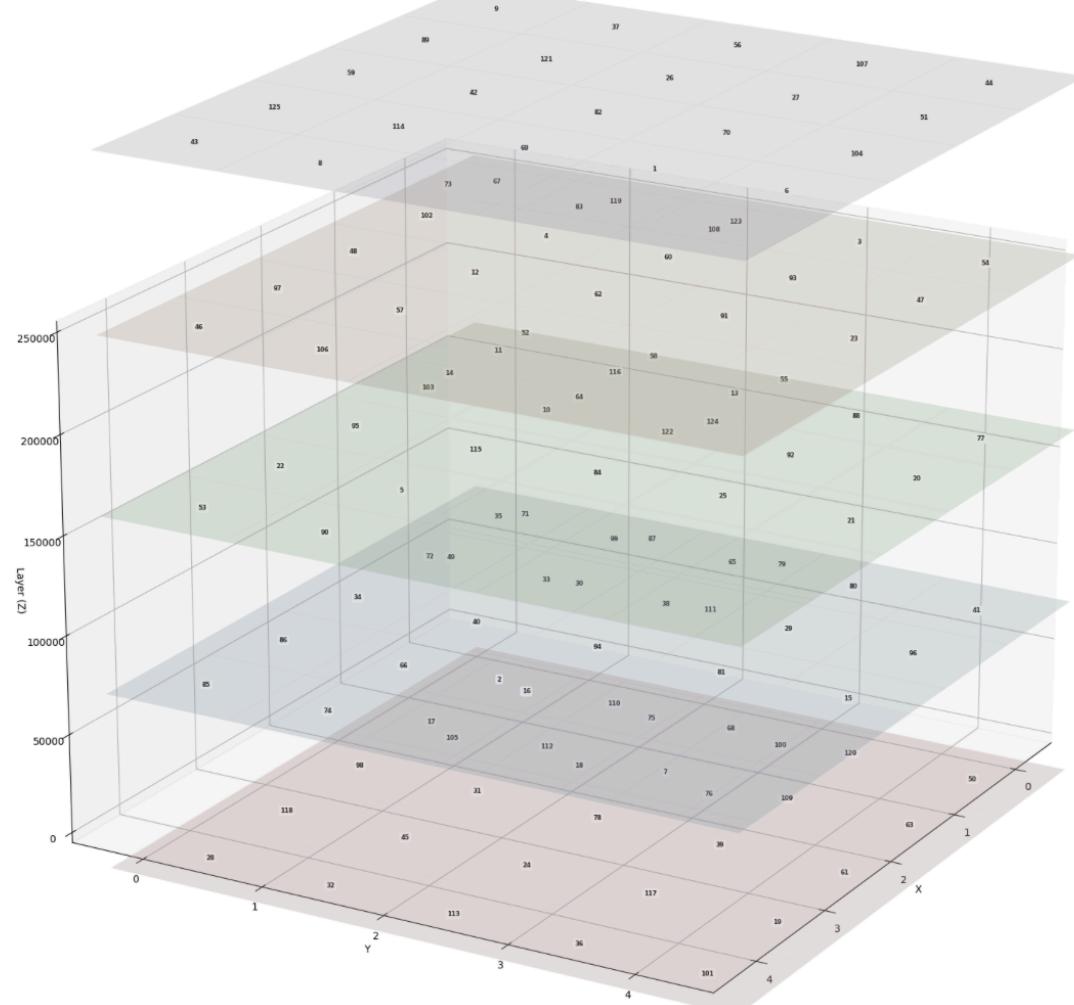


Restart 10:
Jumlah iterasi: 5000



Jumlah restart: 10
Optimized Cube:

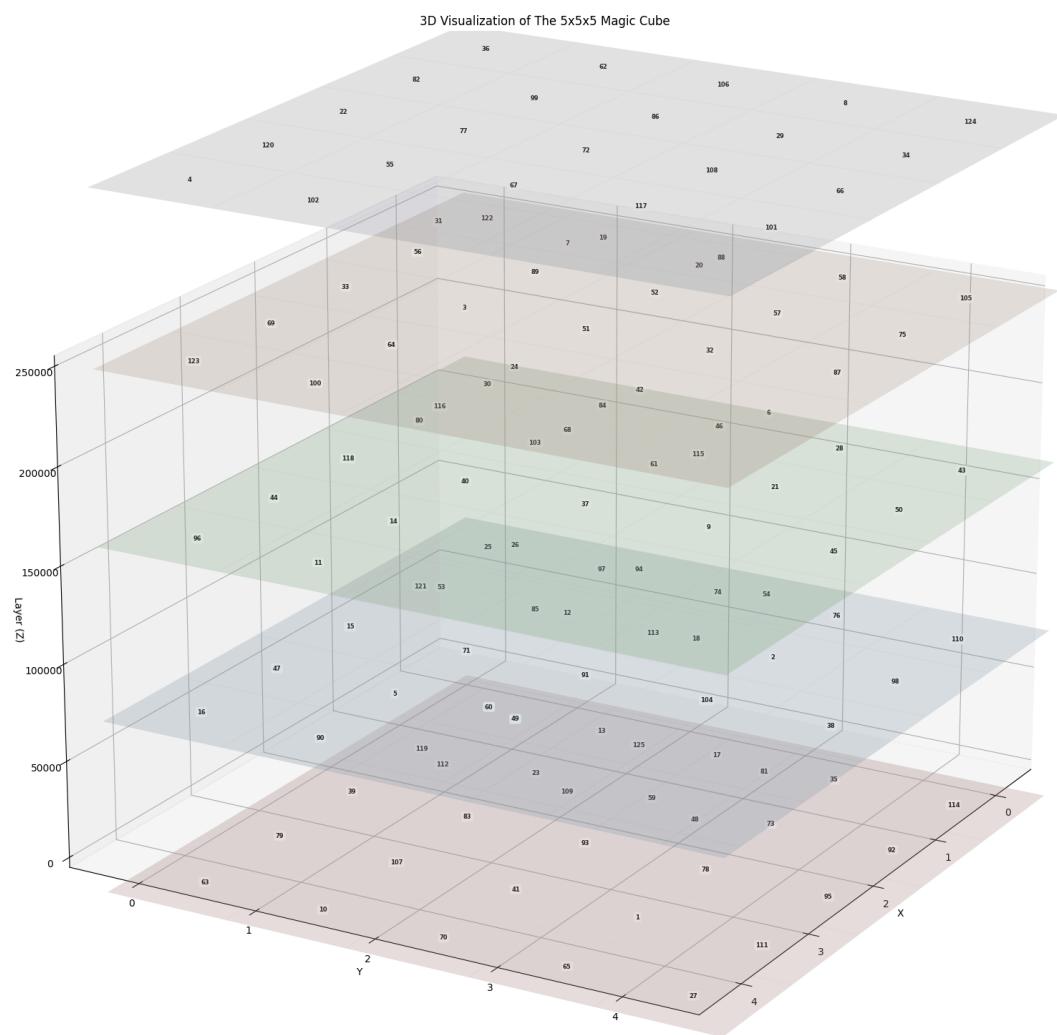
3D Visualization of The 5x5 Magic Cube



Best Cost: (2357, 100)
Duration: 16.69 seconds

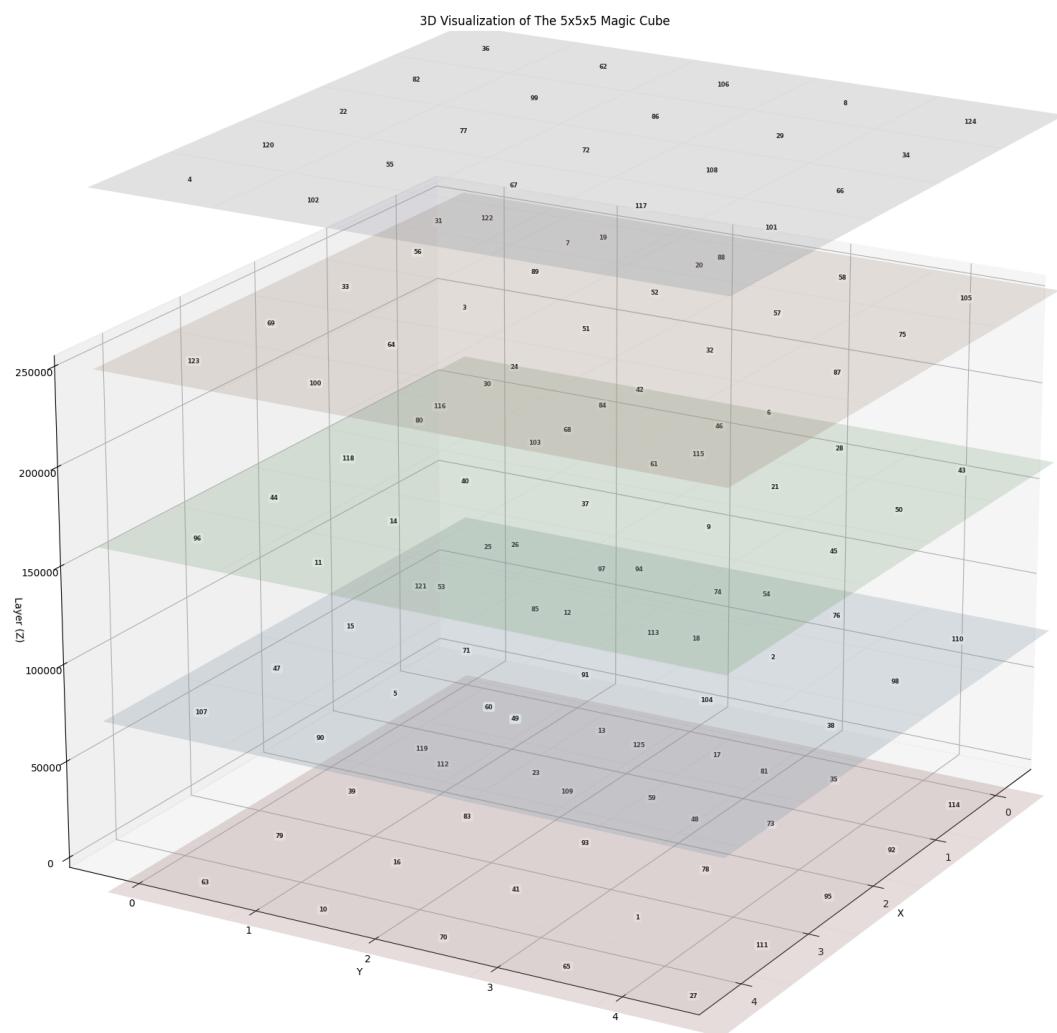
Test Case 2

Initial Cube

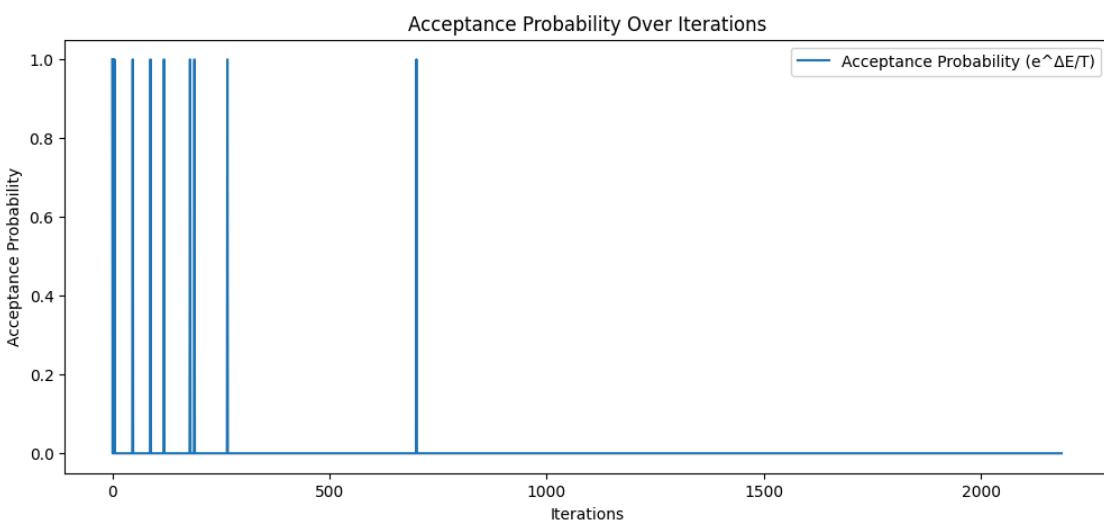
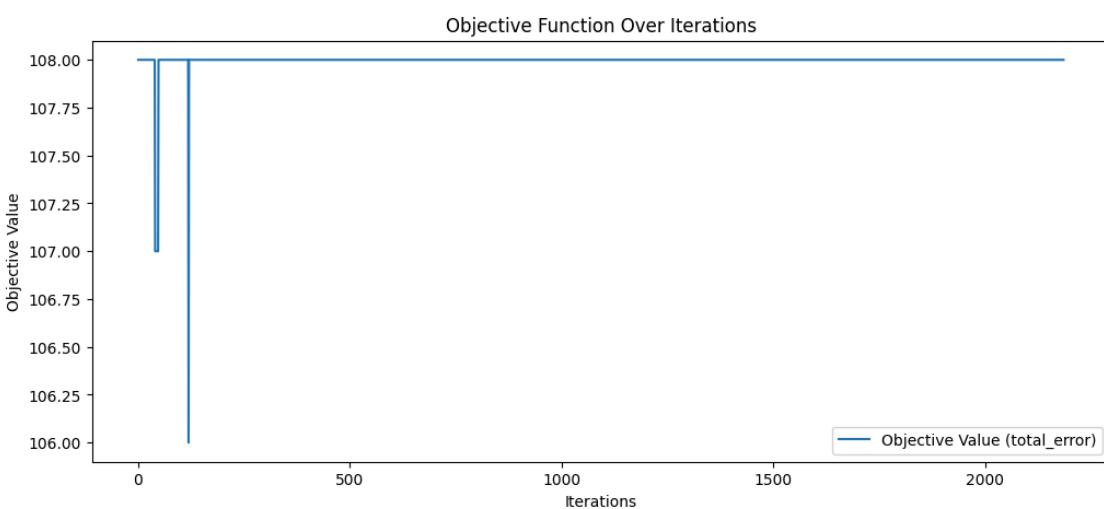


Initial Objective Function Value (total_error, error_count): (7207, 108)

Hasil Simulated Annealing



Final Objective Function Value (total_error, error_count): (7122, 106)
Duration: 1.13 seconds
Frequency of being 'stuck' at local optima: 2176

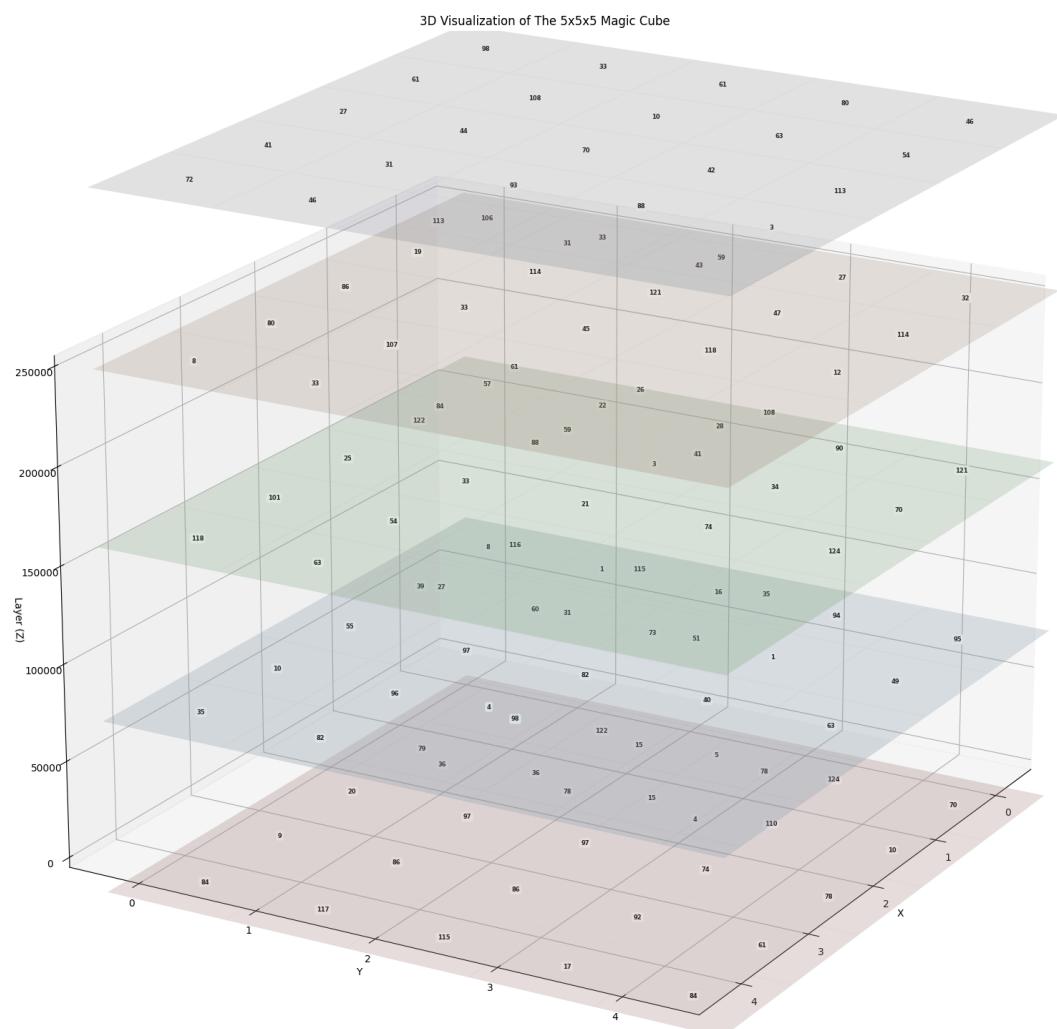


Hasil Genetic Algorithm

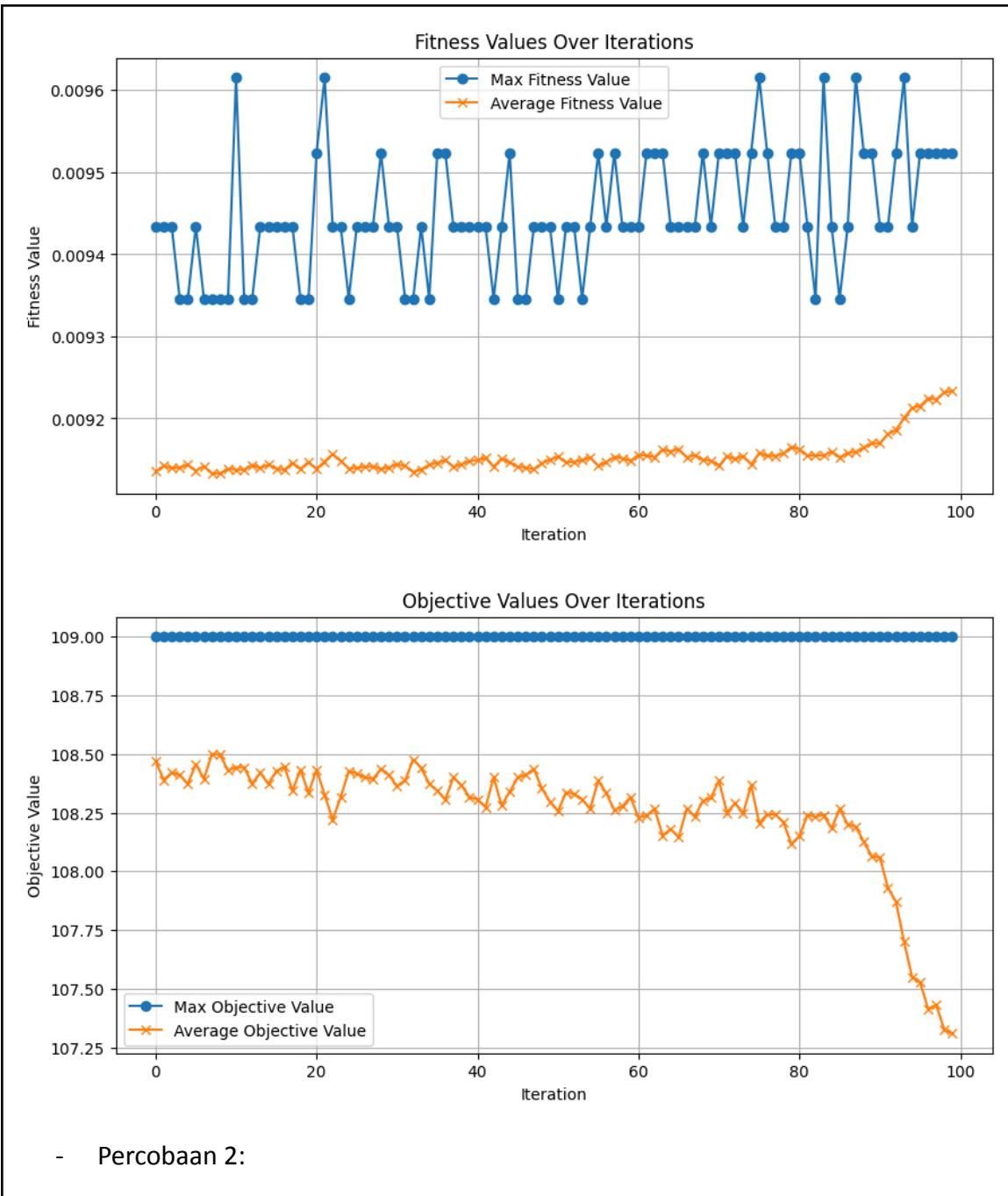
Populasi sebagai kontrol (Populasi: 300)

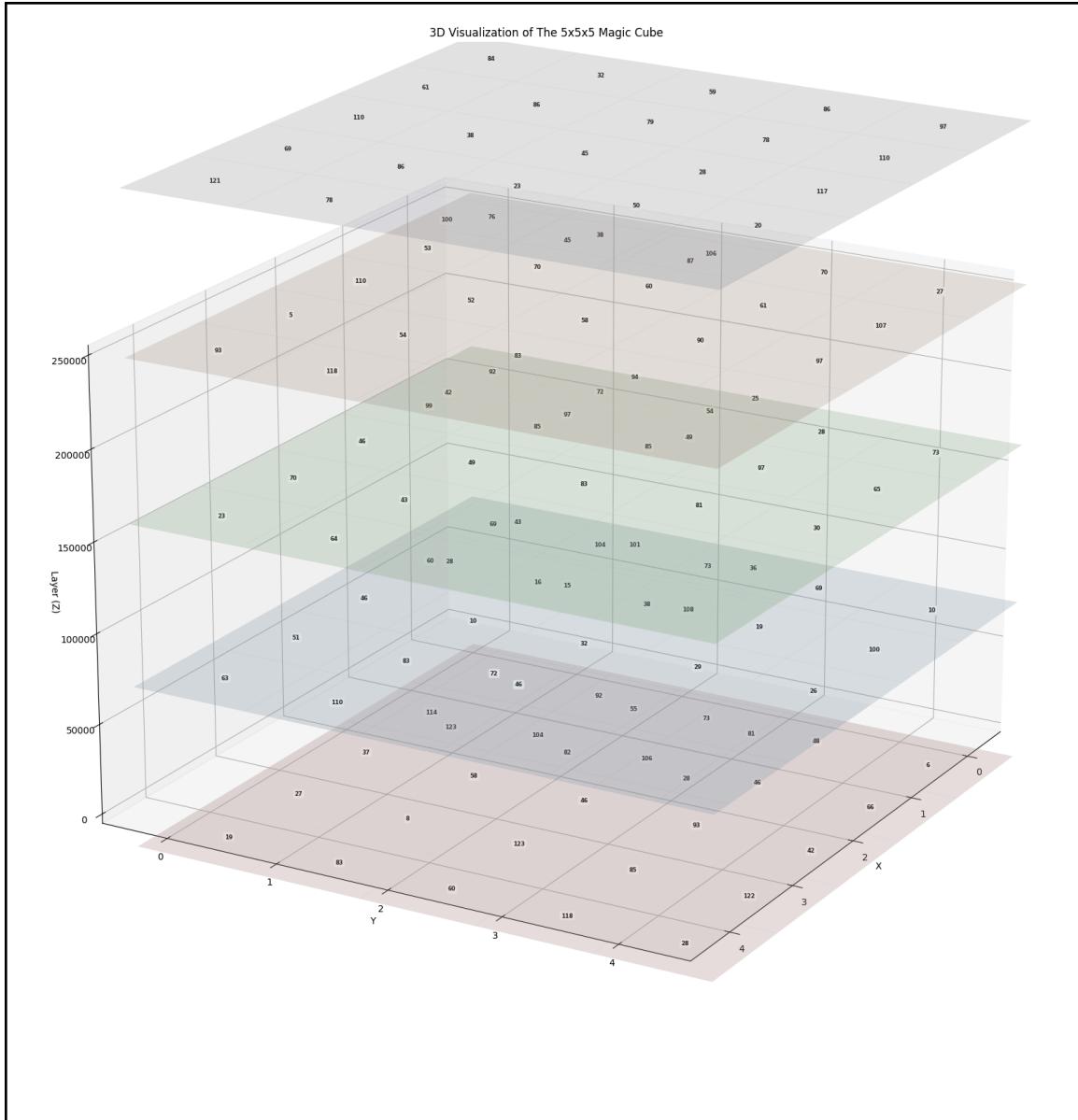
Variasi 1 (Iterasi: 100):

- Percobaan 1:

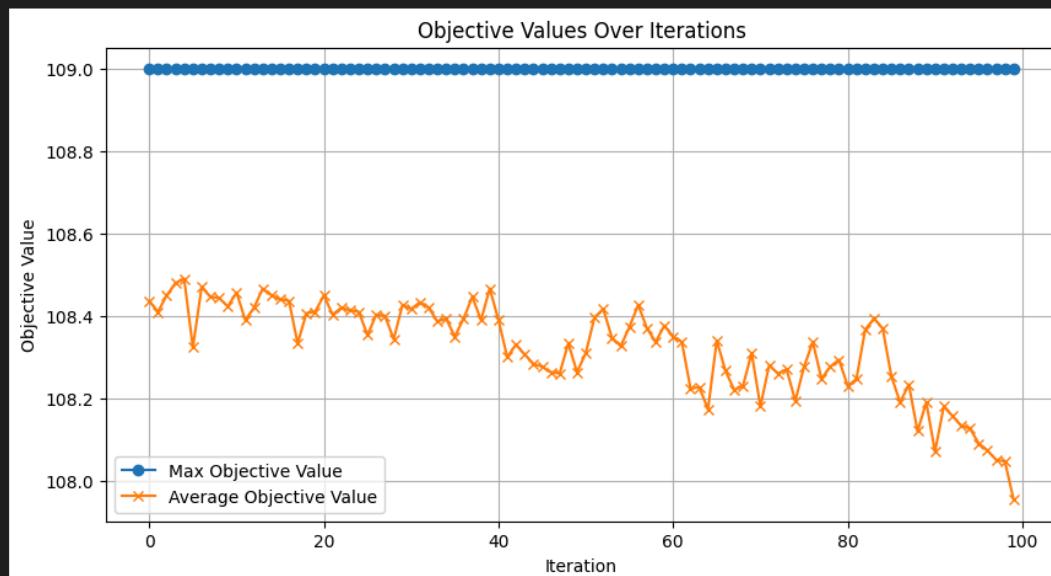
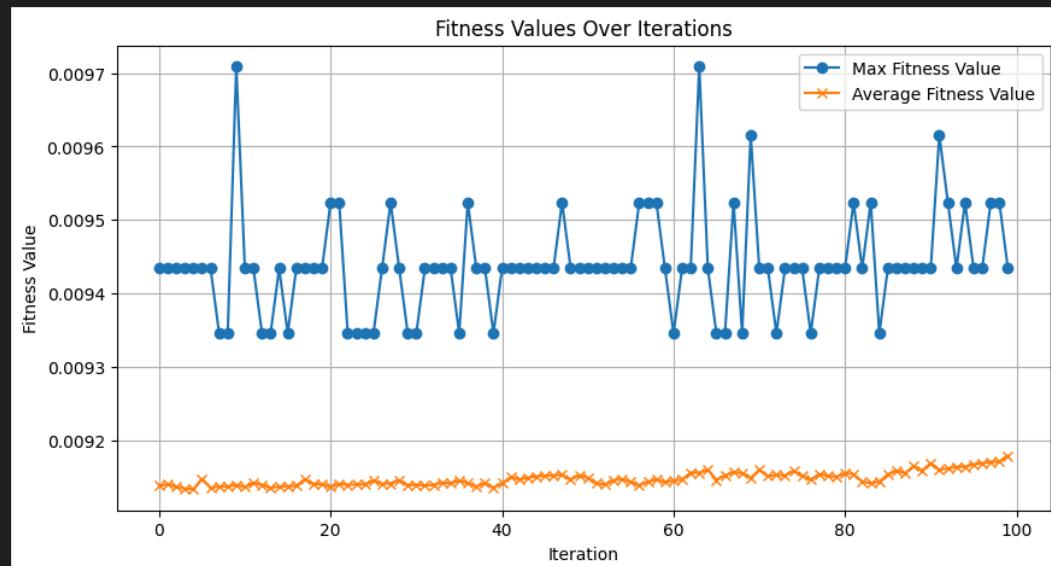


Final Objective Value: (6456, 108)
Population Size: 300
Iterations: 100
Duration: 35.30985236167908 seconds

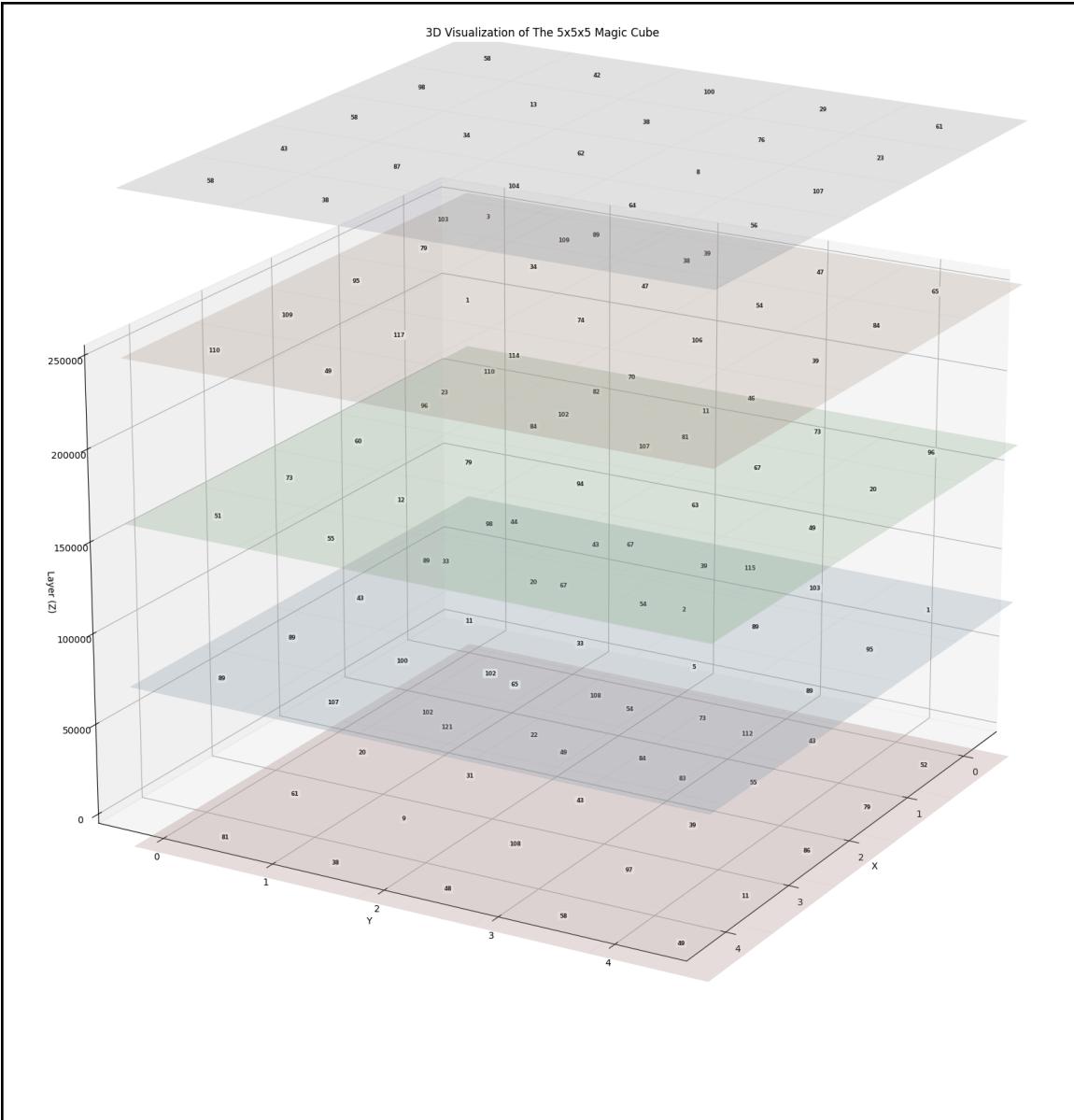




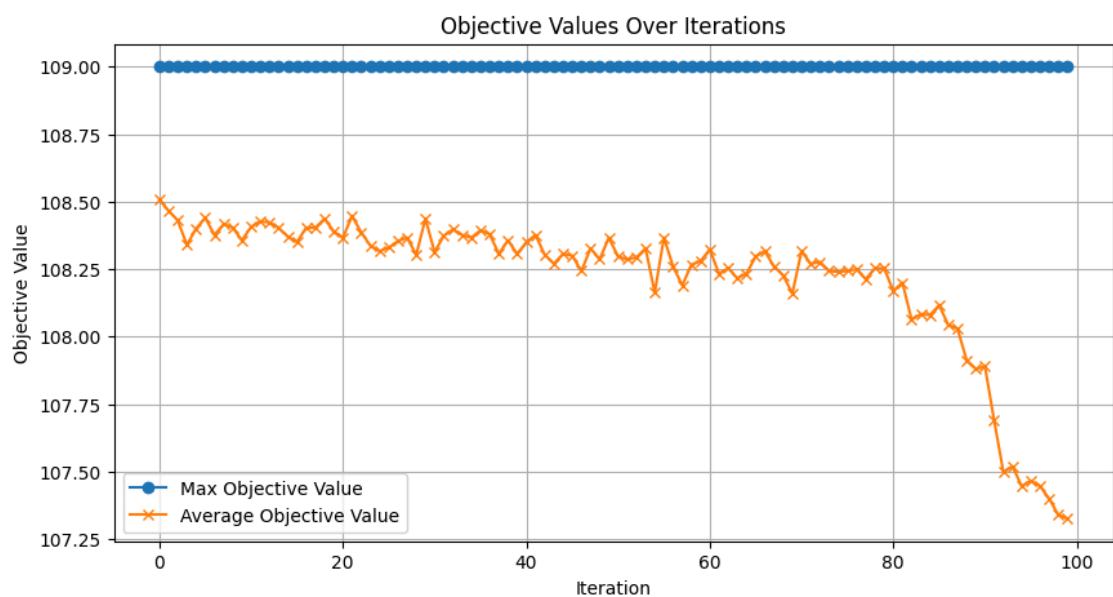
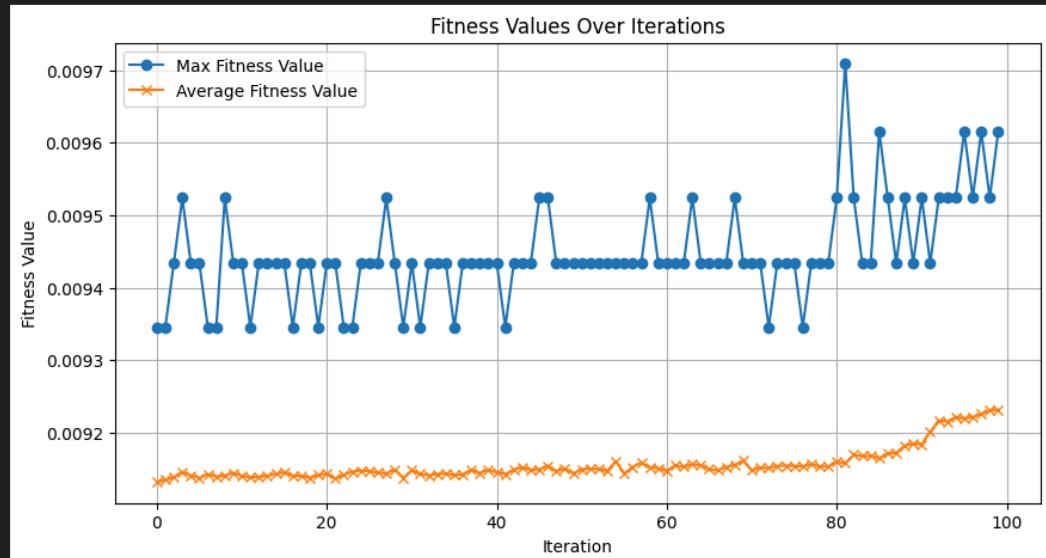
Final Objective Value: (5291, 109)
Population Size: 300
Iterations: 100
Duration: 35.58703589439392 seconds



- Percobaan 3:

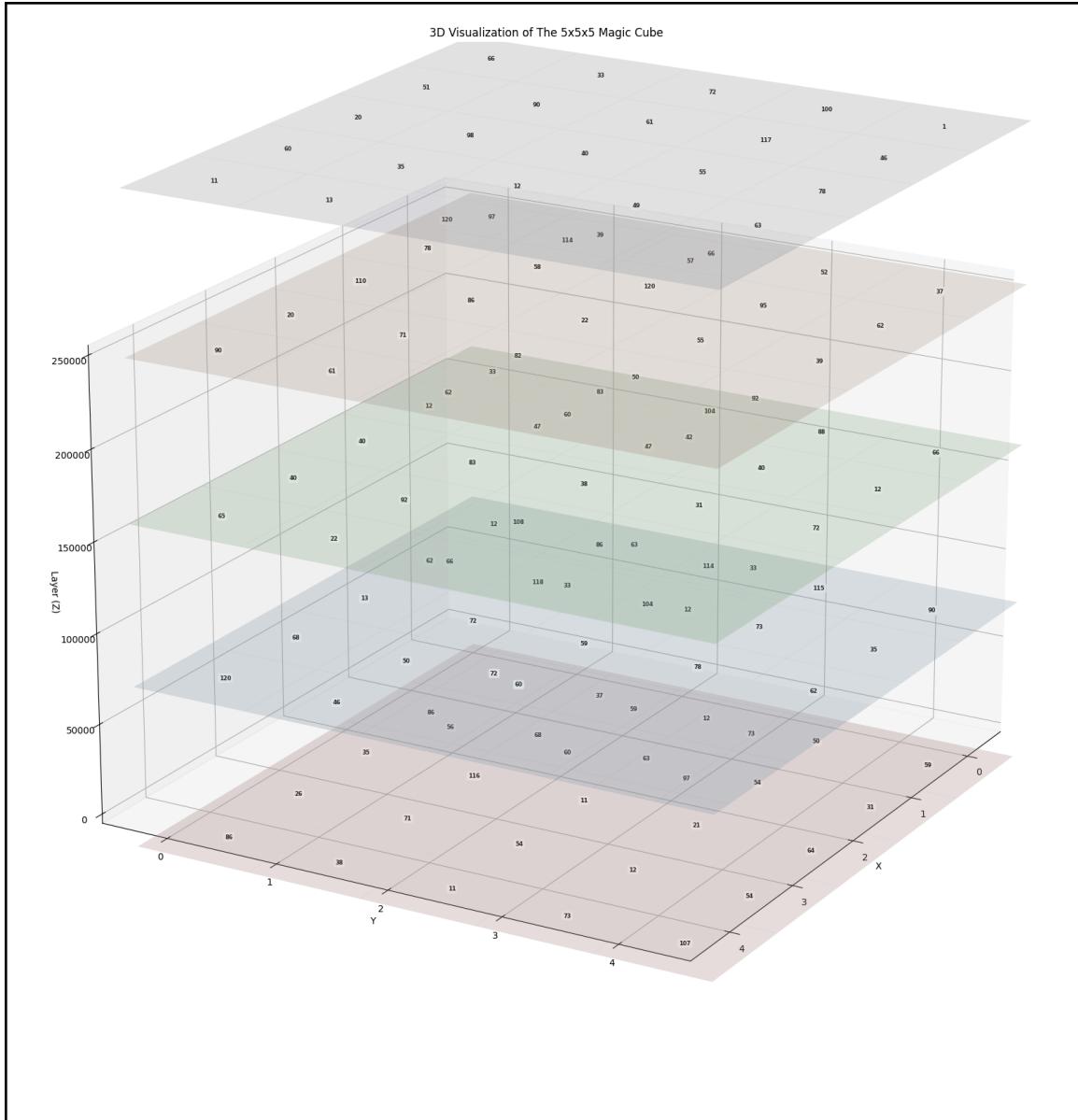


Final Objective Value: (5493, 106)
Population Size: 300
Iterations: 100
Duration: 35.26971697807312 seconds

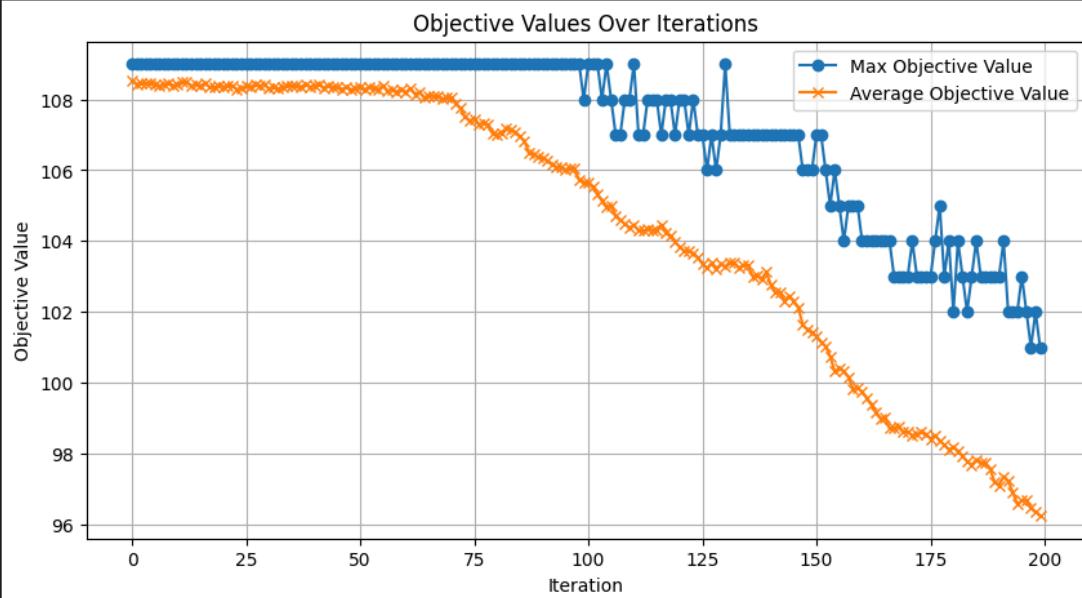
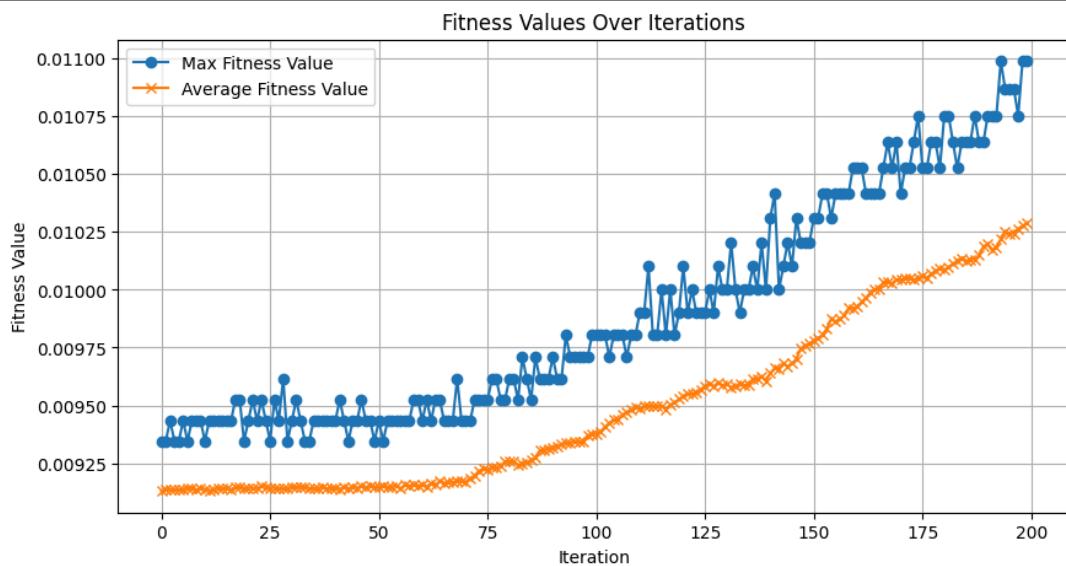


Variasi 2 (Iterasi: 200):

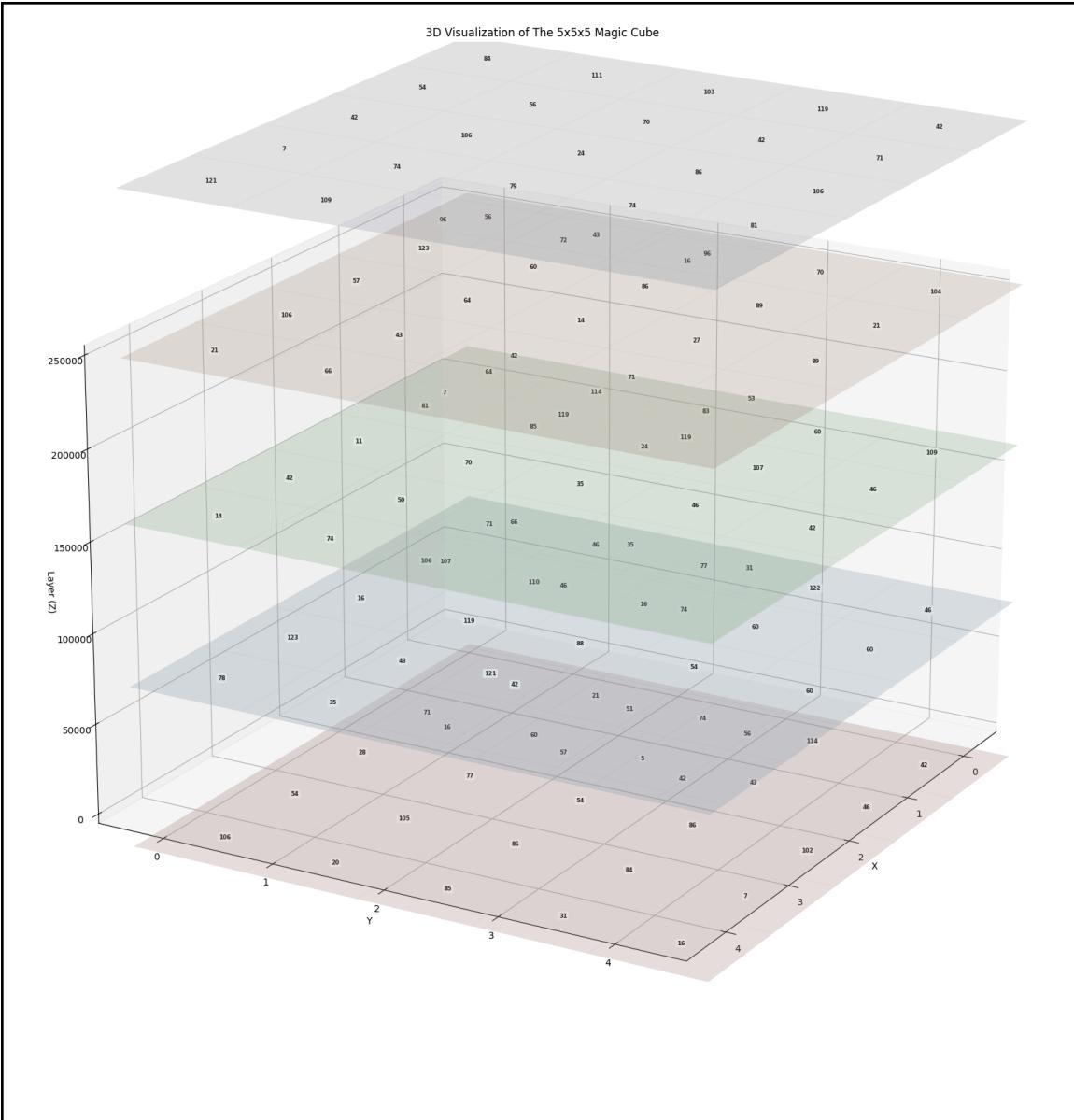
- Percobaan 1:



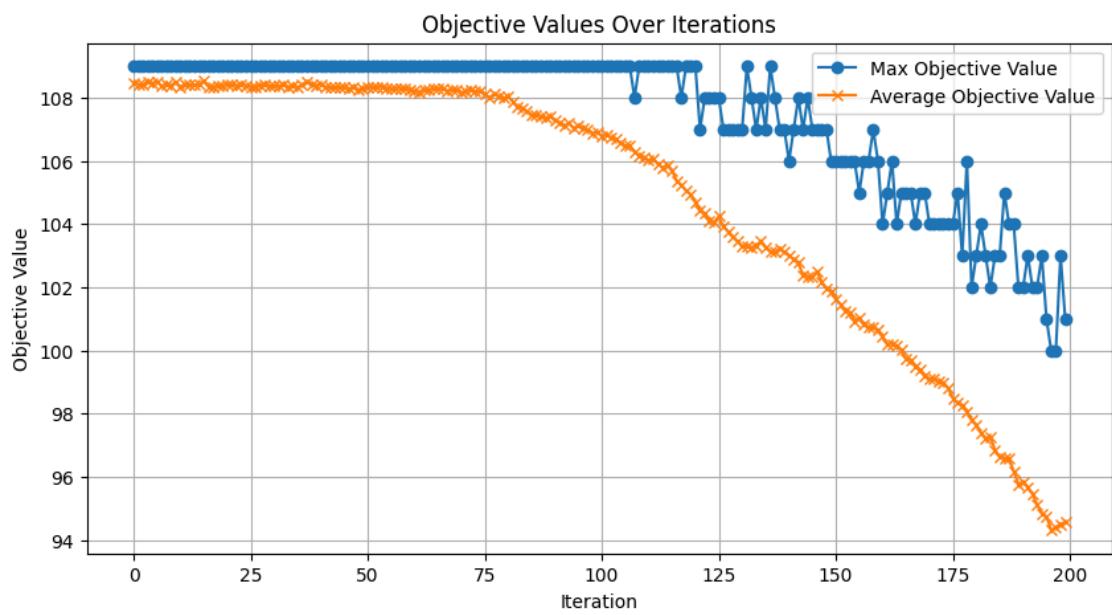
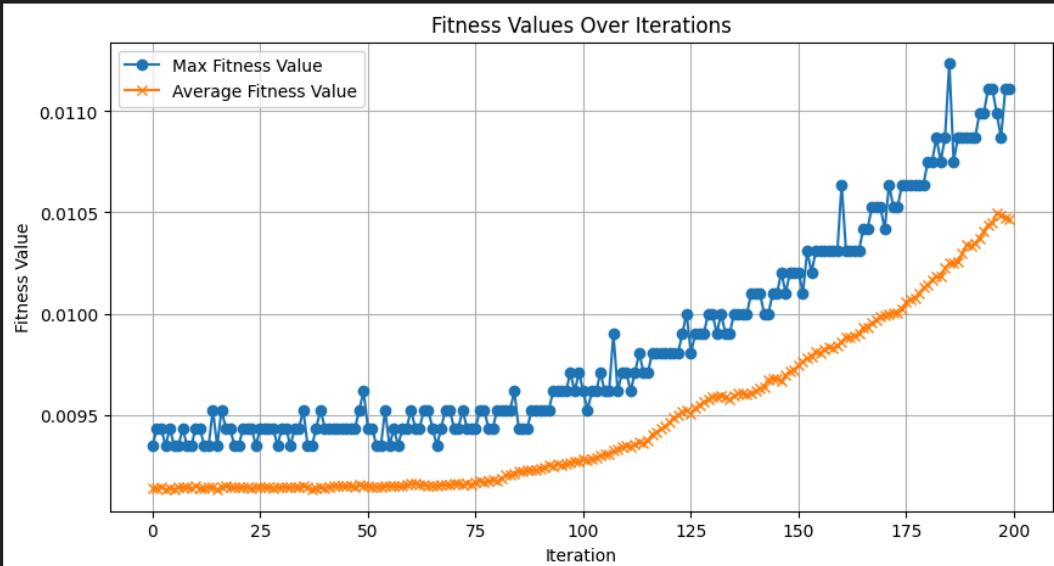
Final Objective Value: (5506, 93)
Population Size: 300
Iterations: 200
Duration: 74.99682998657227 seconds



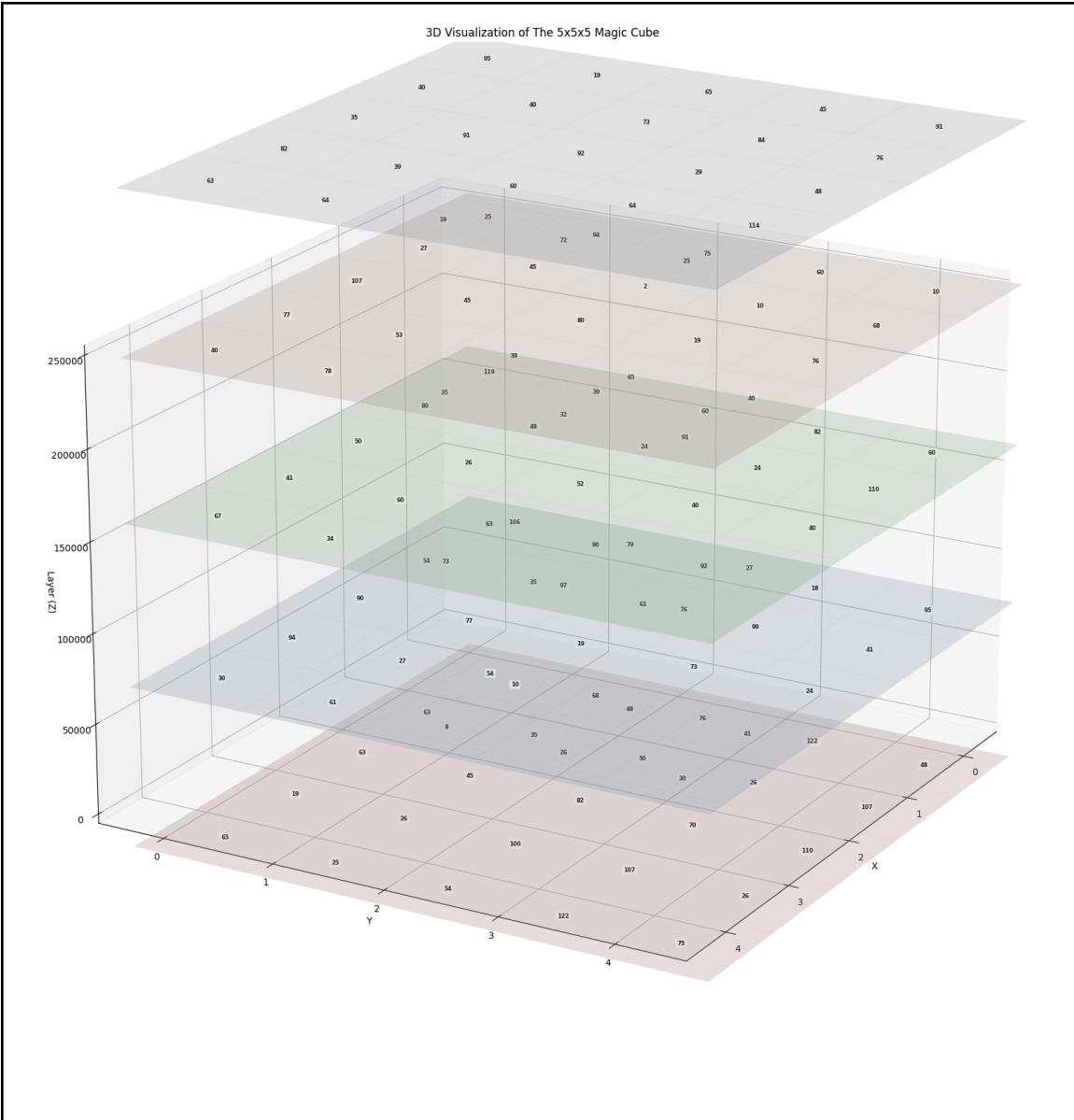
- Percobaan 2



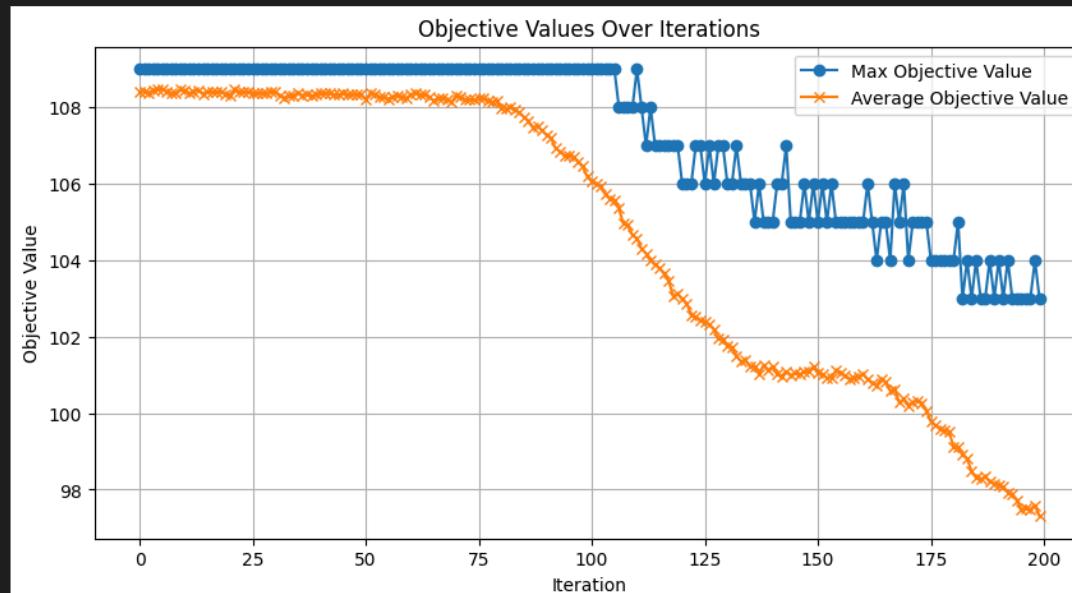
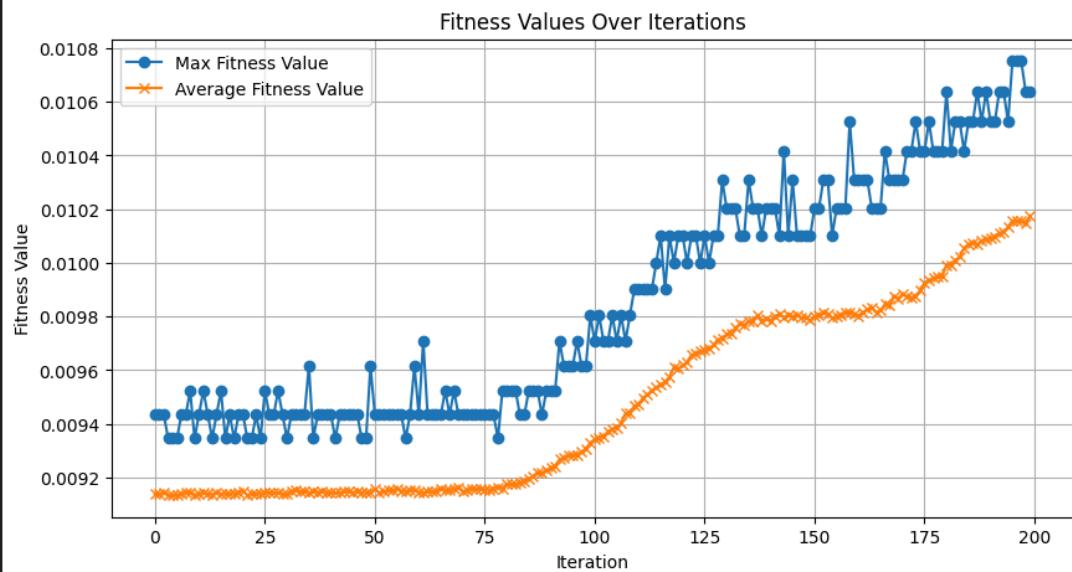
Final Objective Value: (5749, 96)
Population Size: 300
Iterations: 200
Duration: 72.3464150428772 seconds



- Percobaan 3

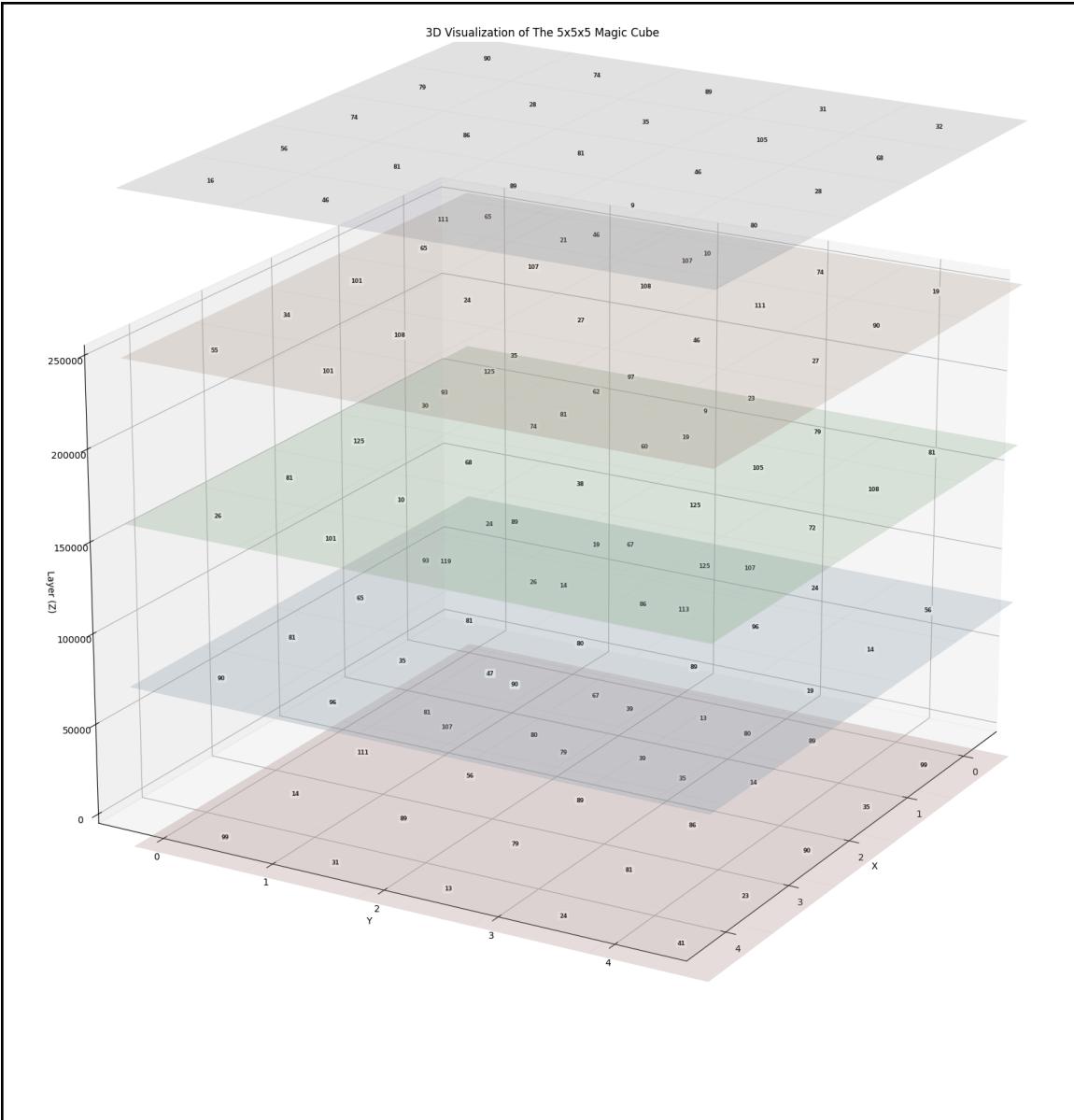


Final Objective Value: (5368, 96)
Population Size: 300
Iterations: 200
Duration: 71.380624294281 seconds

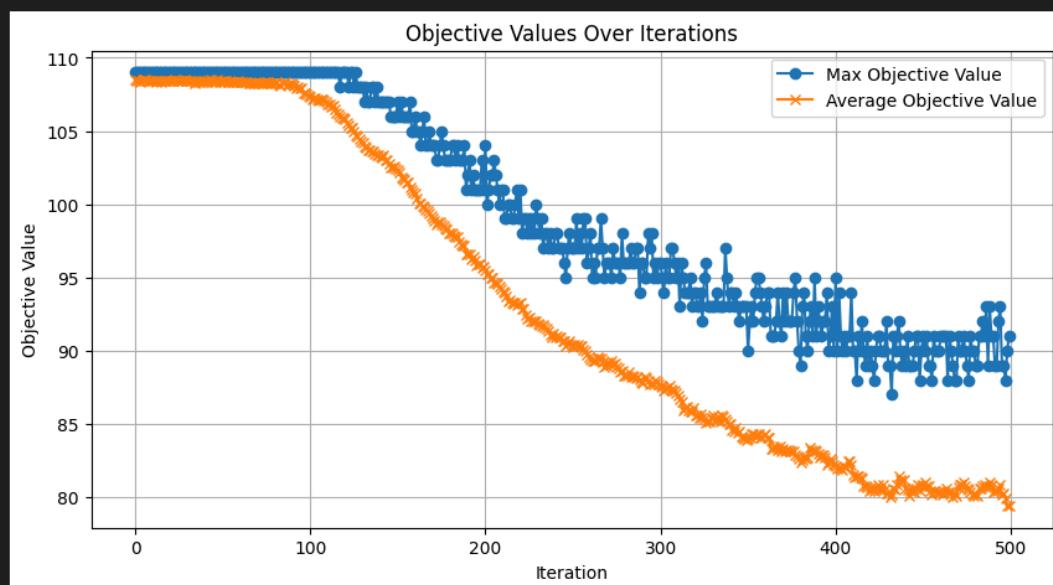
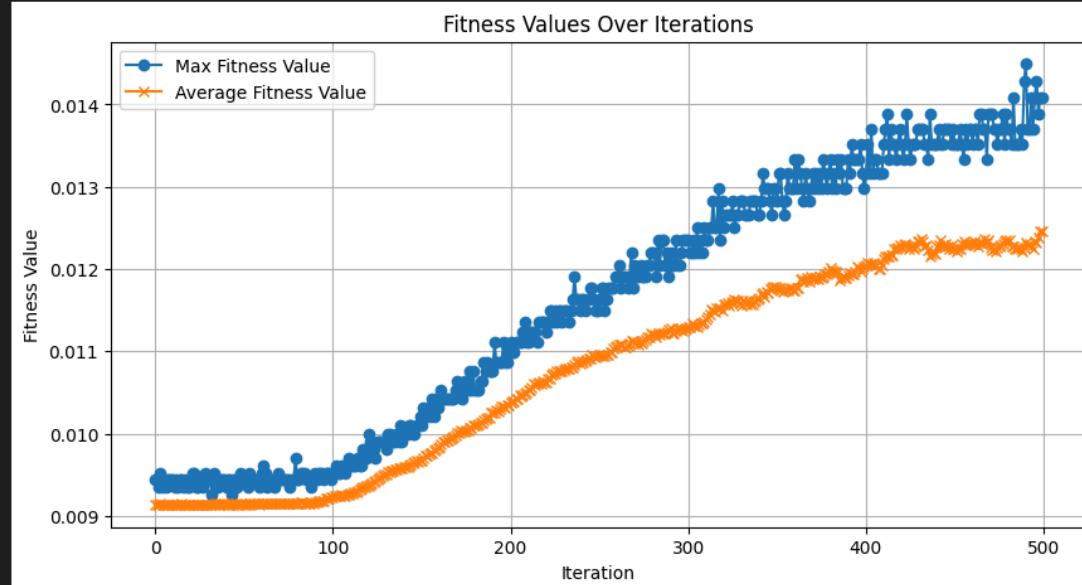


Variasi 3 (Iterasi: 500):

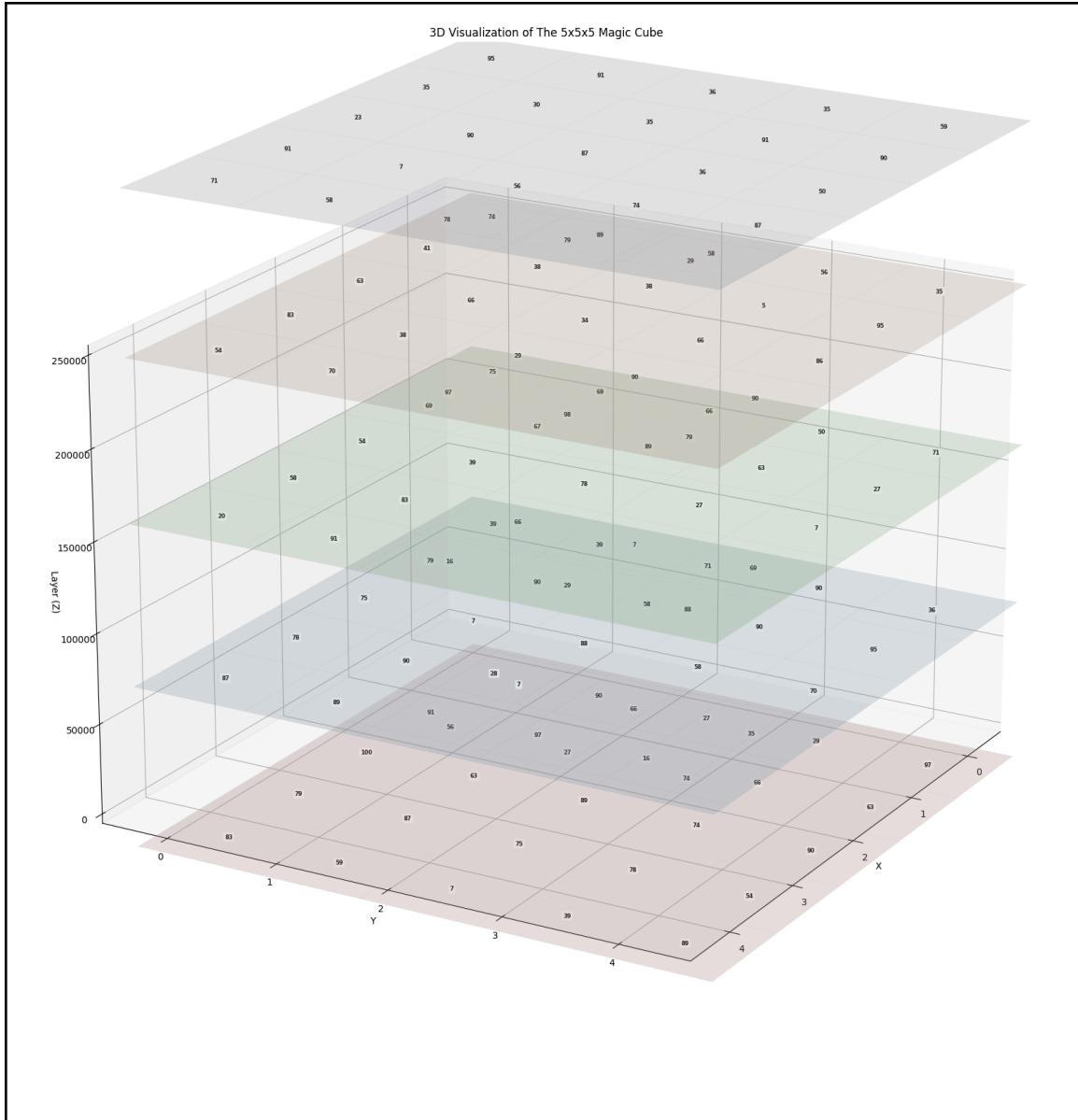
- Percobaan 1



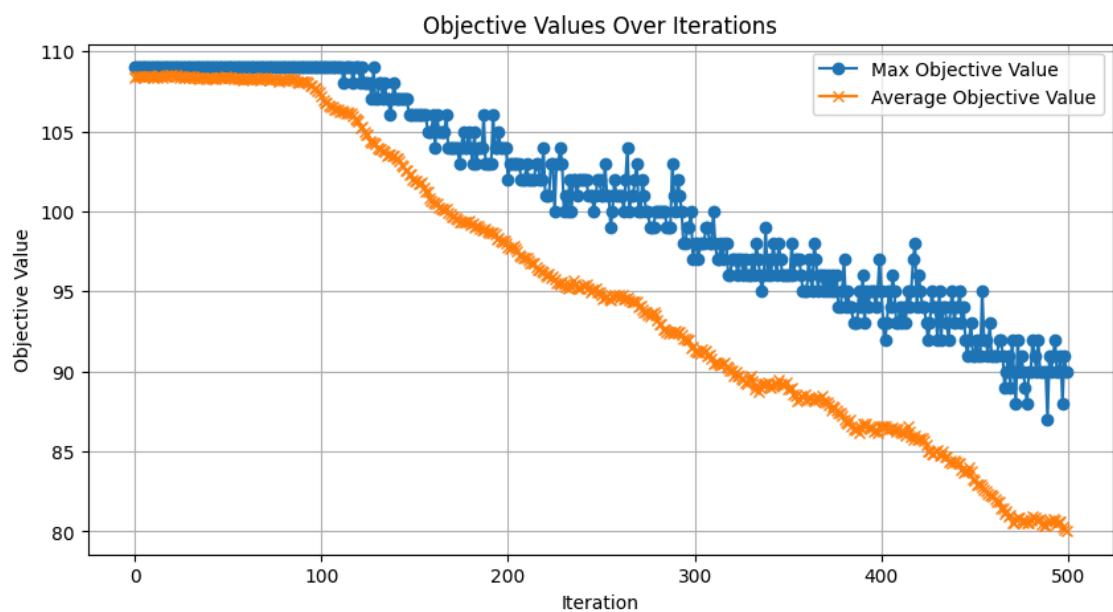
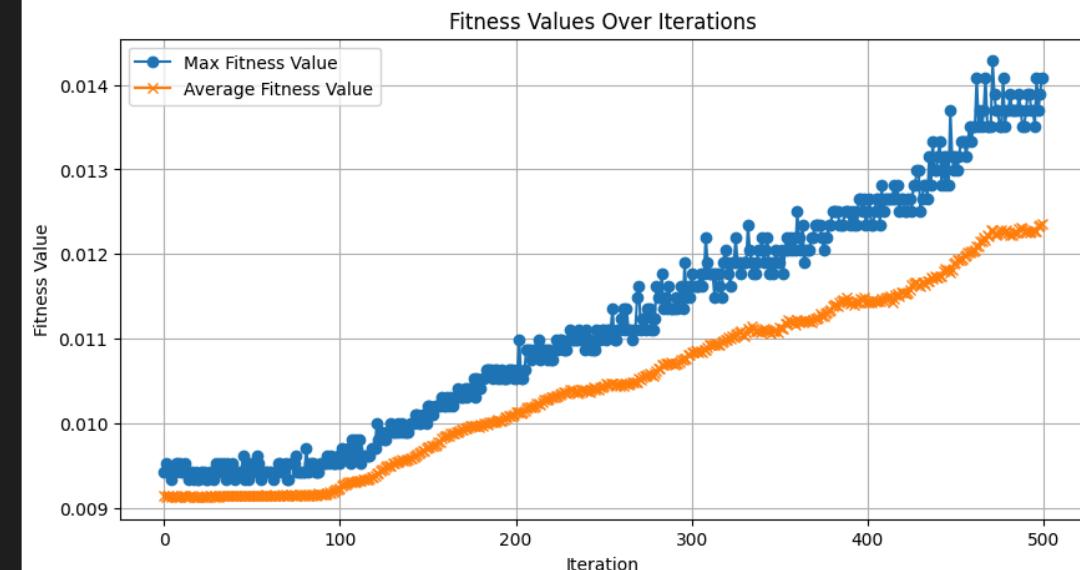
Final Objective Value: (5090, 80)
Population Size: 300
Iterations: 500
Duration: 182.03244352340698 seconds



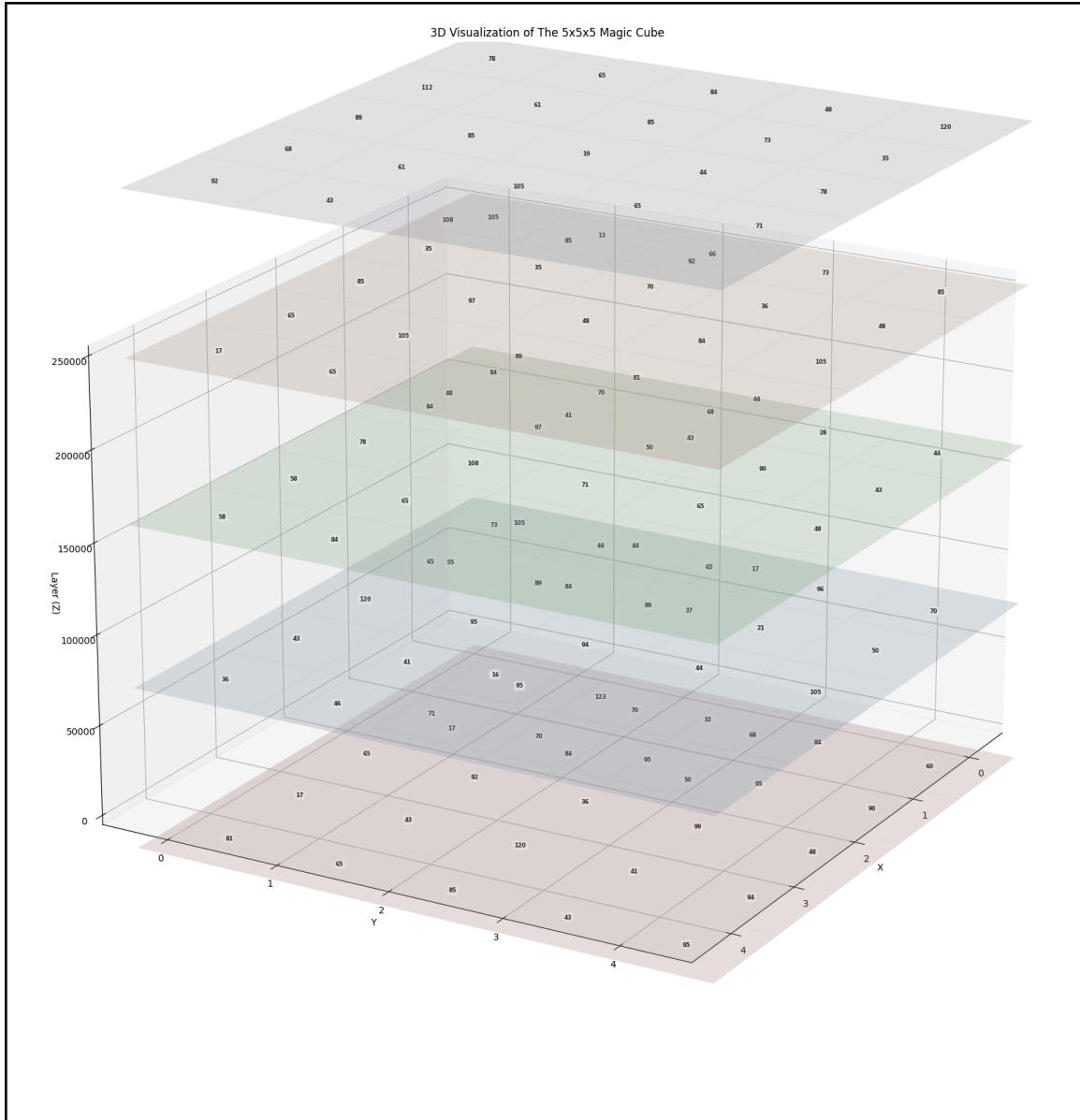
- Percobaan 2:



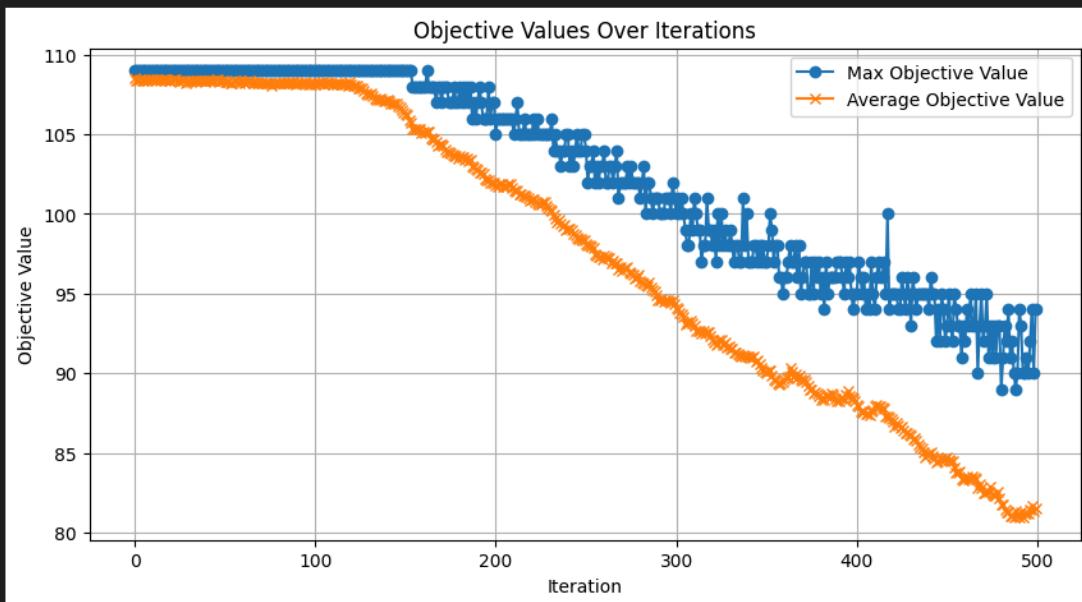
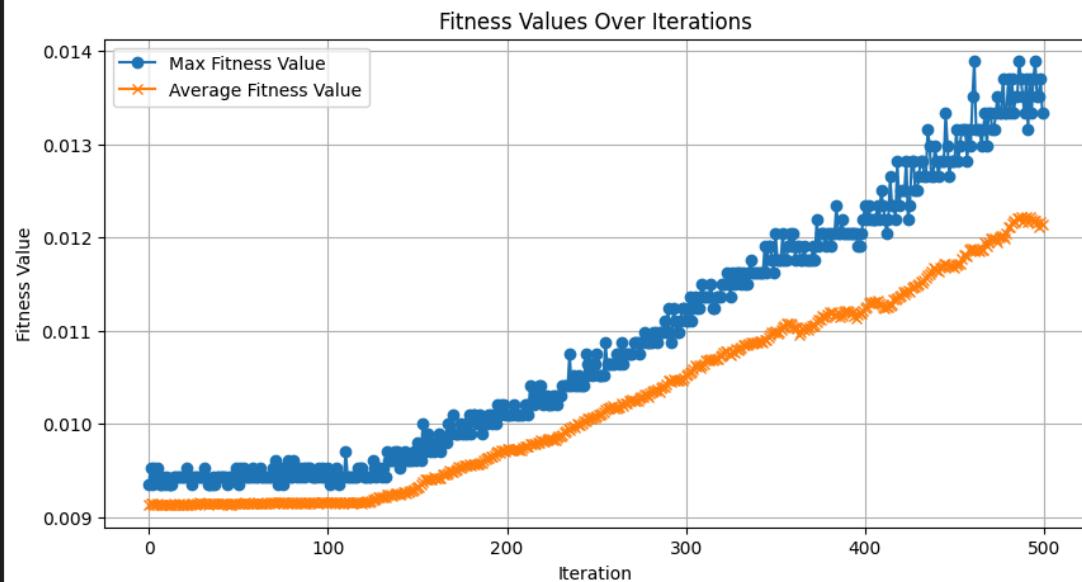
Final Objective Value: (4106, 79)
Population Size: 300
Iterations: 500
Duration: 179.811776638031 seconds



- Percobaan 3:



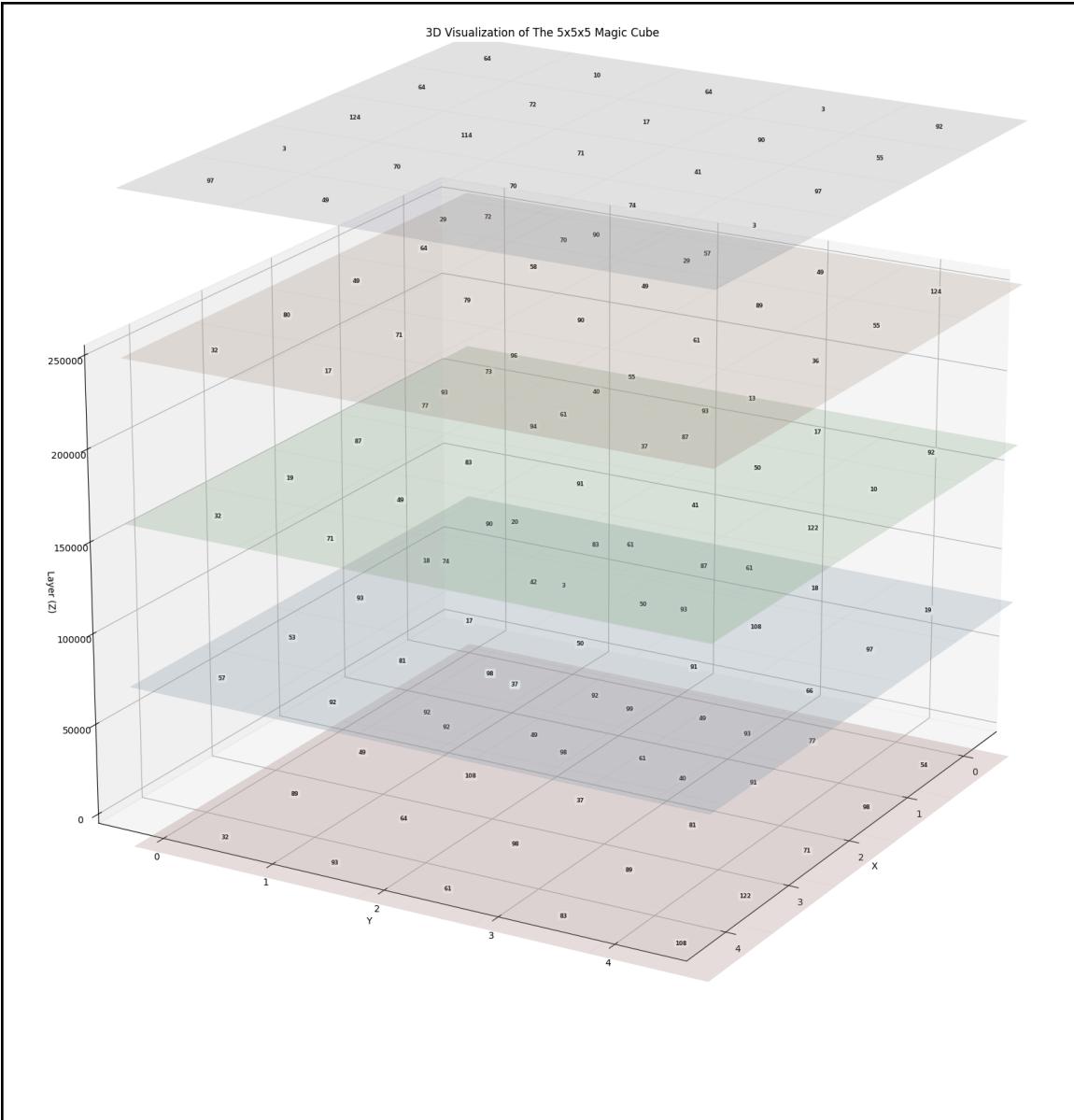
Final Objective Value: (4702, 90)
Population Size: 300
Iterations: 500
Duration: 189.2927680015564 seconds



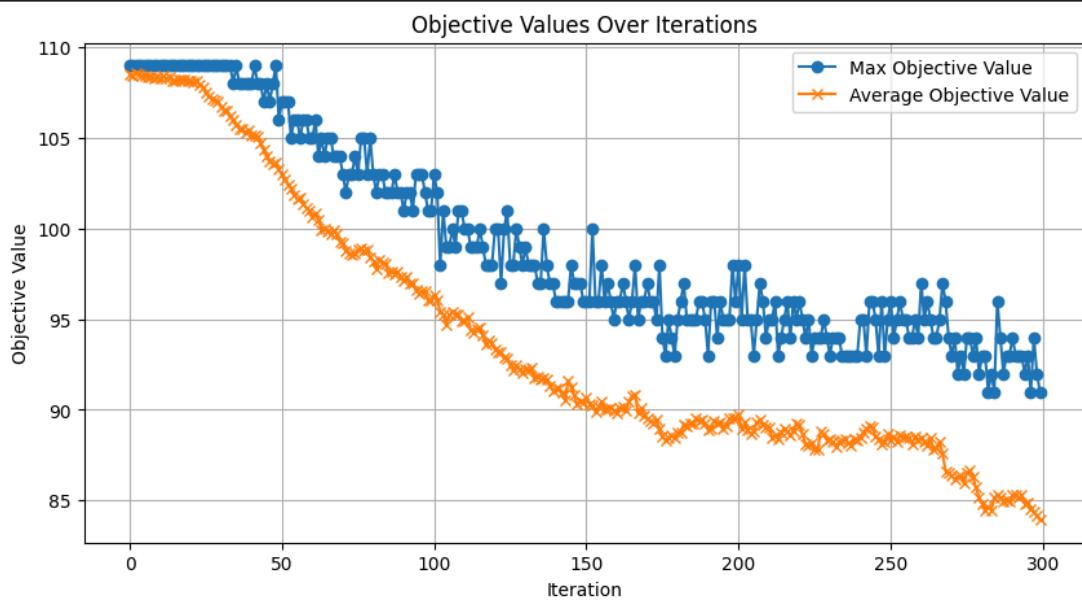
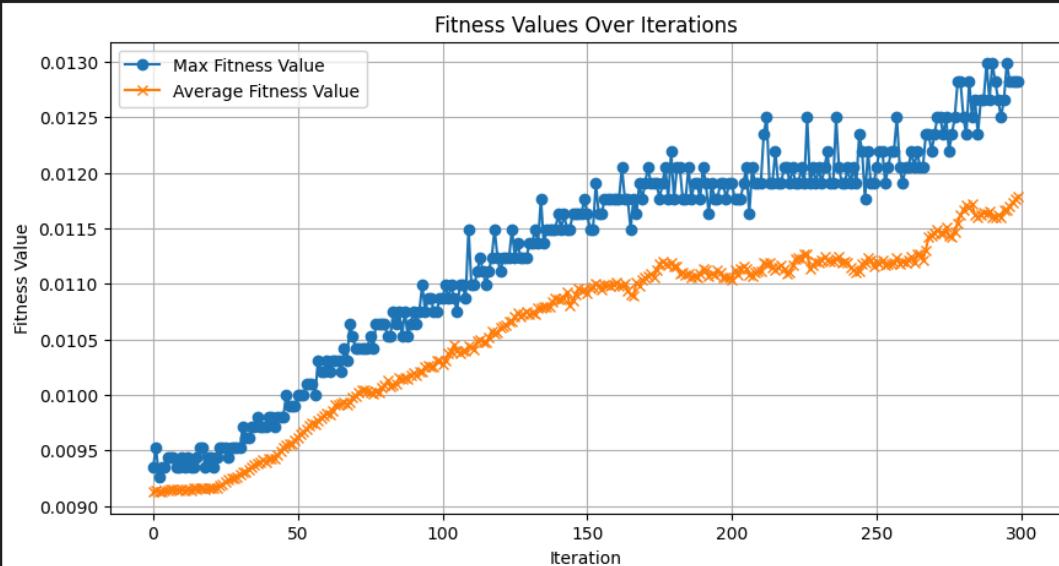
Iterasi sebagai kontrol (Iterasi: 300)

Variasi 1 (Populasi: 100):

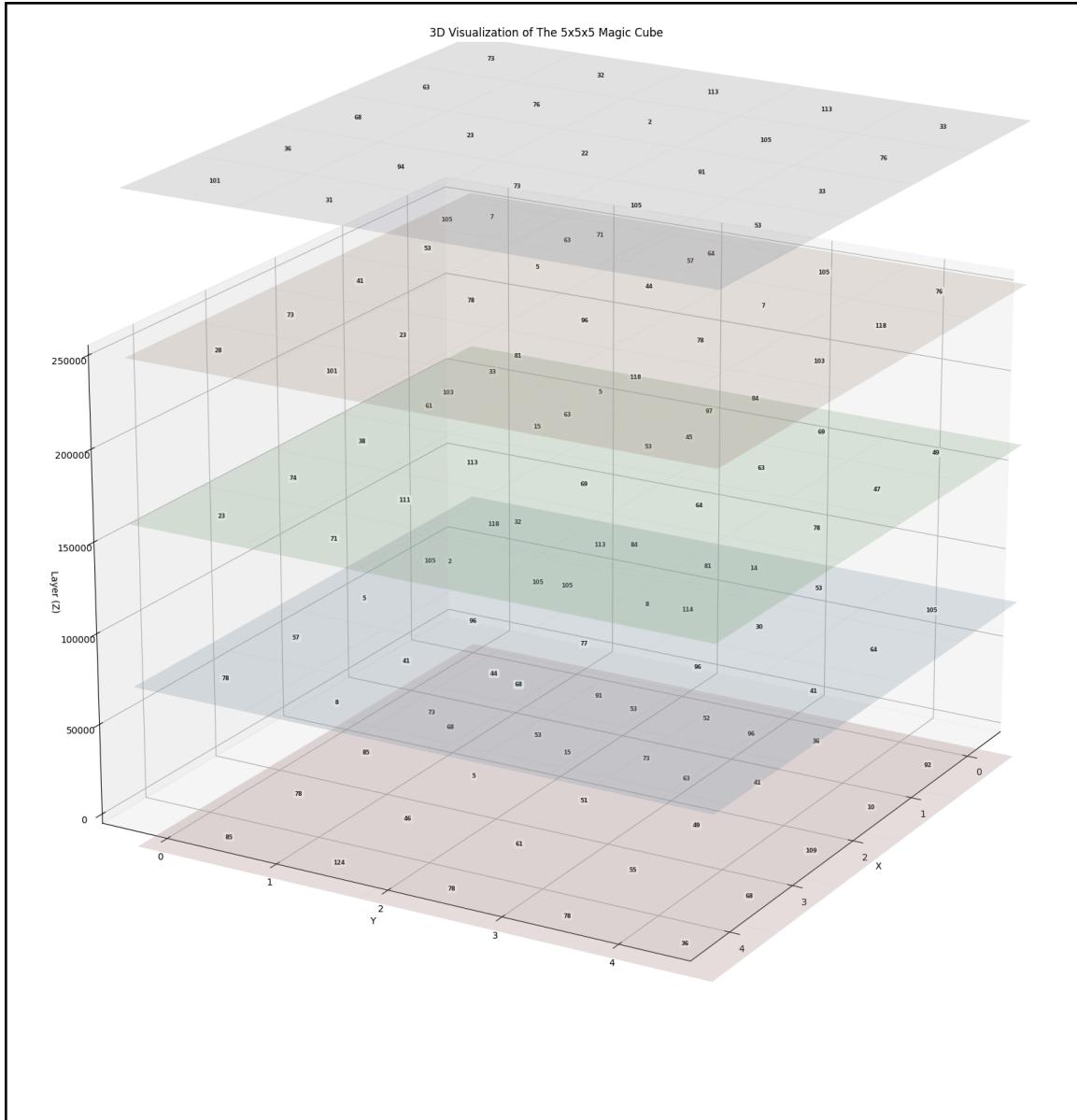
- Percobaan 1:



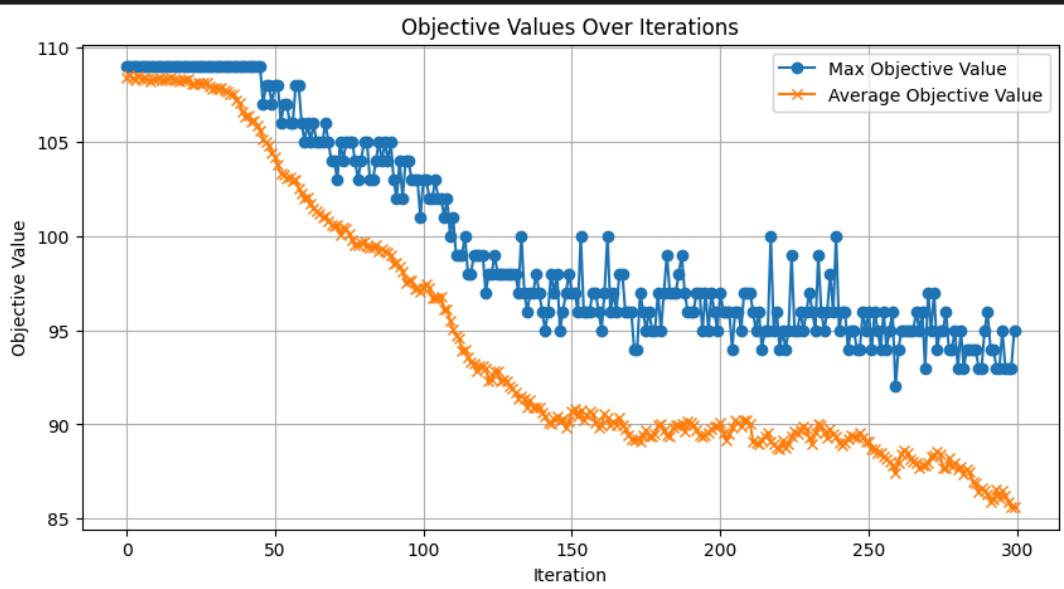
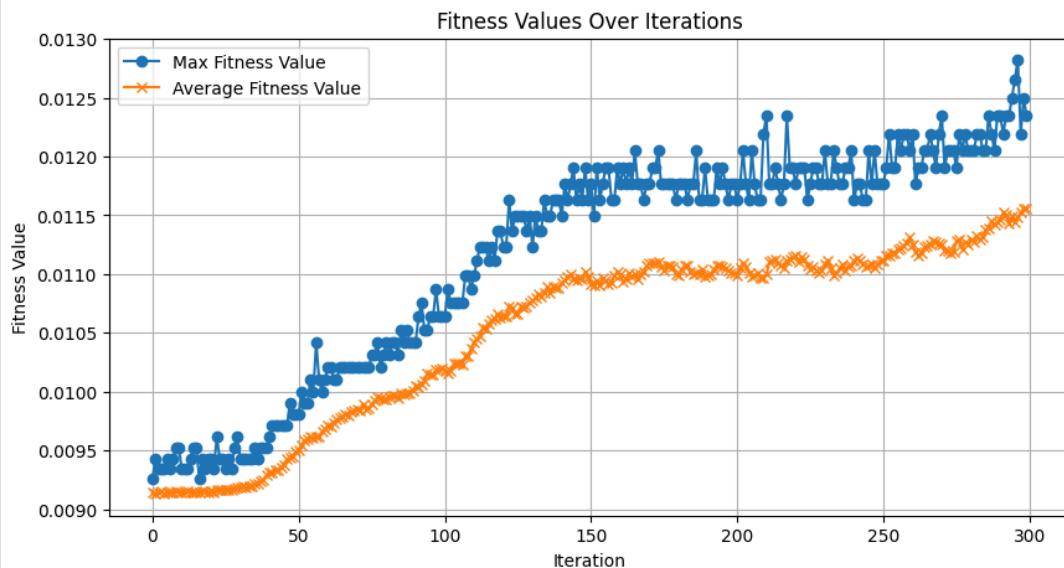
Final Objective Value: (4882, 83)
Population Size: 100
Iterations: 300
Duration: 30.973894119262695 seconds



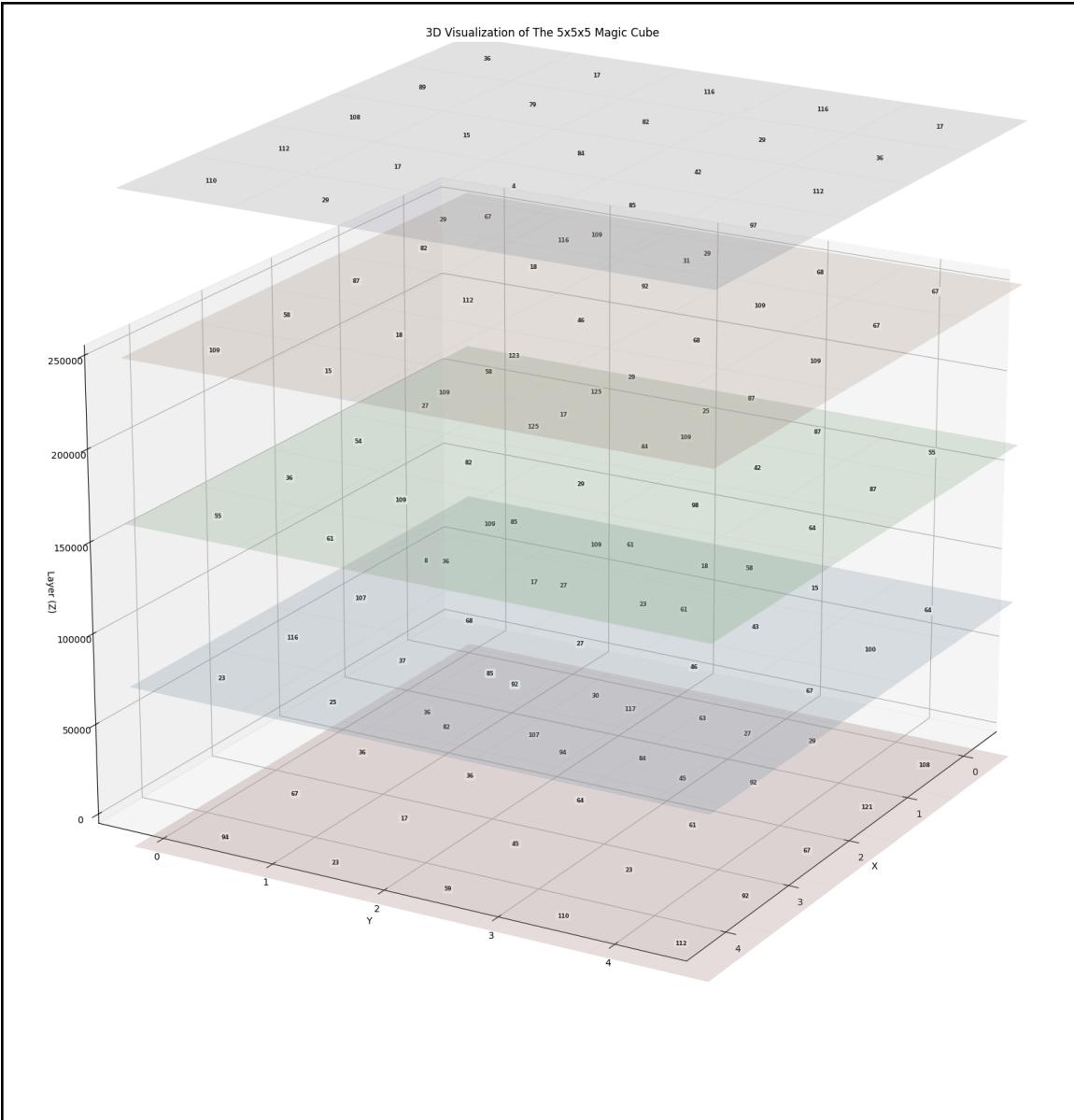
- Percobaan 2:



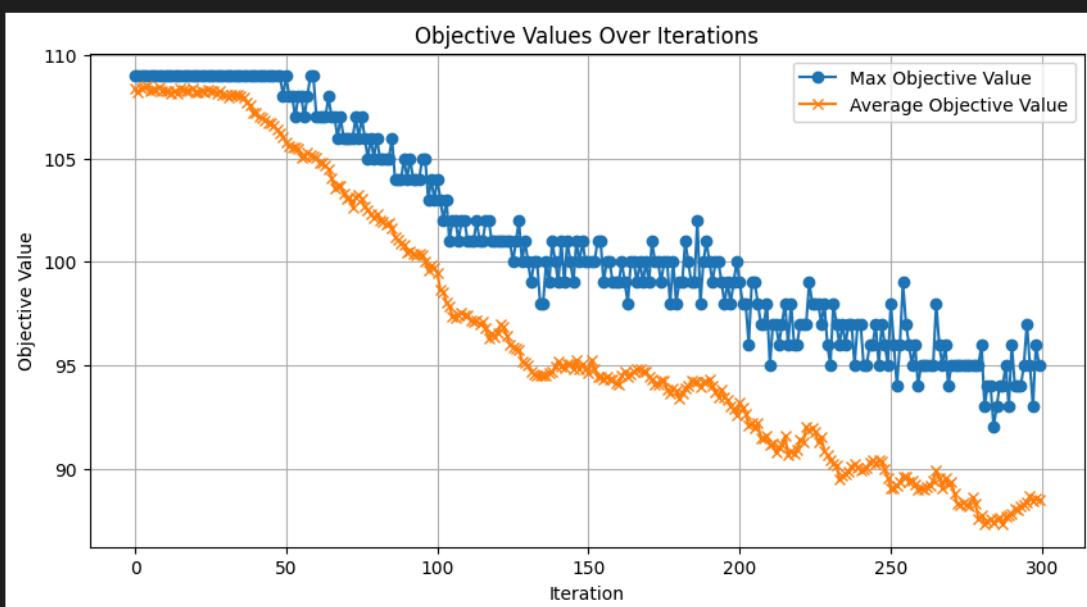
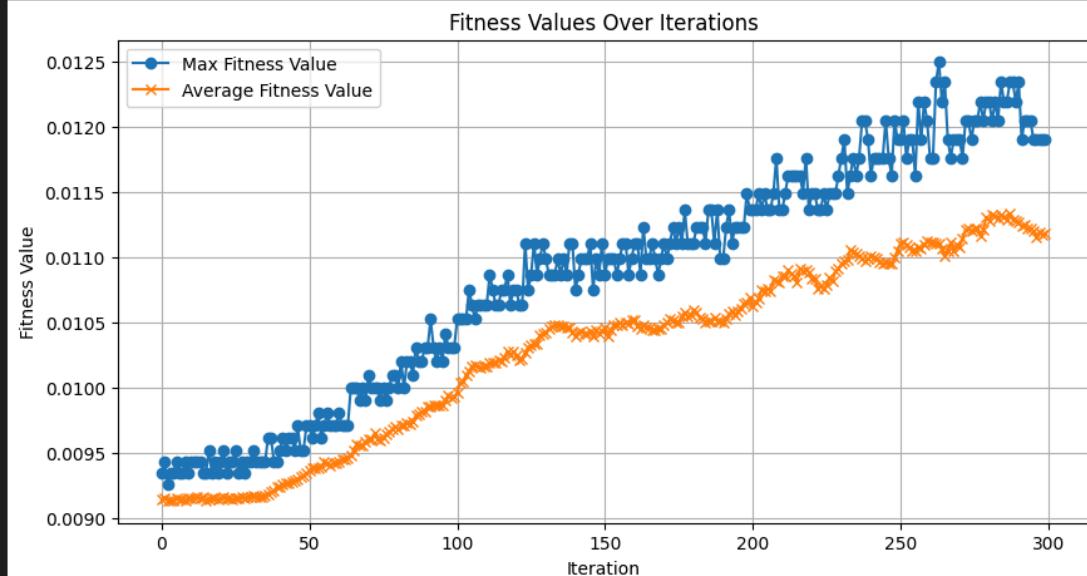
```
Final Objective Value: (4896, 85)
Population Size: 100
Iterations: 300
Duration: 31.694042205810547 seconds
```



- Percobaan 3:

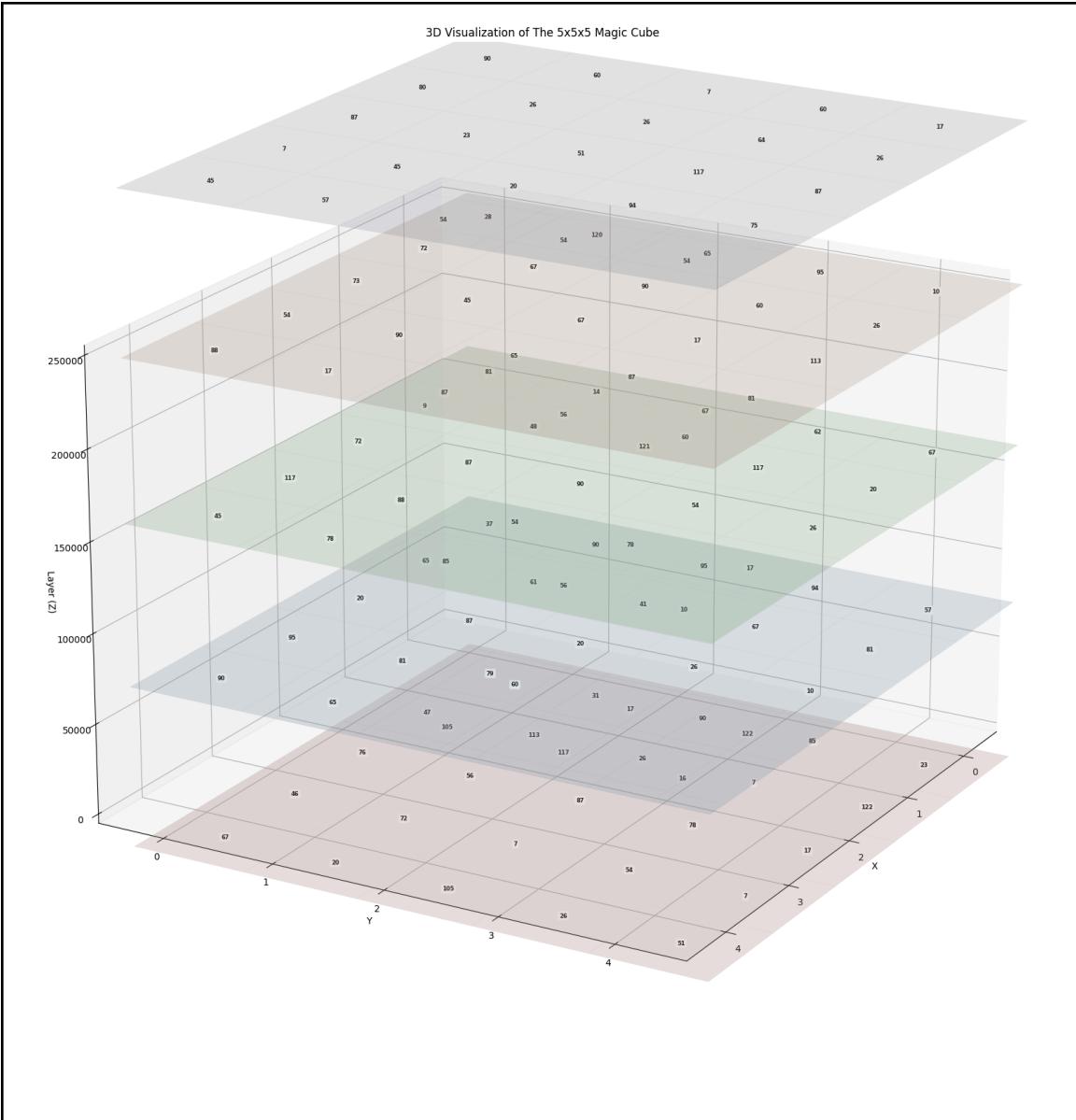


Final Objective Value: (5613, 86)
Population Size: 100
Iterations: 300
Duration: 31.291903495788574 seconds

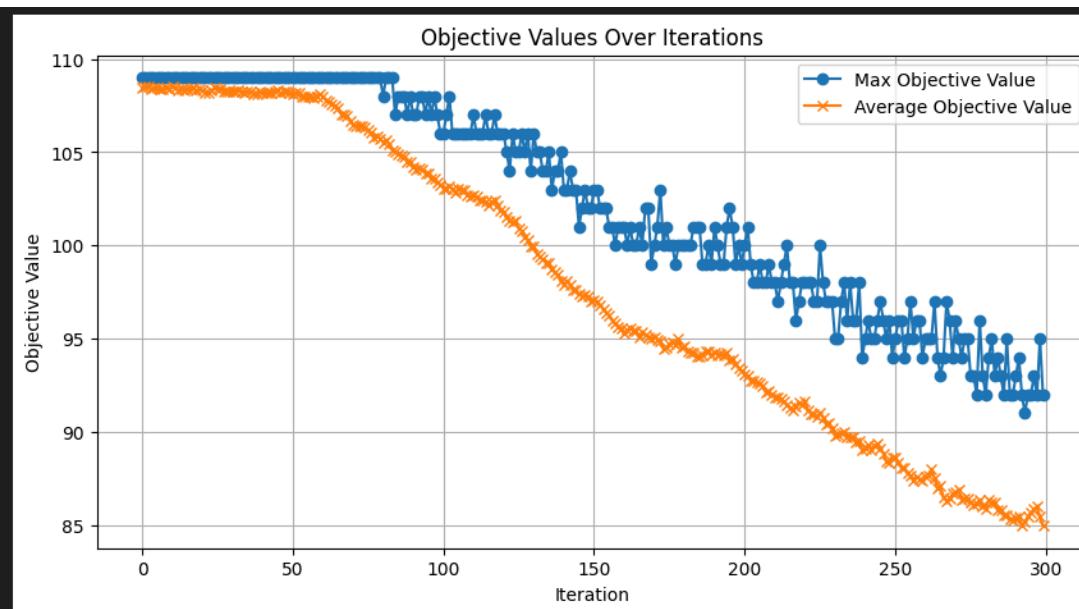
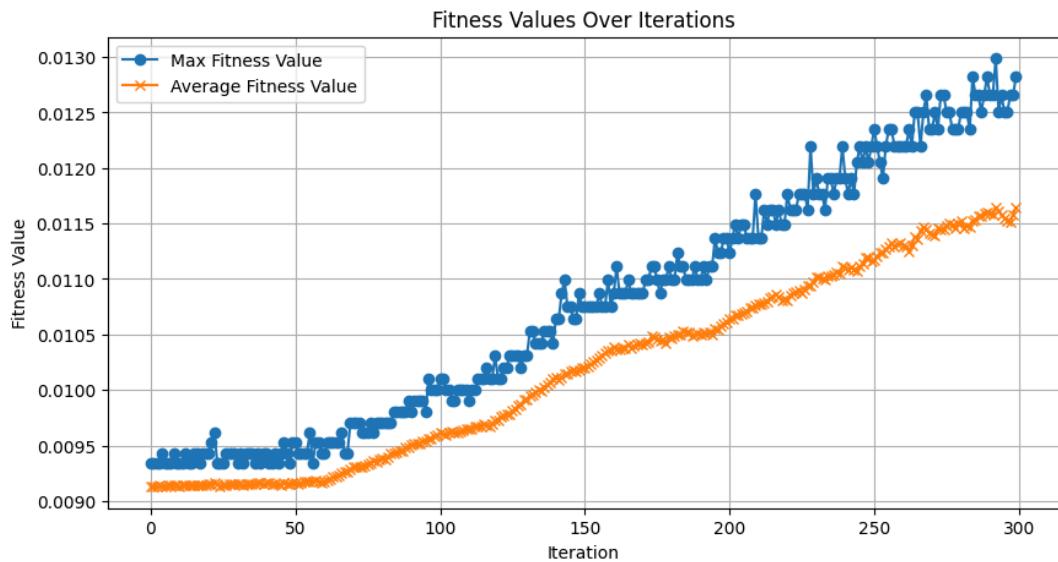


Variasi 2 (Populasi: 200):

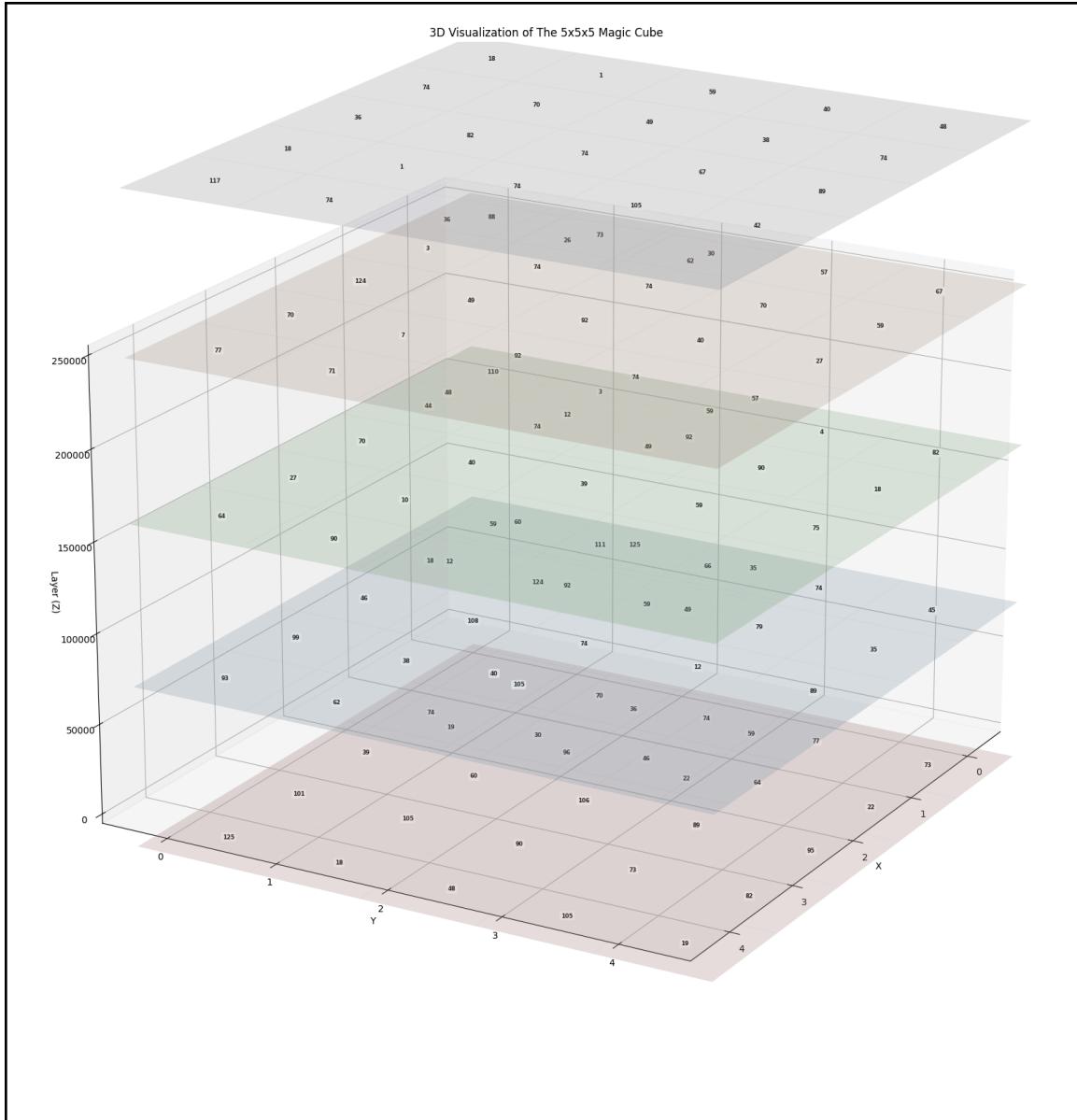
- Percobaan 1:



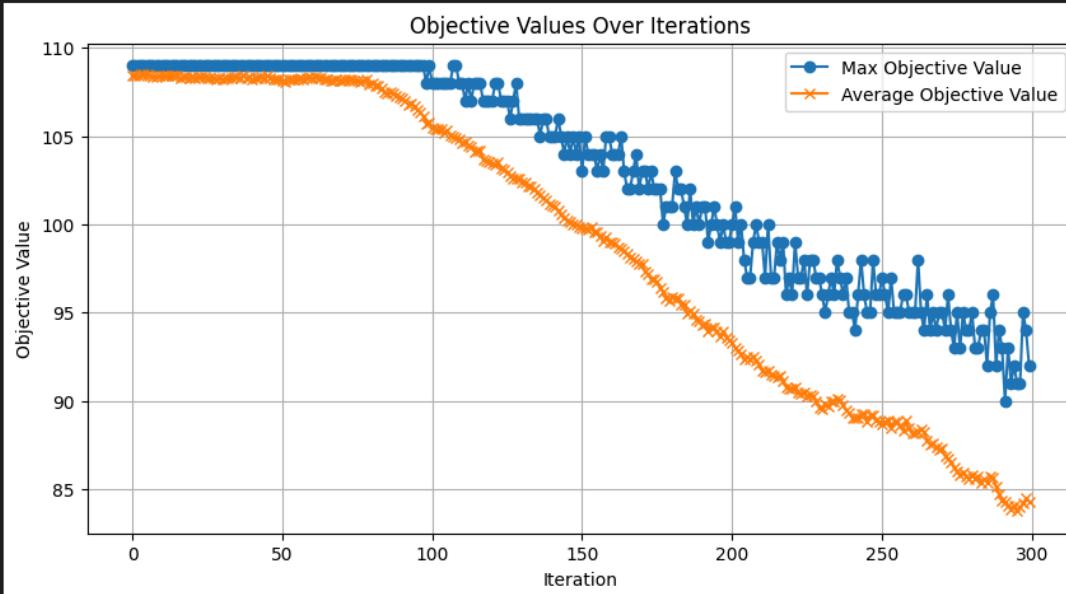
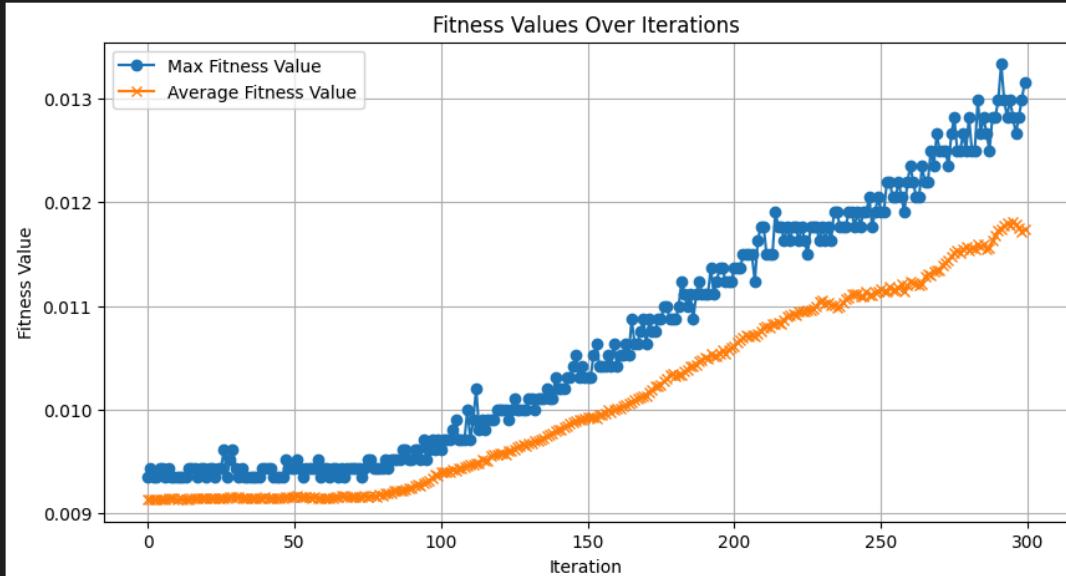
Final Objective Value: (4509, 83)
Population Size: 200
Iterations: 300
Duration: 72.58274579048157 seconds



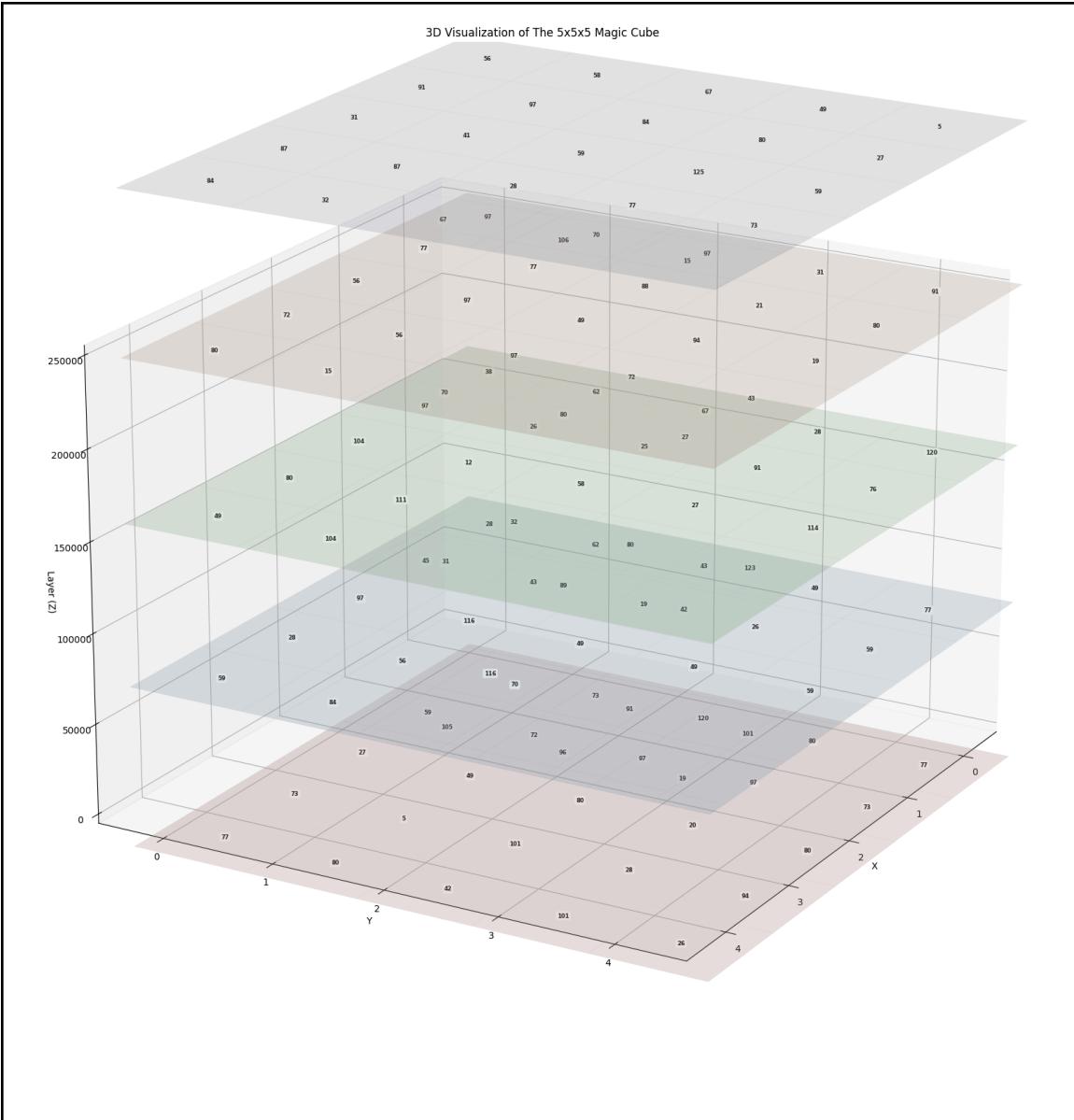
- Percobaan 2:



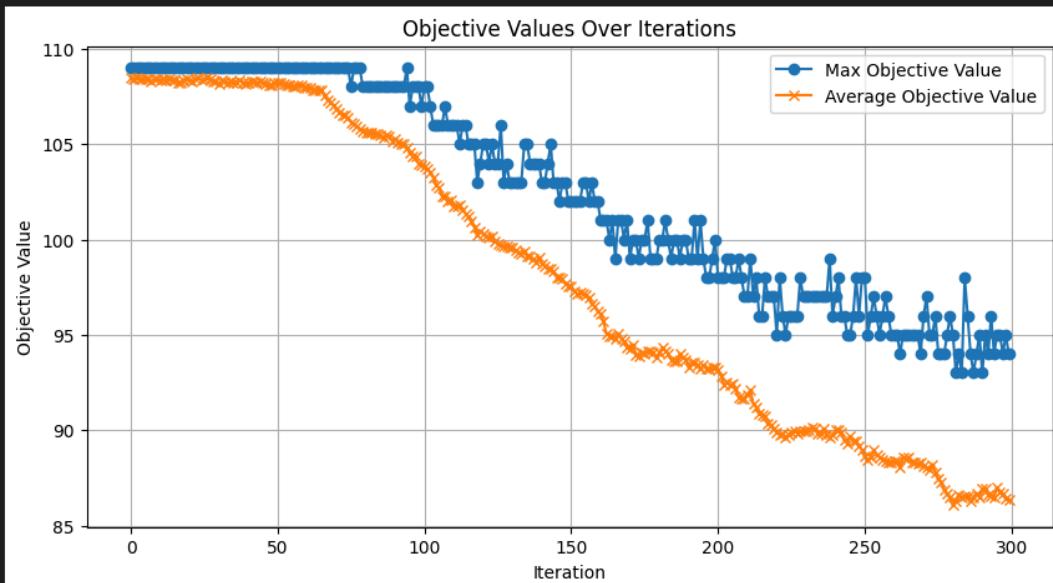
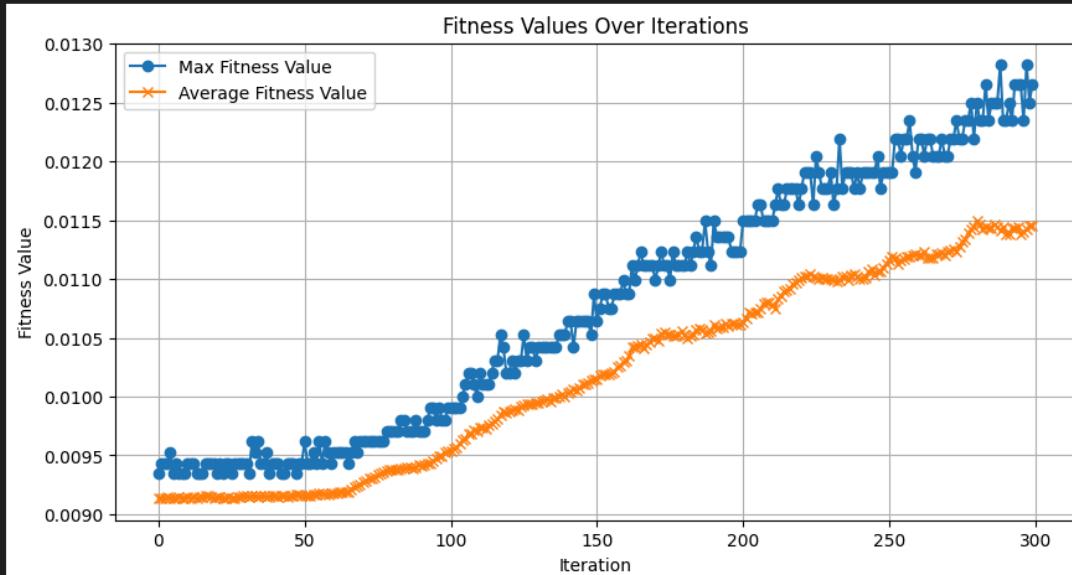
Final Objective Value: (4737, 87)
Population Size: 200
Iterations: 300
Duration: 63.62735366821289 seconds



- Percobaan 3:

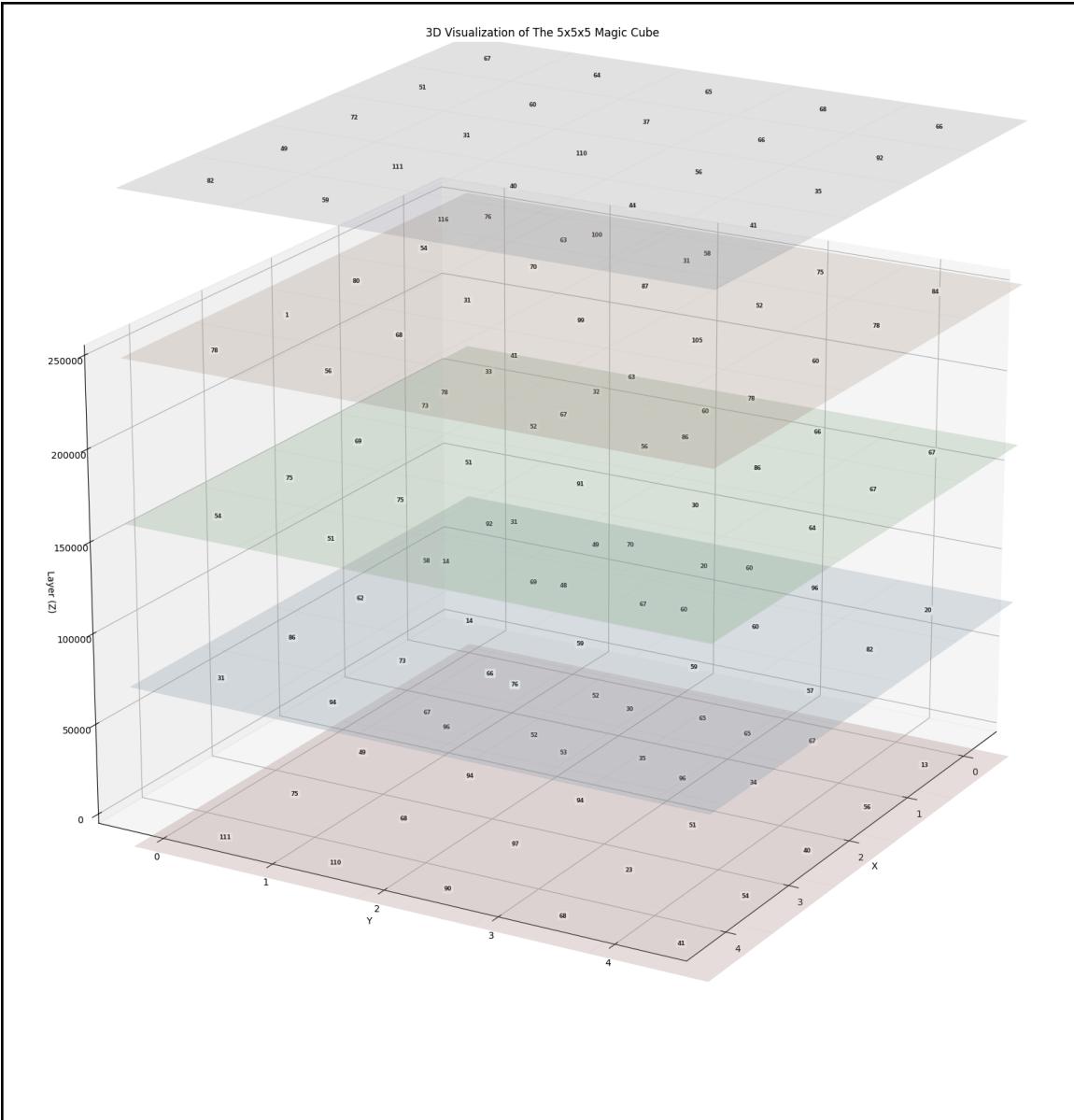


Final Objective Value: (4333, 86)
Population Size: 200
Iterations: 300
Duration: 74.75563335418701 seconds

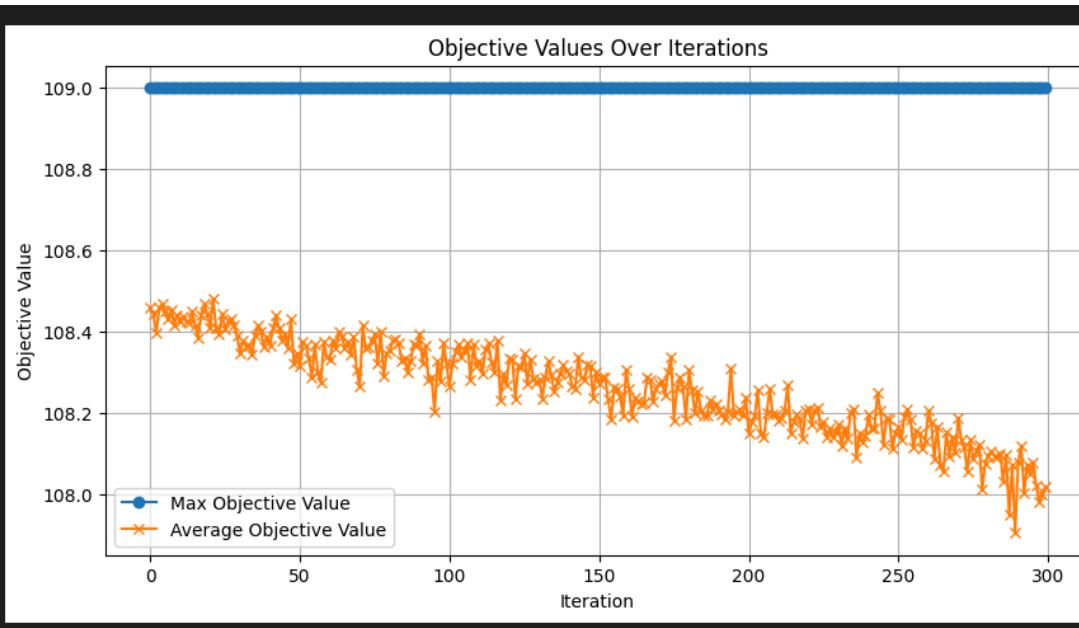
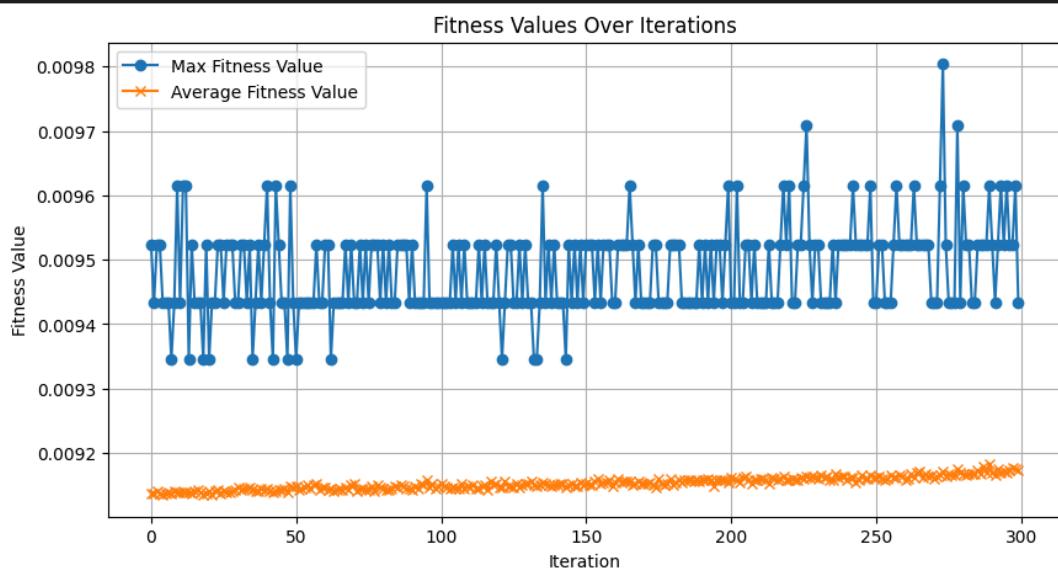


Variasi 3 (Populasi: 500):

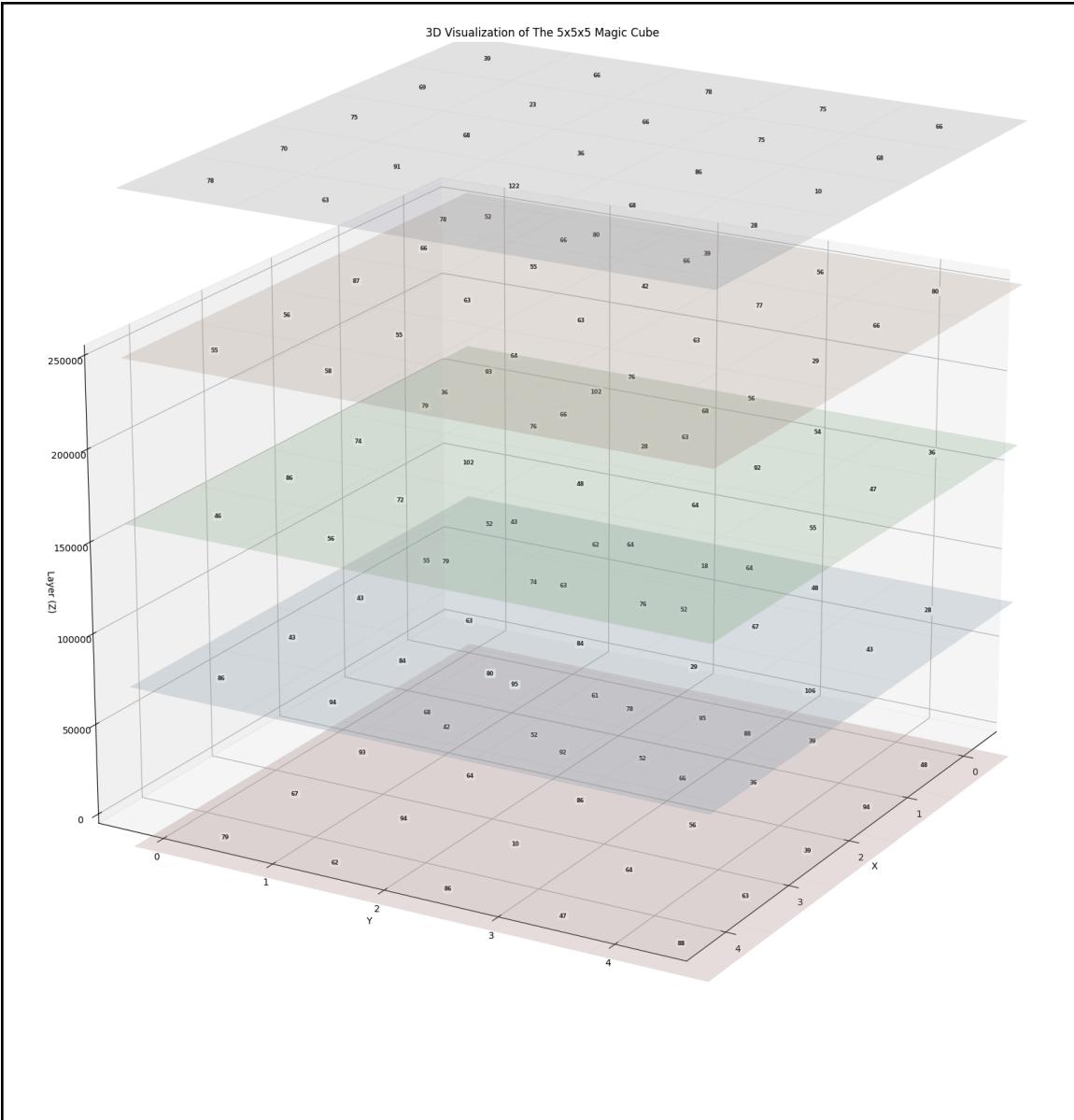
- Percobaan 1:



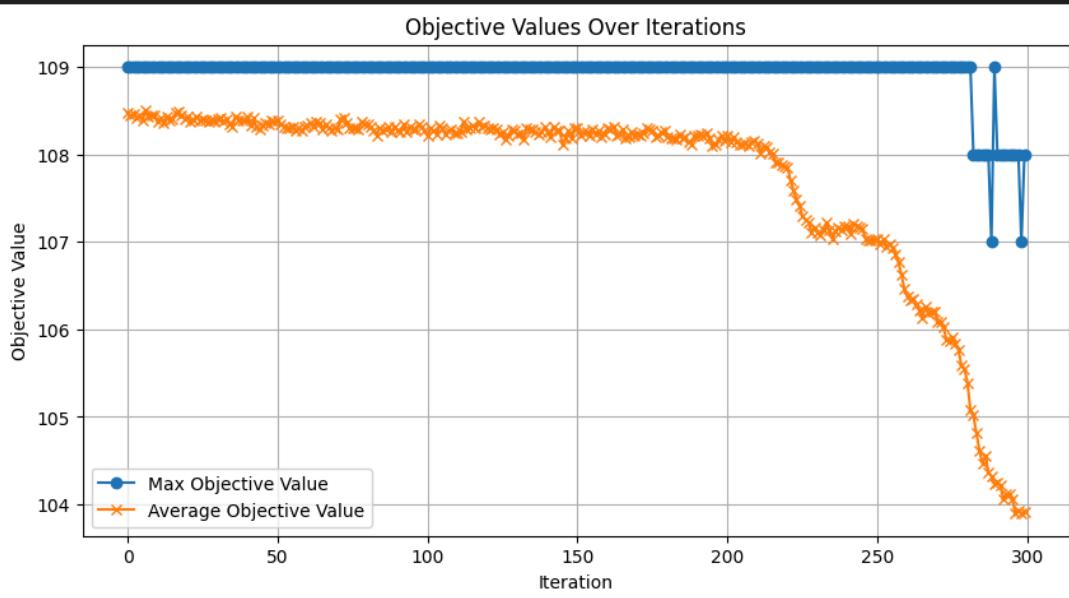
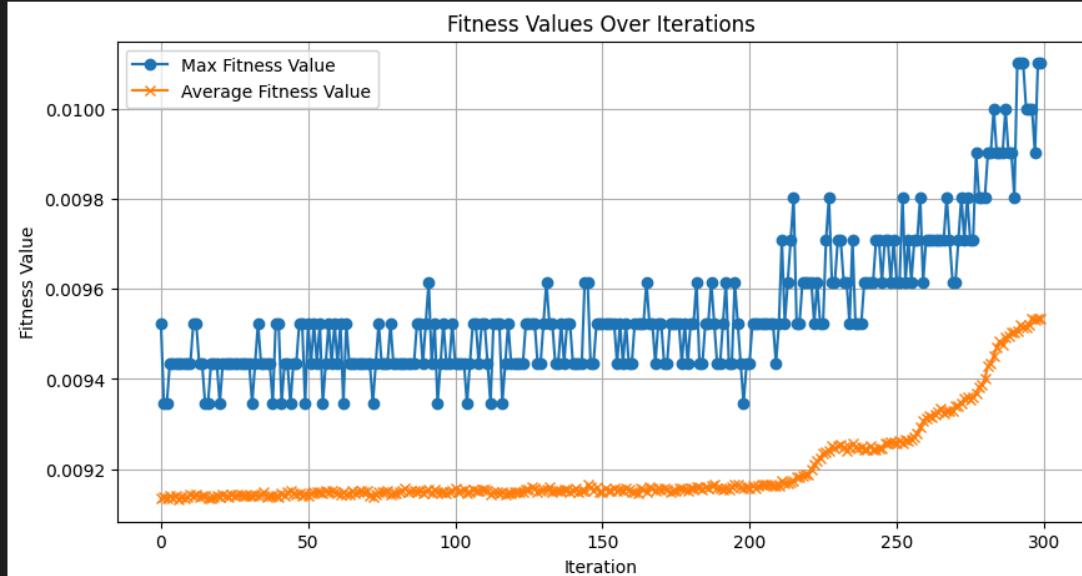
Final Objective Value: (4391, 109)
Population Size: 500
Iterations: 300
Duration: 169.58246207237244 seconds



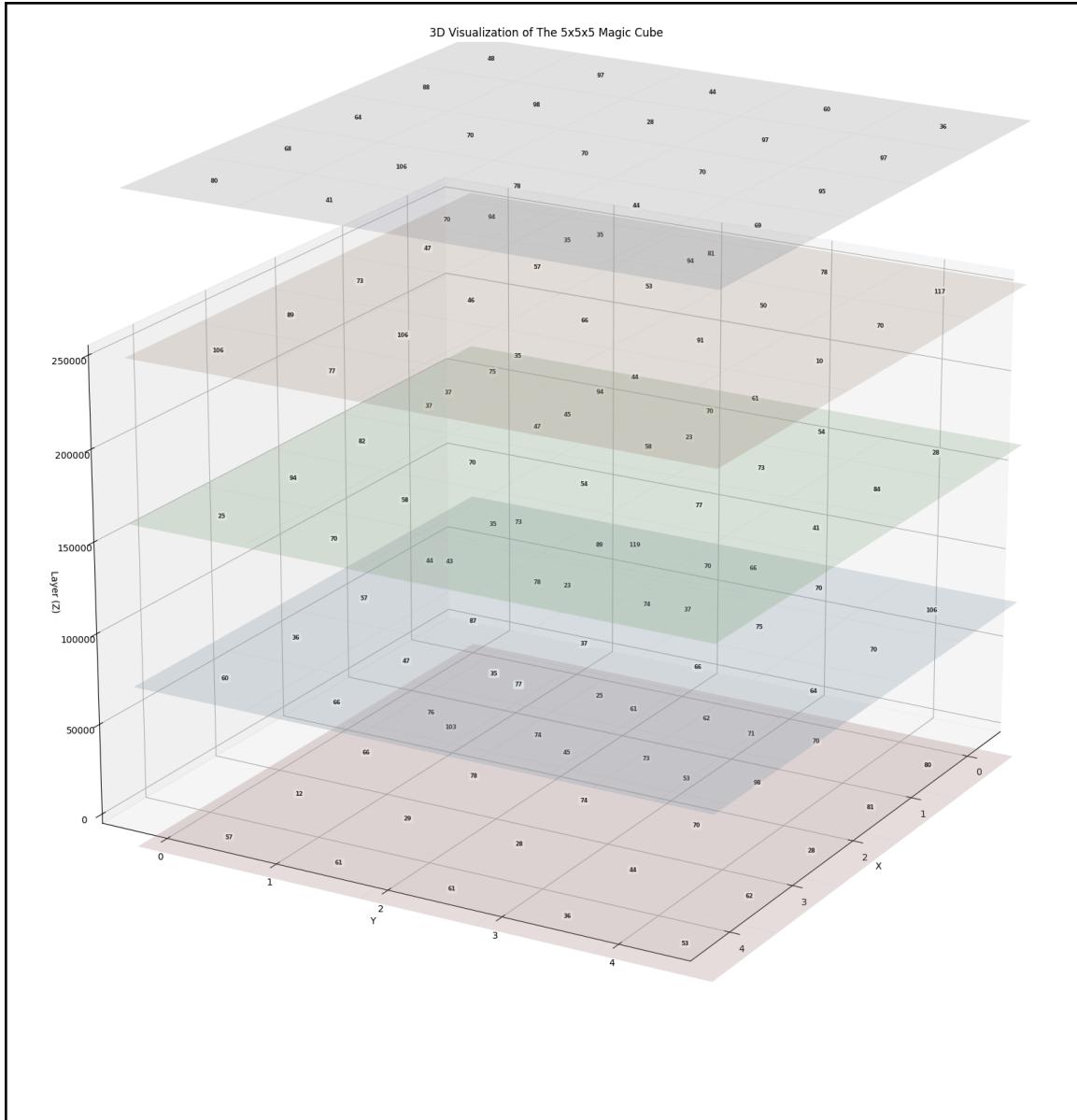
- Percobaan 2:



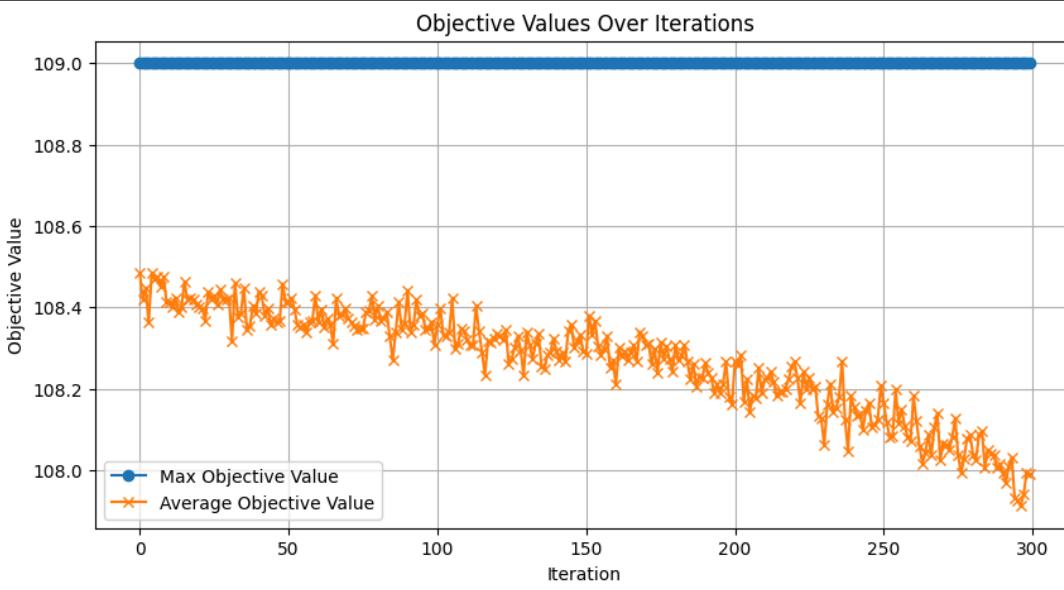
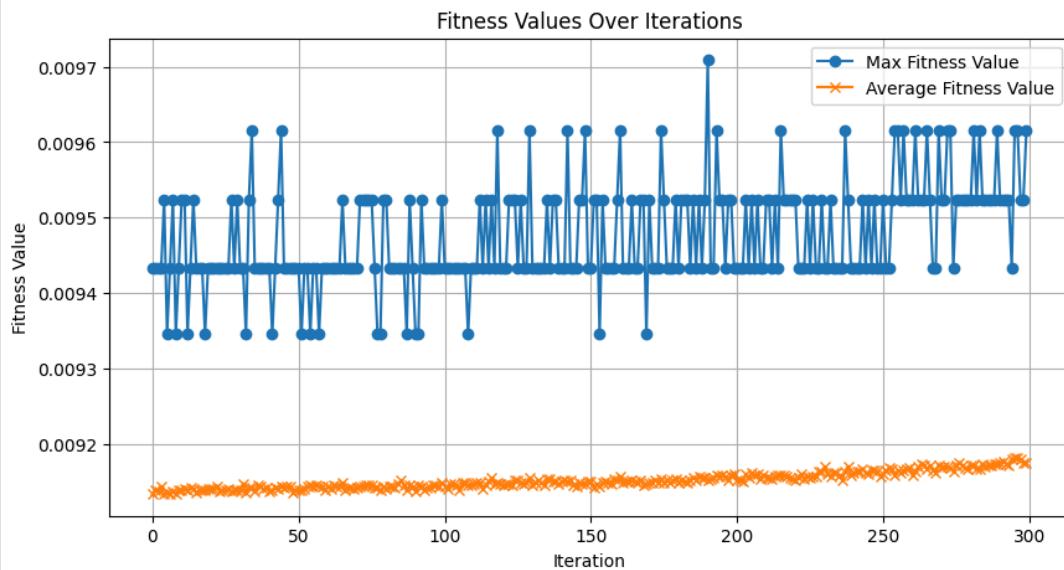
Final Objective Value: (3508, 106)
Population Size: 500
Iterations: 300
Duration: 157.0055639743805 seconds



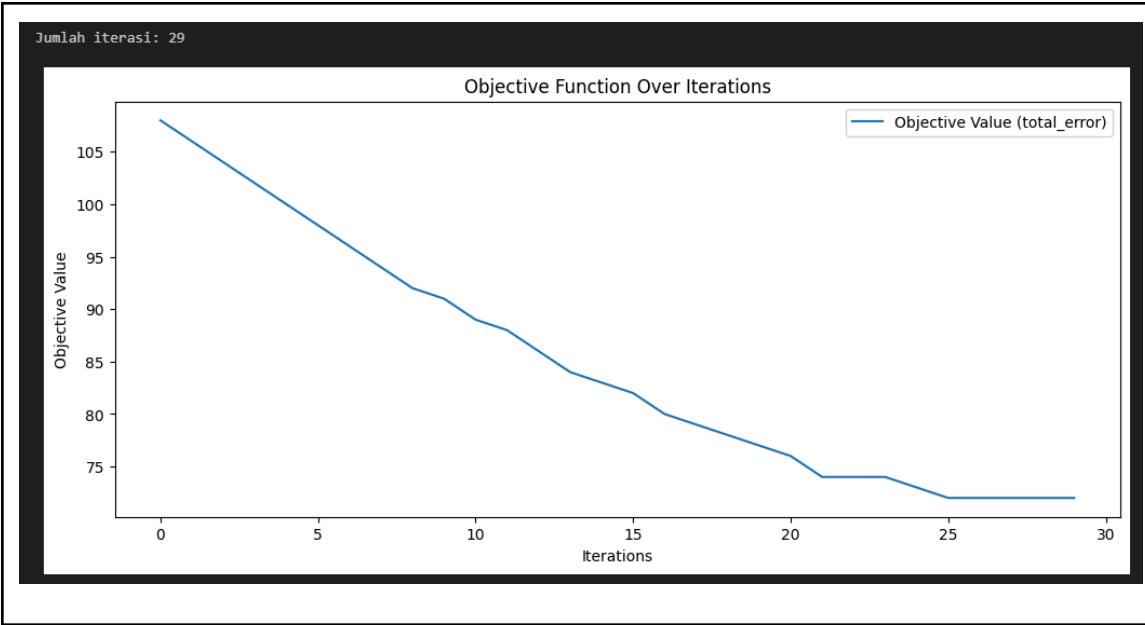
- Percobaan 3:

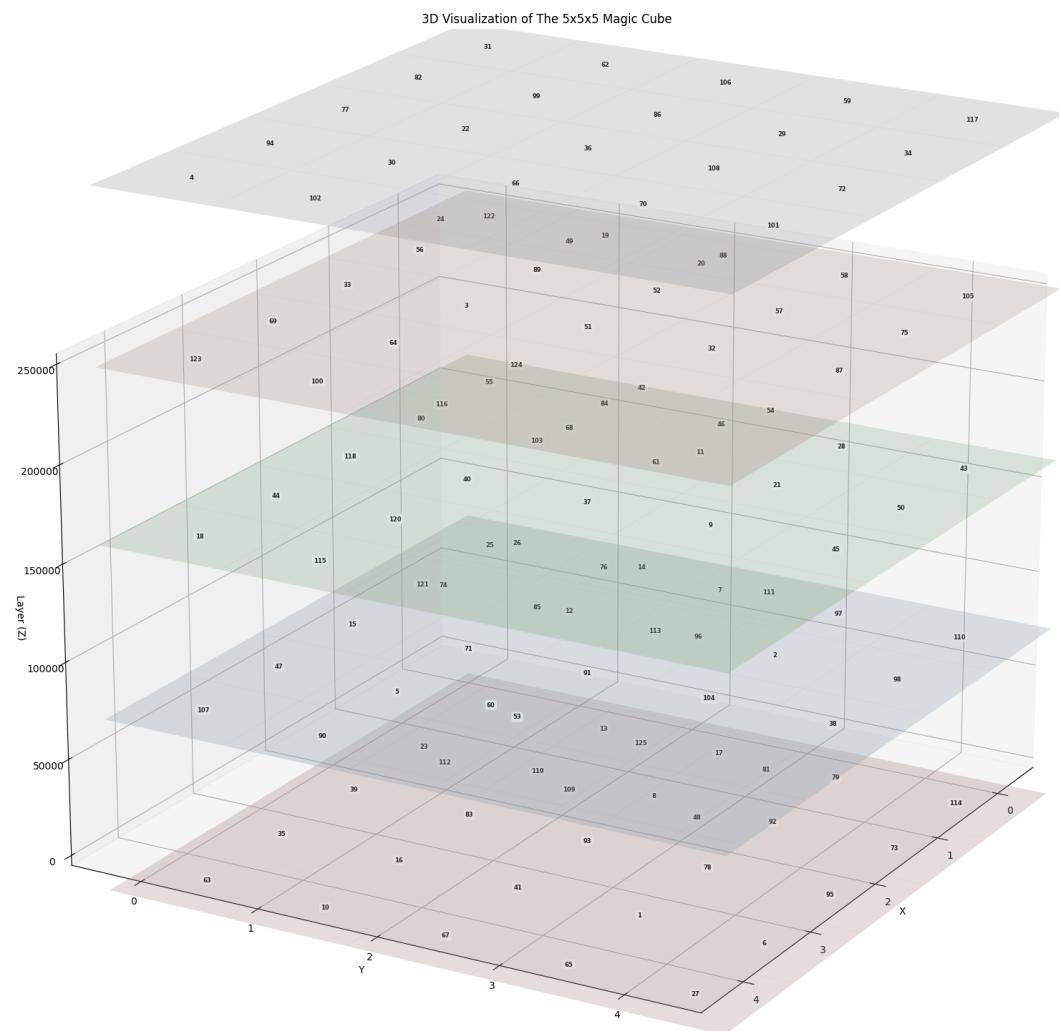


Final Objective Value: (4341, 108)
Population Size: 500
Iterations: 300
Duration: 166.38285732269287 seconds



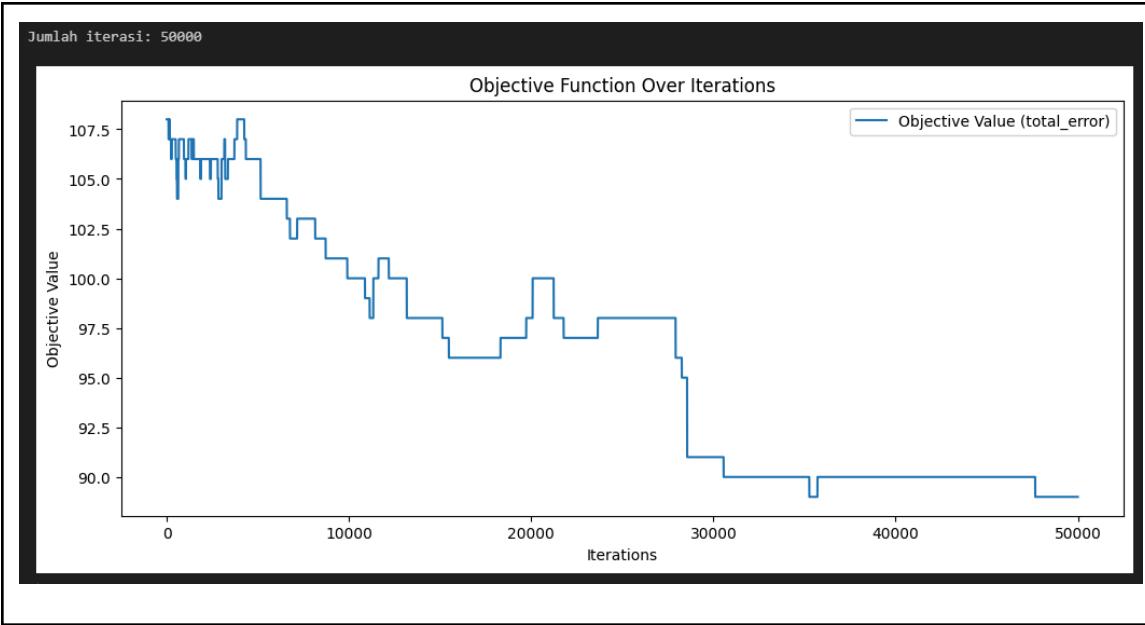
Hasil Sideways Move Hill Climbing

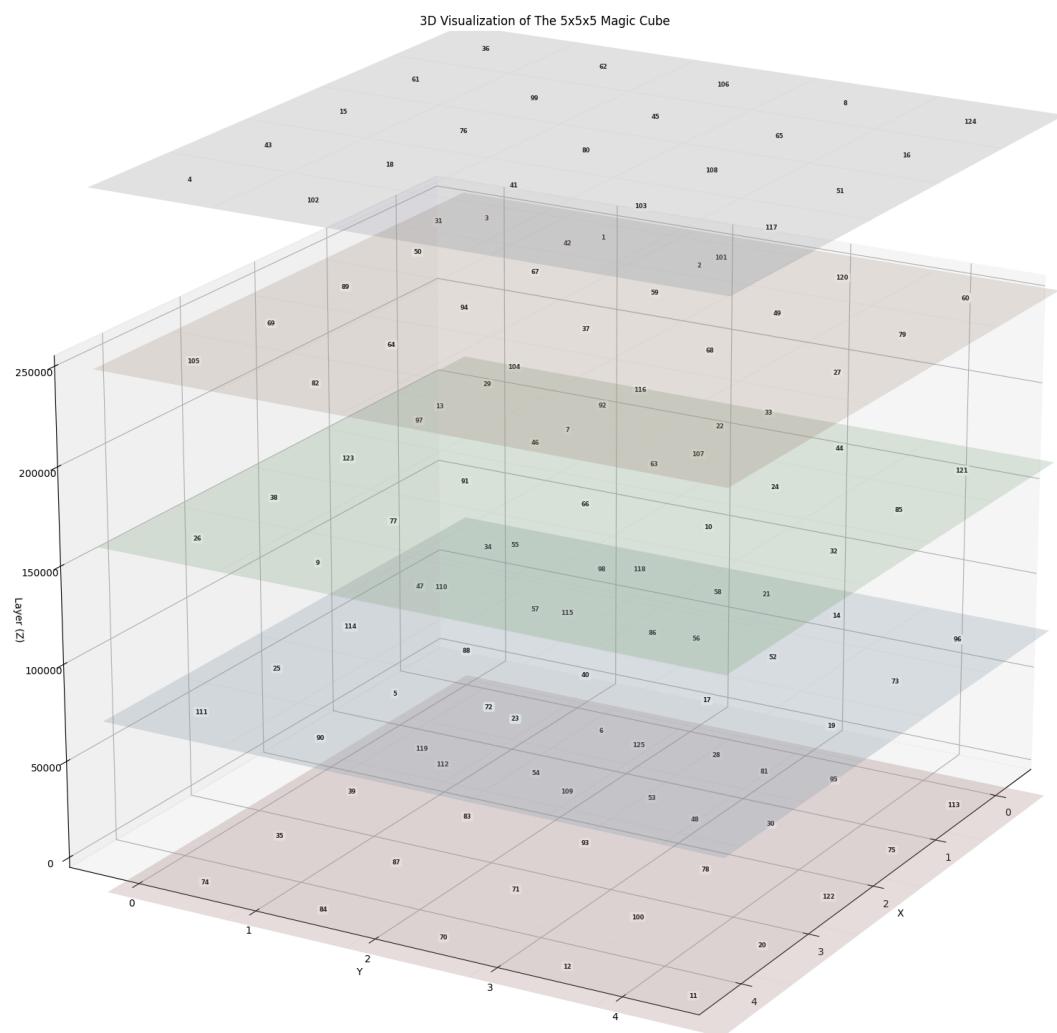




Best Cost: (4246, 72)
Duration: 186.97 seconds

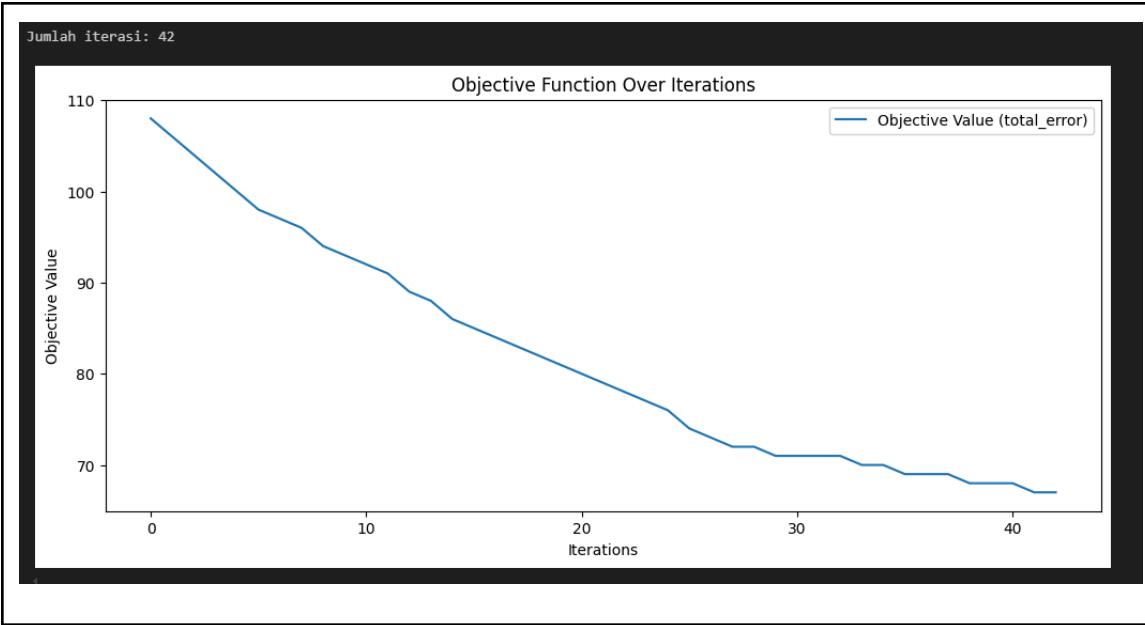
Hasil Stochastic Hill Climbing

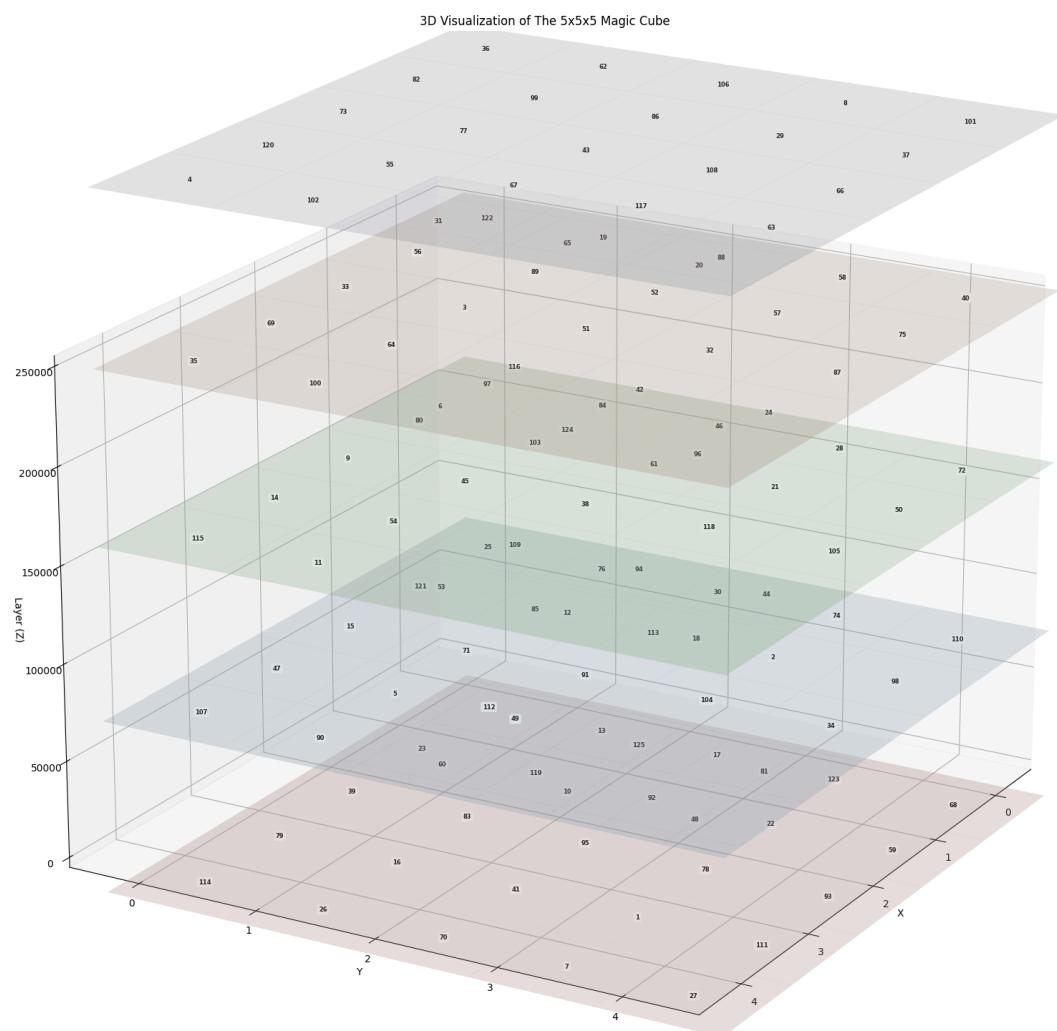




Best Cost: (2172, 89)
Duration: 37.67 seconds

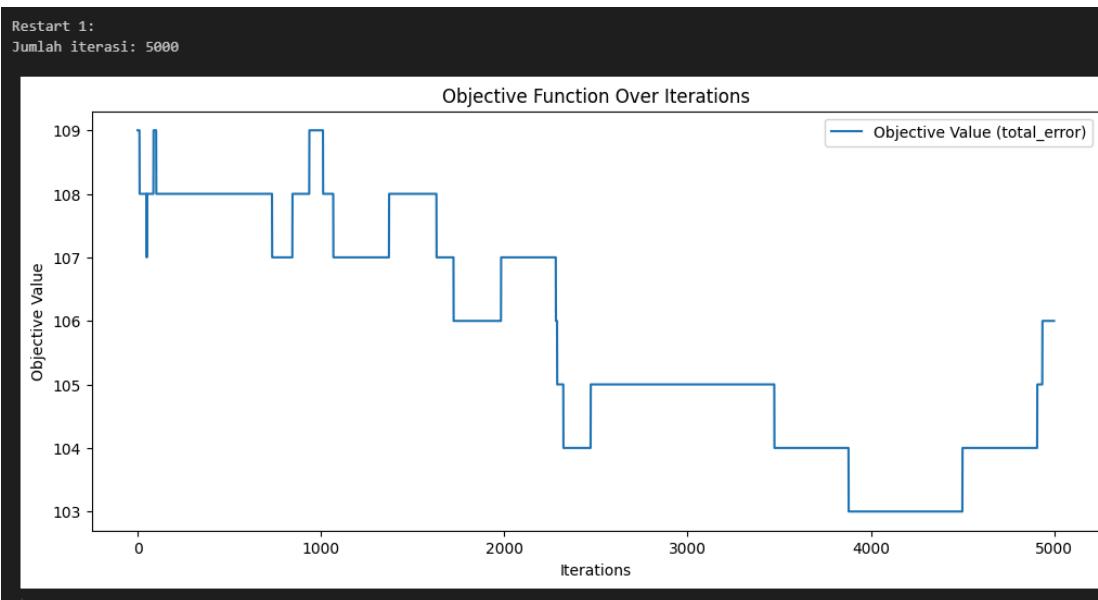
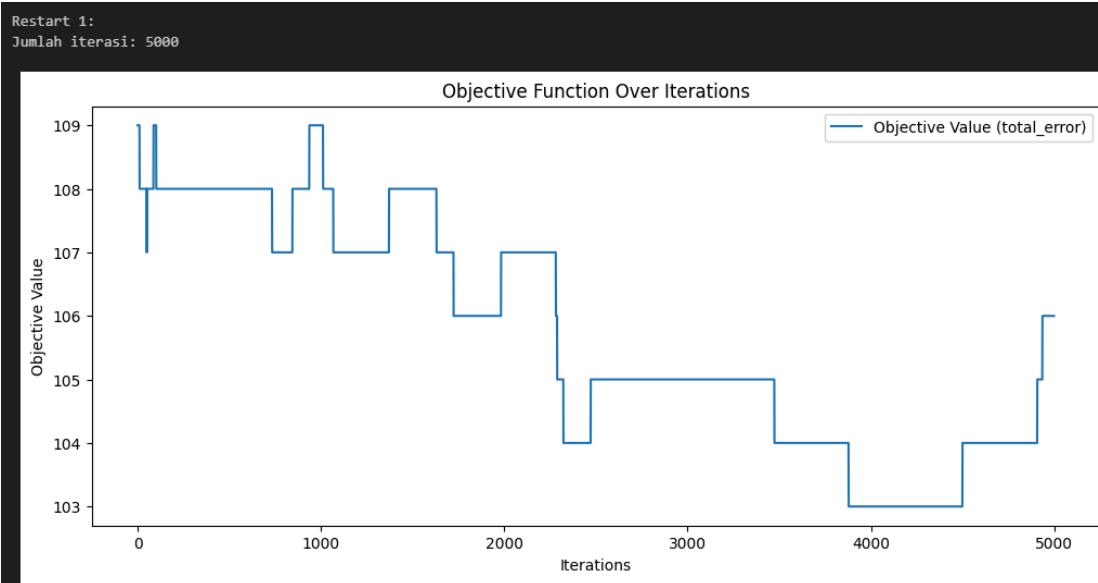
Hasil Steepest Ascent Hill Climbing





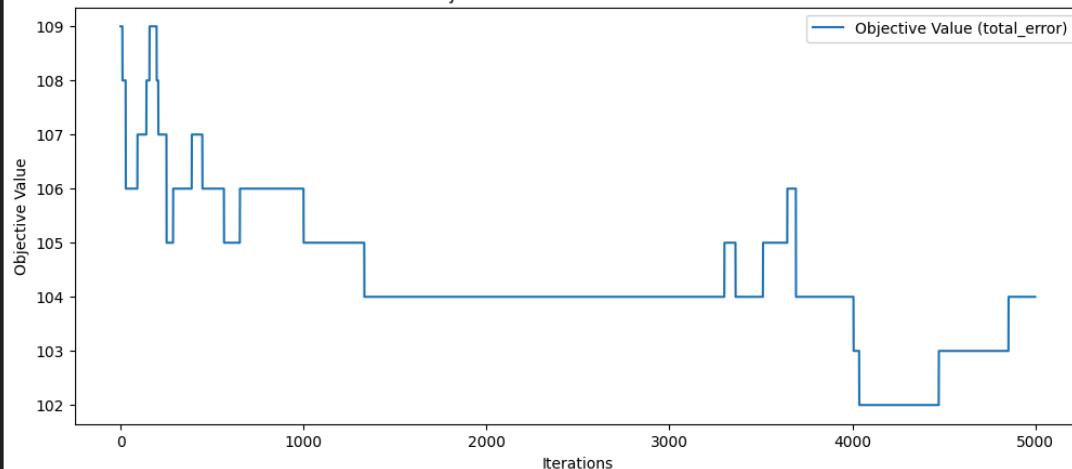
Best Cost: (3410, 67)
Duration: 254.54 seconds

Hasil Random Restart Hill Climbing



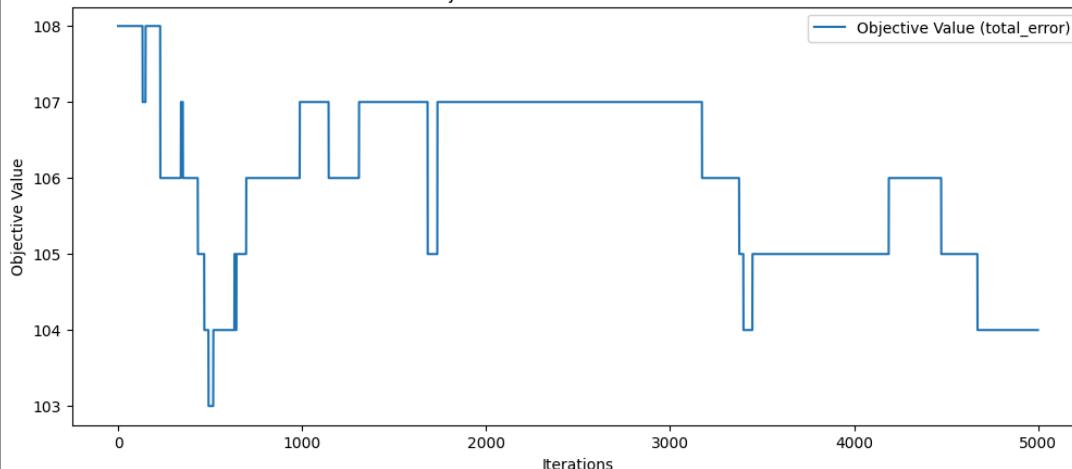
Restart 3:
Jumlah iterasi: 5000

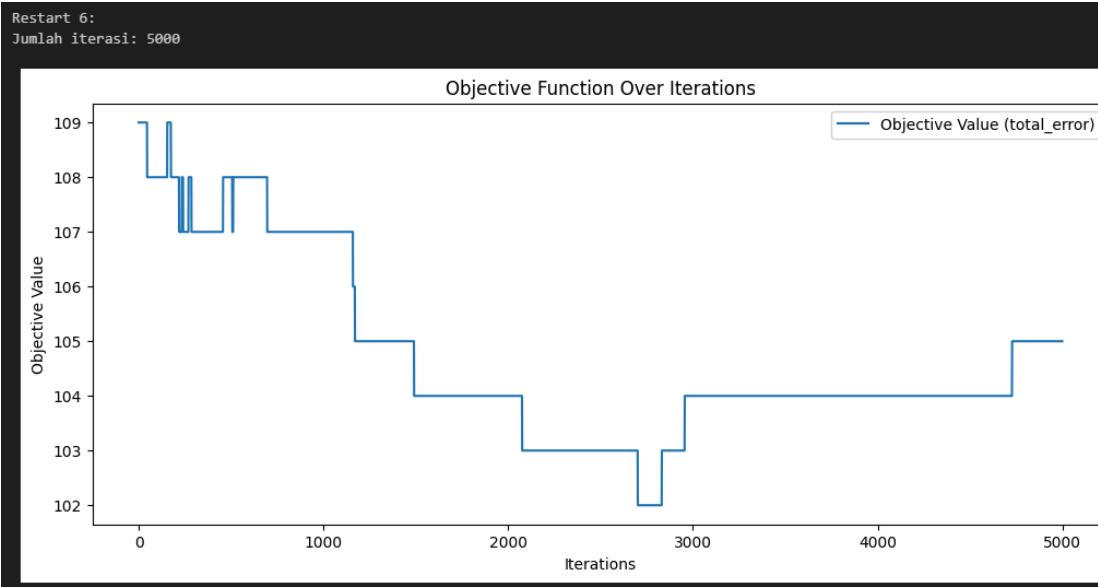
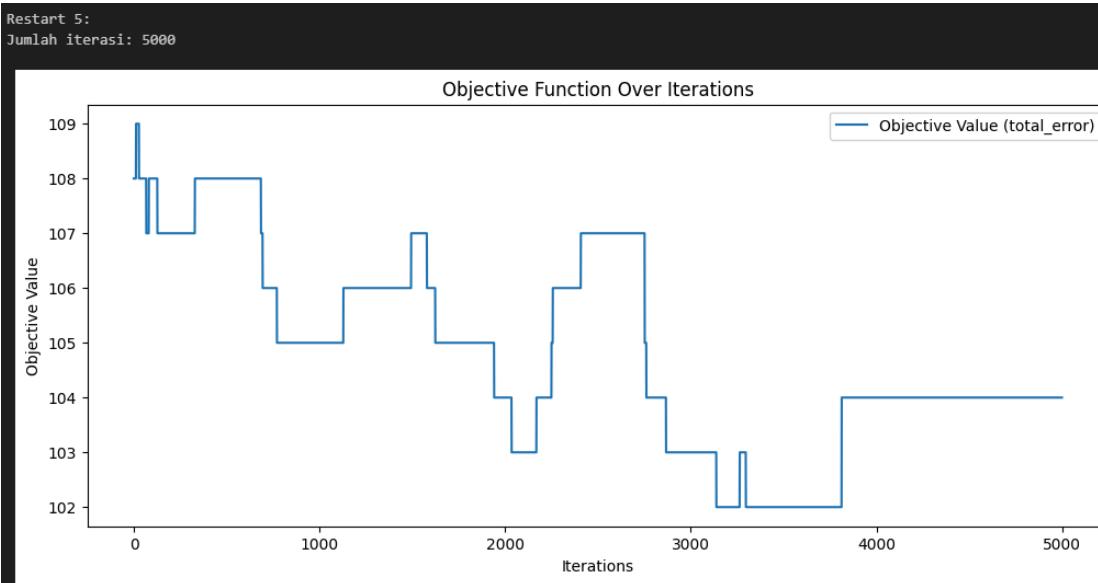
Objective Function Over Iterations



Restart 4:
Jumlah iterasi: 5000

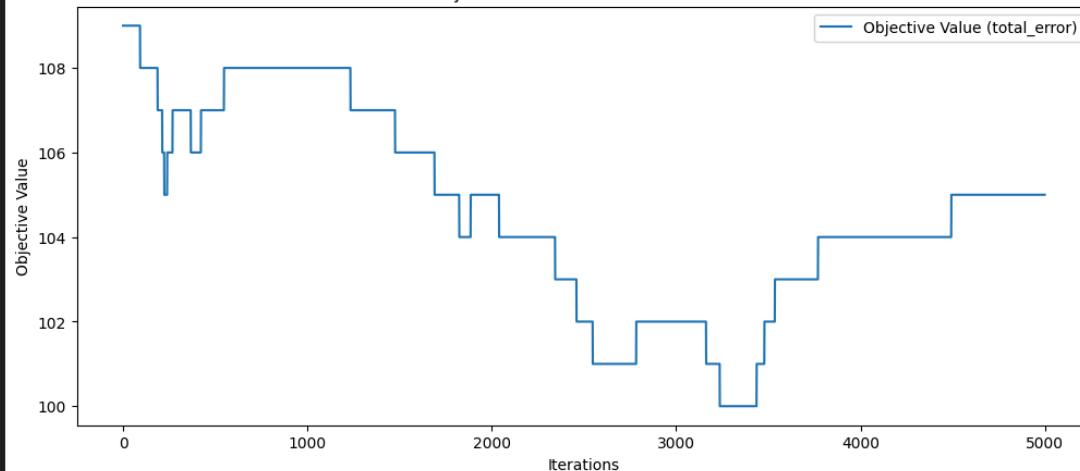
Objective Function Over Iterations





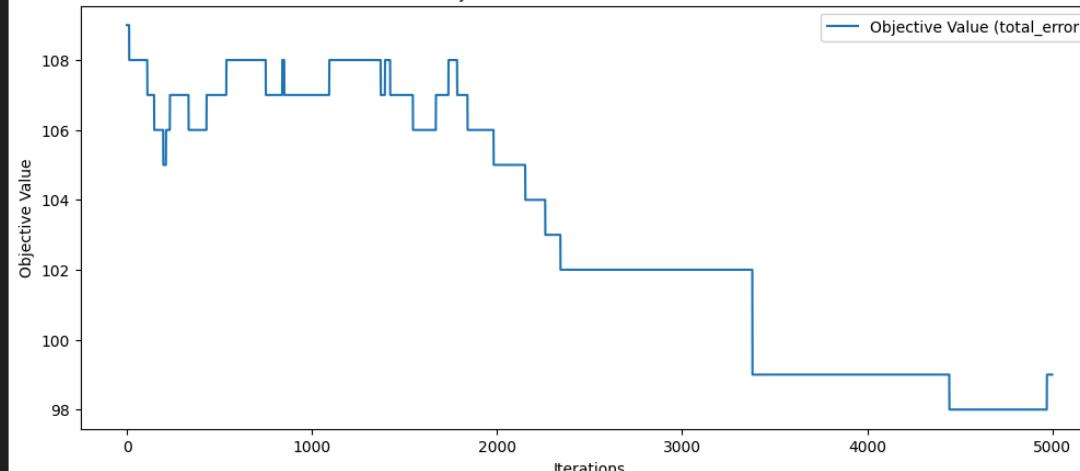
Restart 7:
Jumlah iterasi: 5000

Objective Function Over Iterations



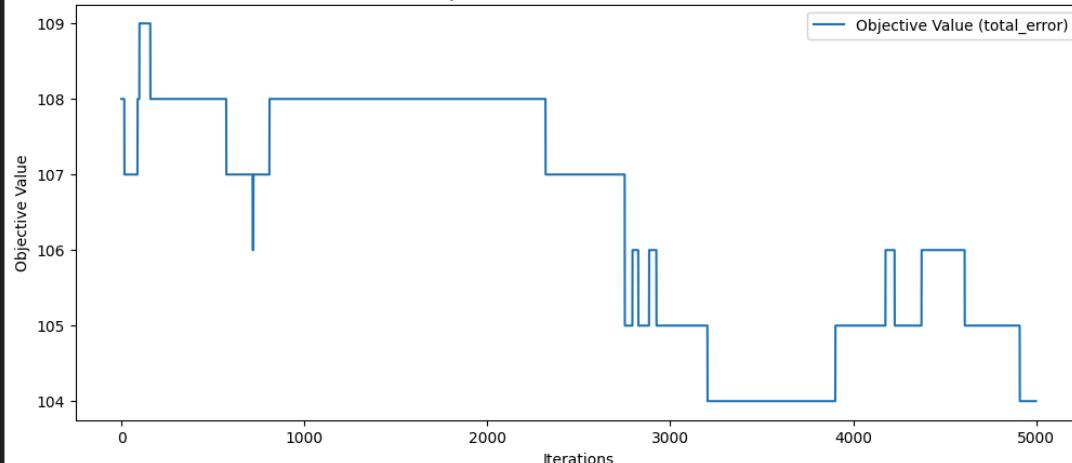
Restart 8:
Jumlah iterasi: 5000

Objective Function Over Iterations



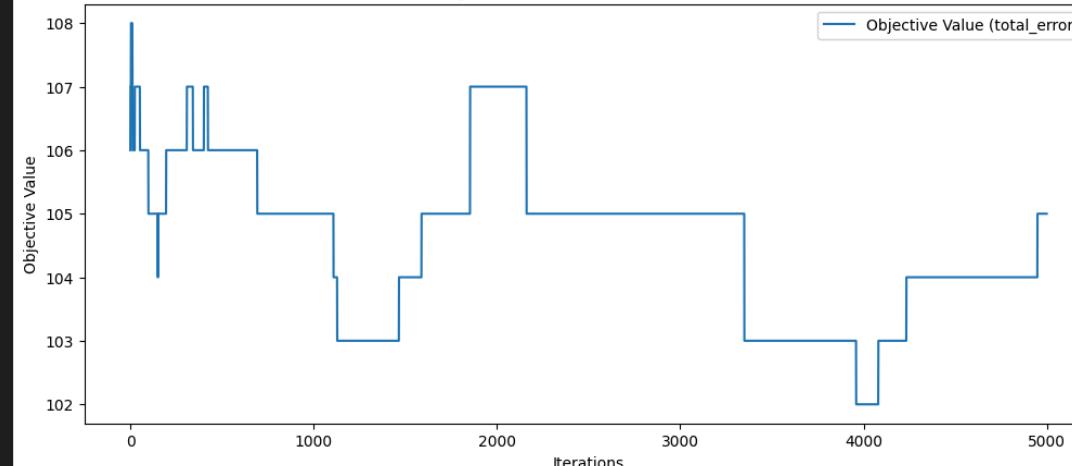
Restart 9:
Jumlah iterasi: 5000

Objective Function Over Iterations

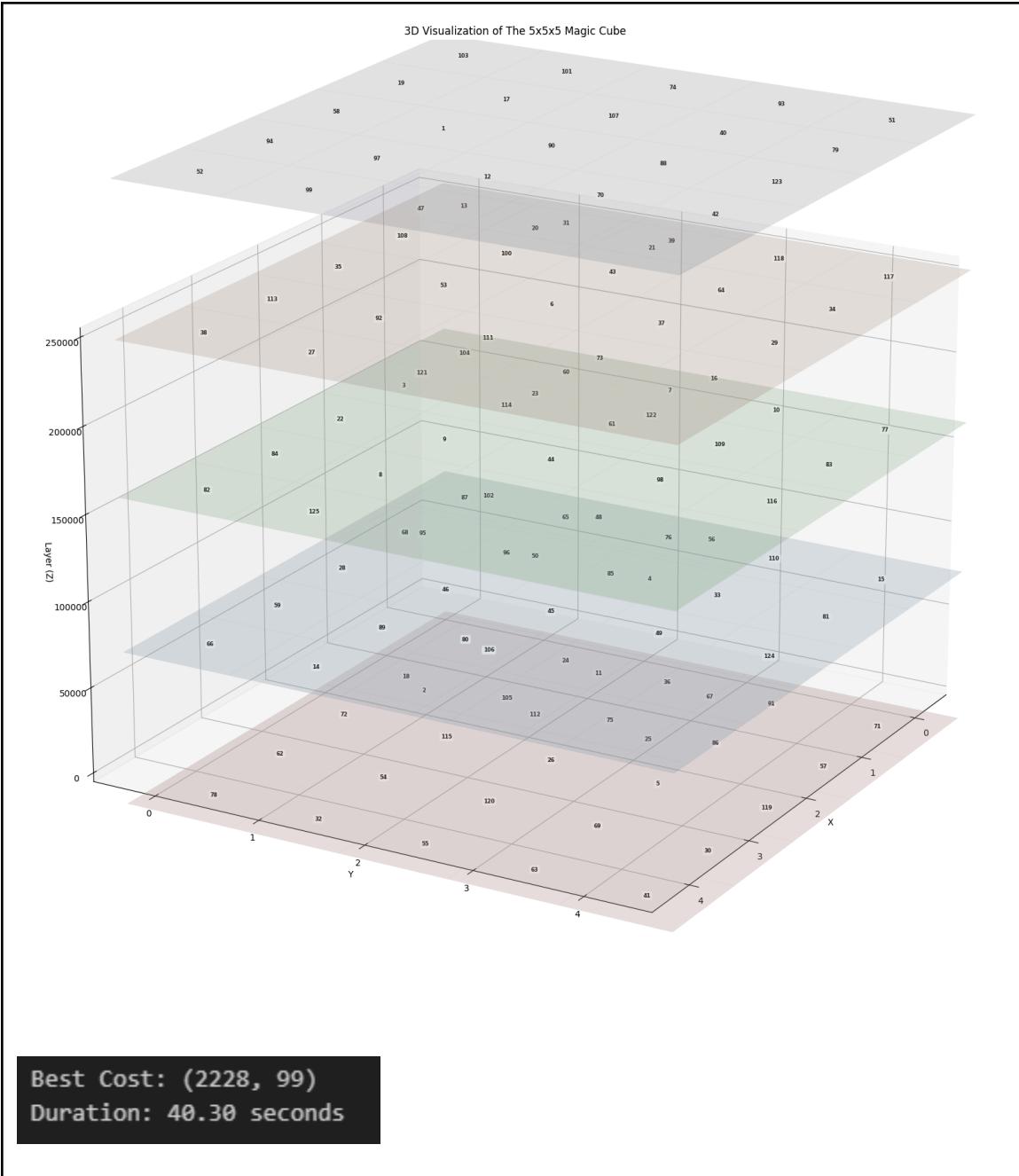


Restart 10:
Jumlah iterasi: 5000

Objective Function Over Iterations

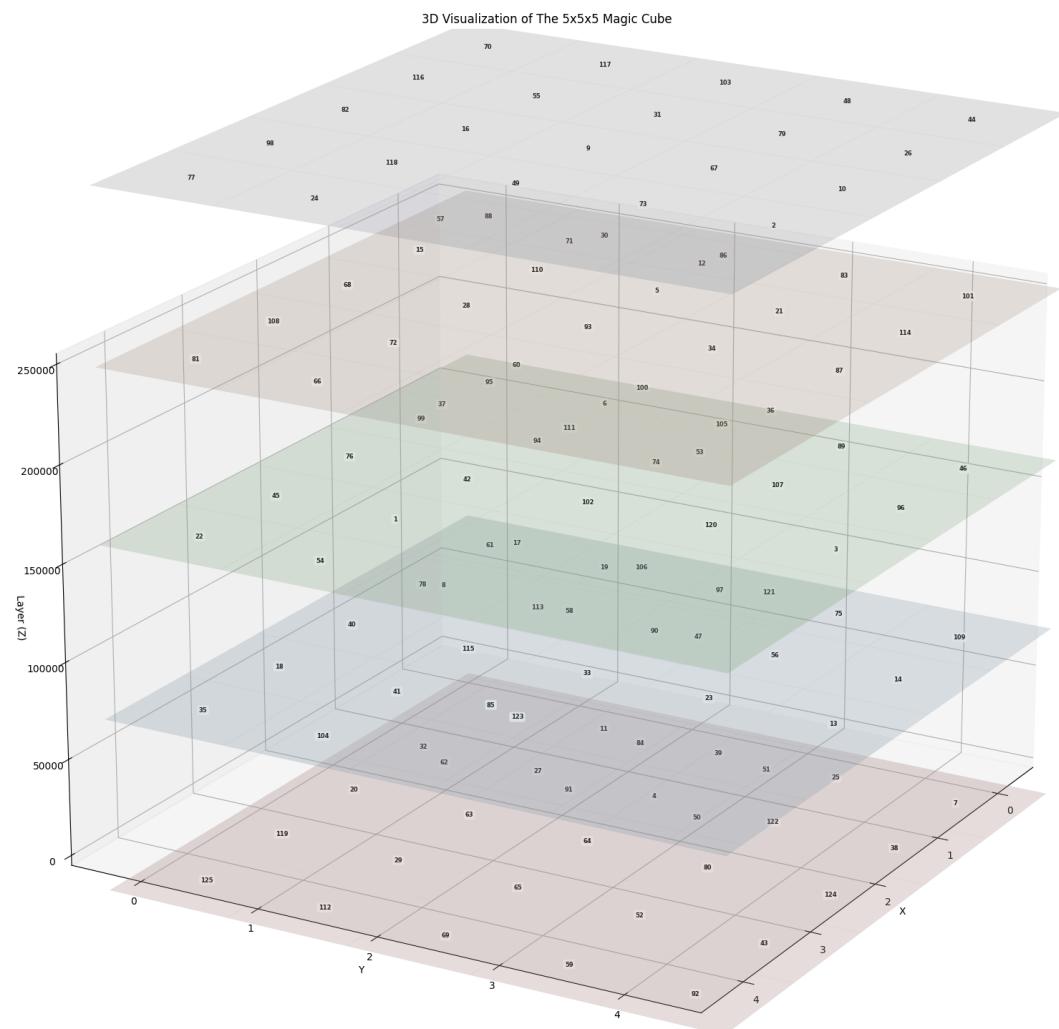


Jumlah restart: 10



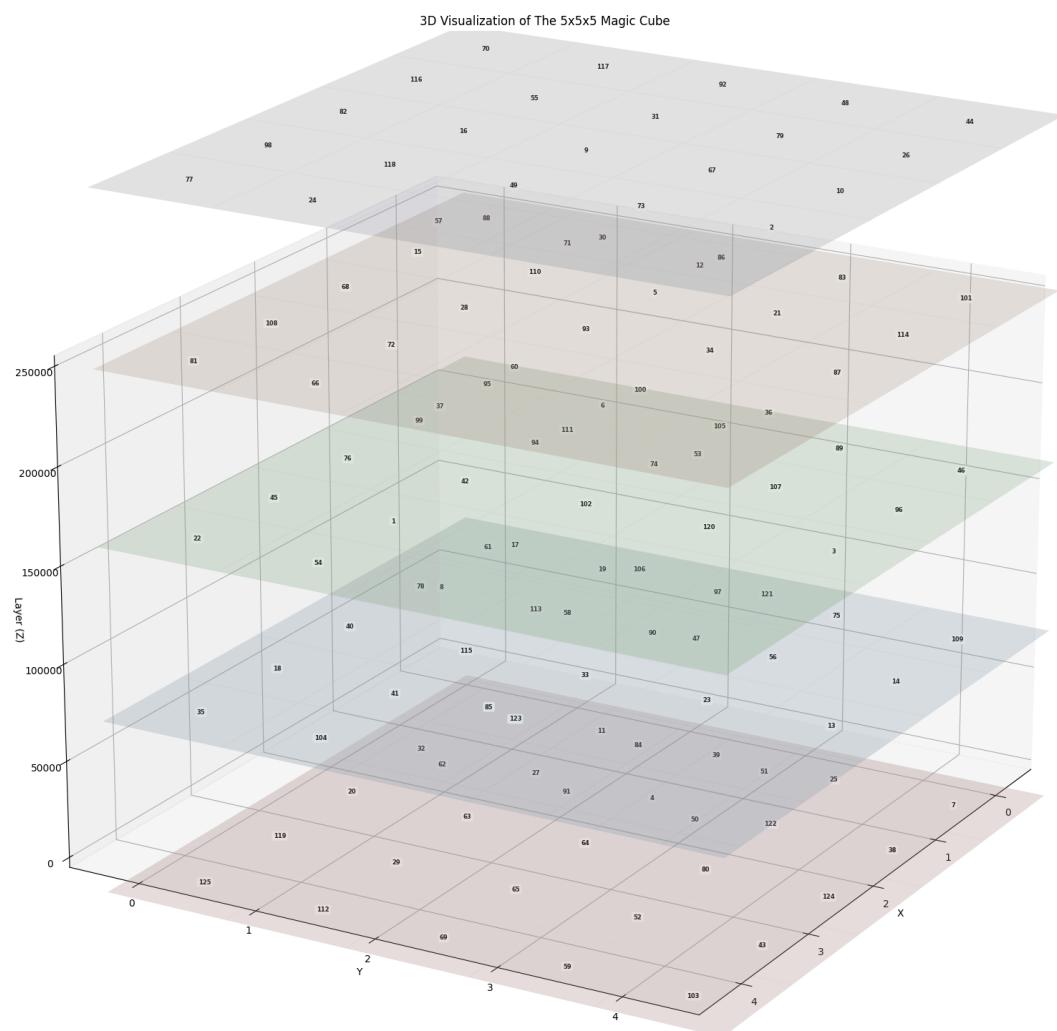
Test Case 3

Initial Cube

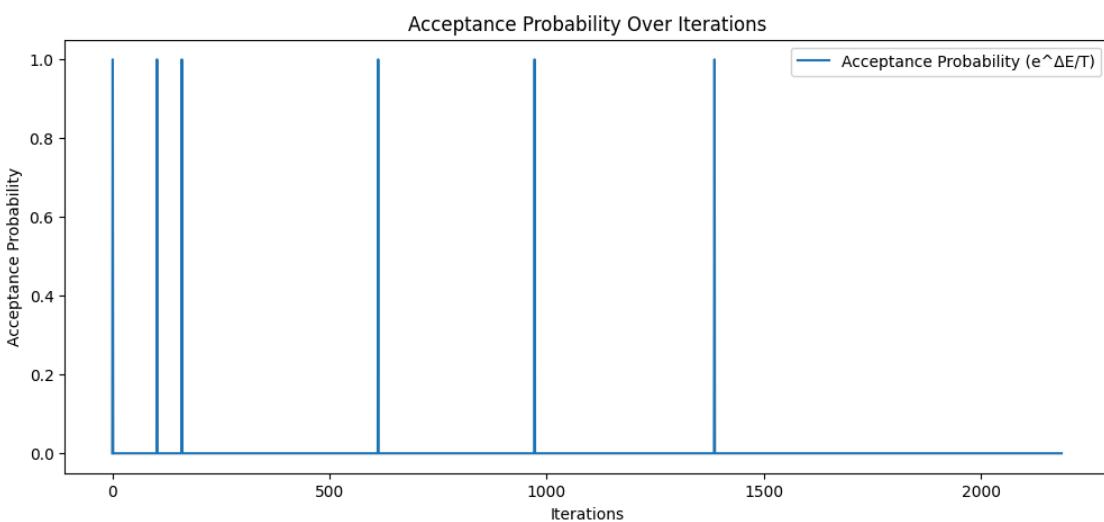
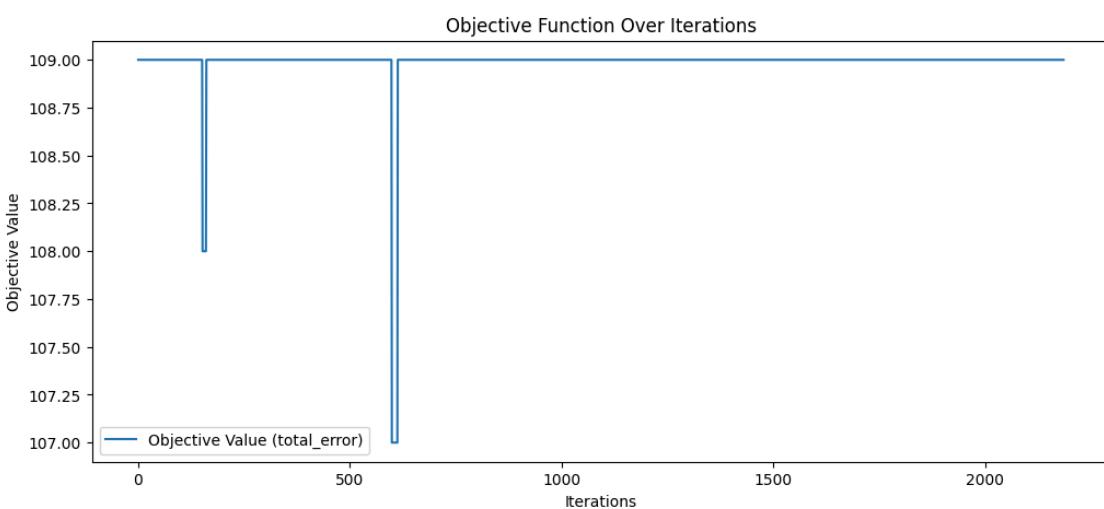


Initial Objective Function Value (total_error, error_count): (6393, 109)

Hasil Simulated Annealing



Final Objective Function Value (total_error, error_count): (6261, 107)
Duration: 1.89 seconds
Frequency of being 'stuck' at local optima: 2180

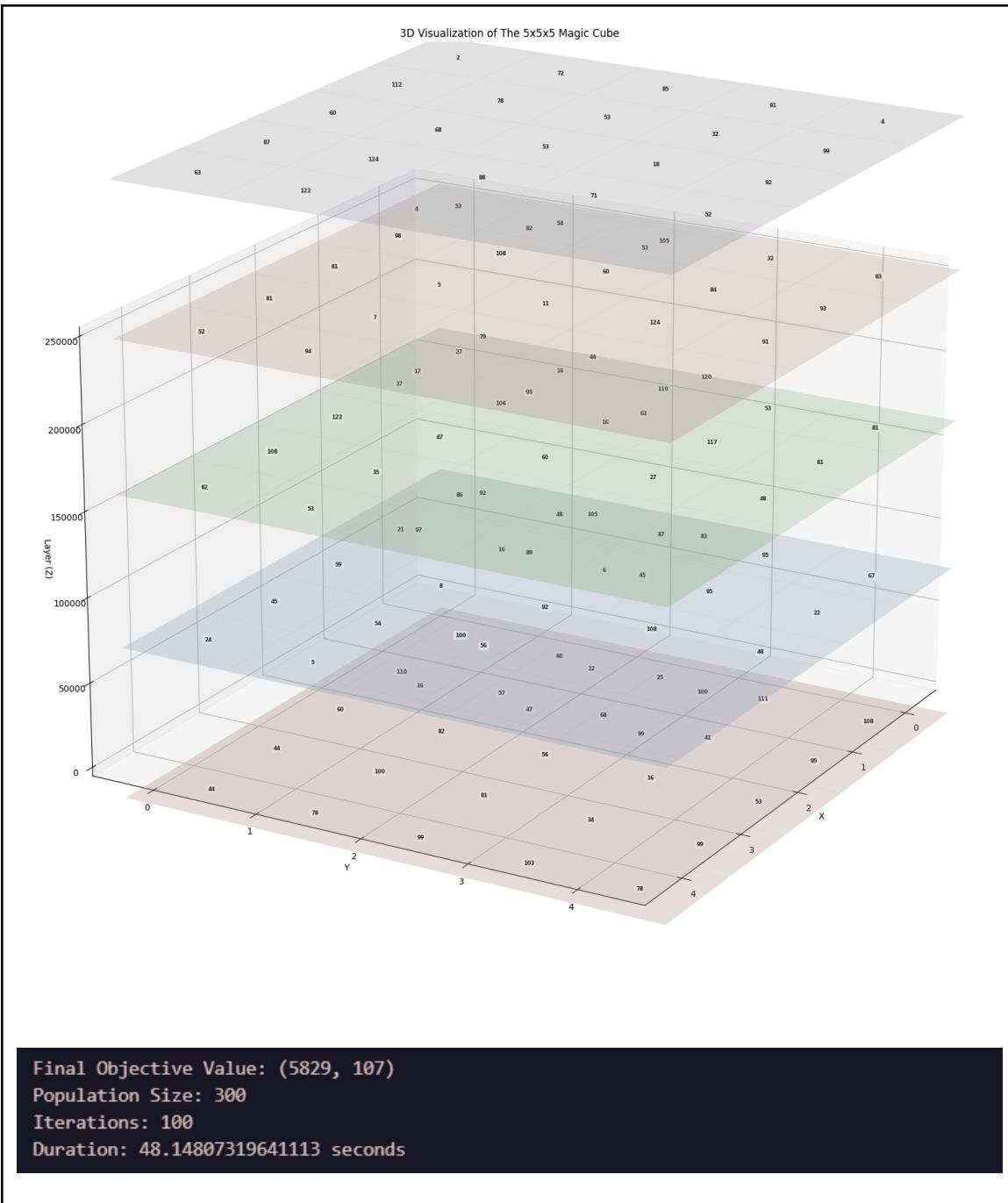


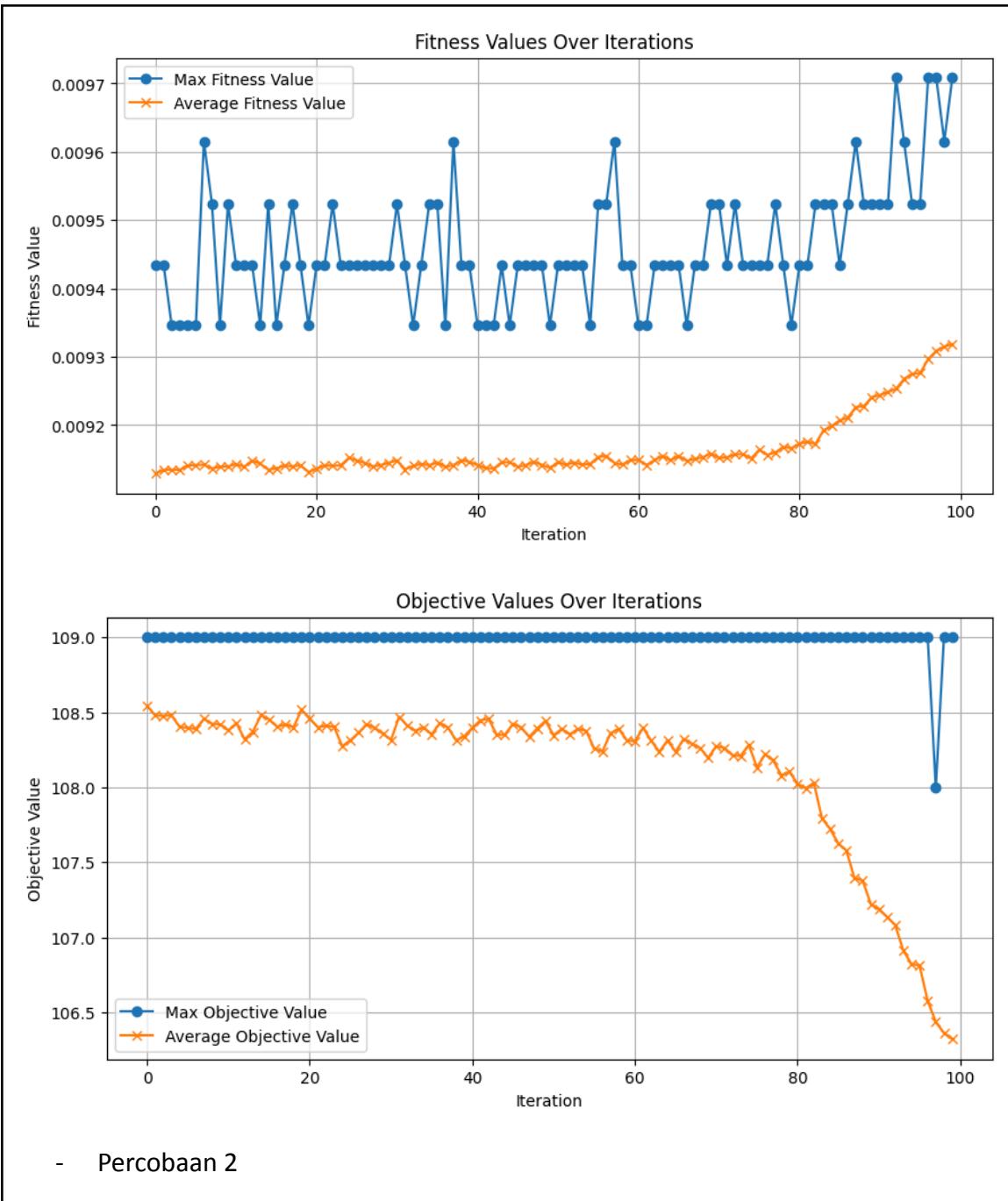
Hasil Genetic Algorithm

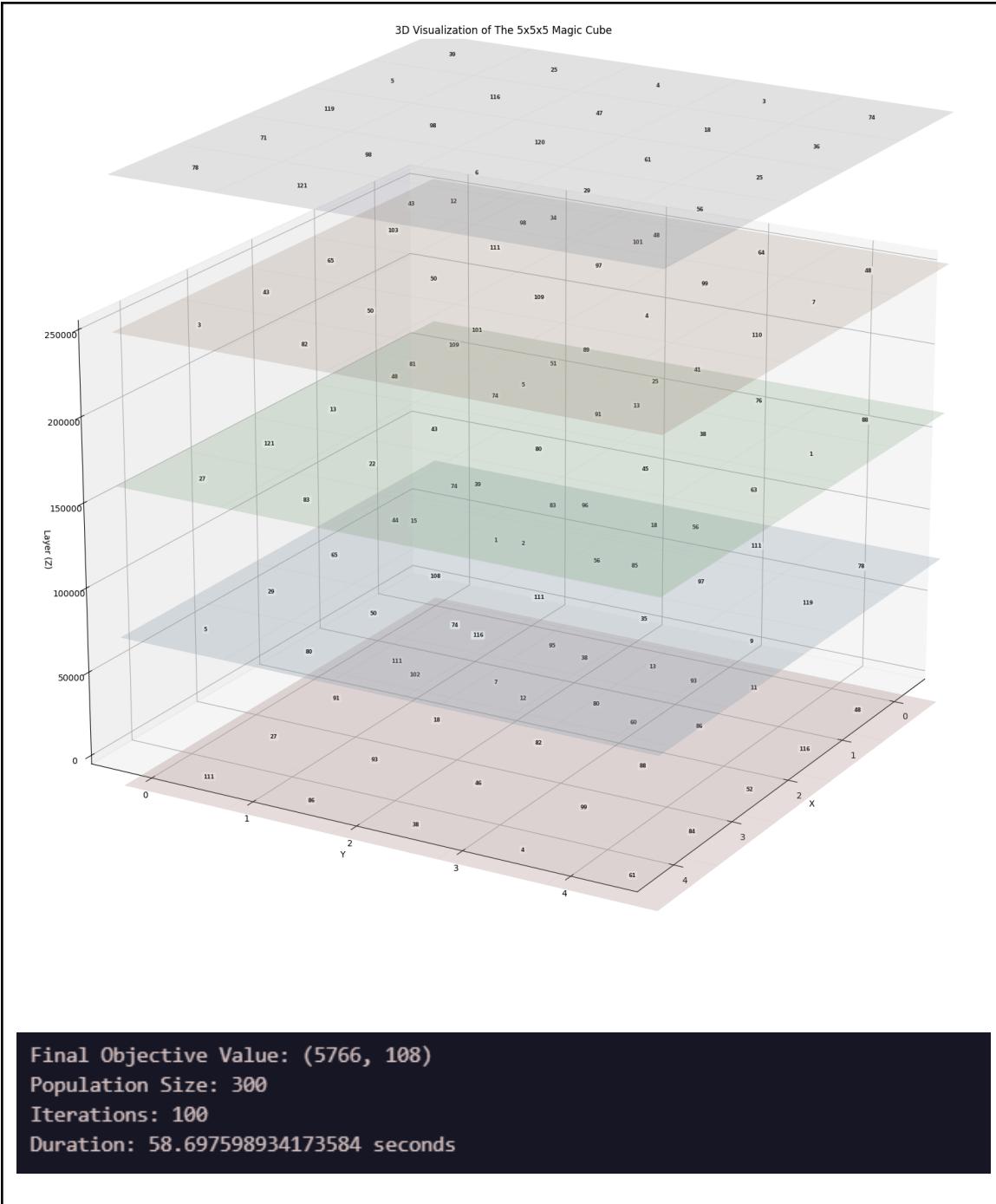
Populasi sebagai kontrol,

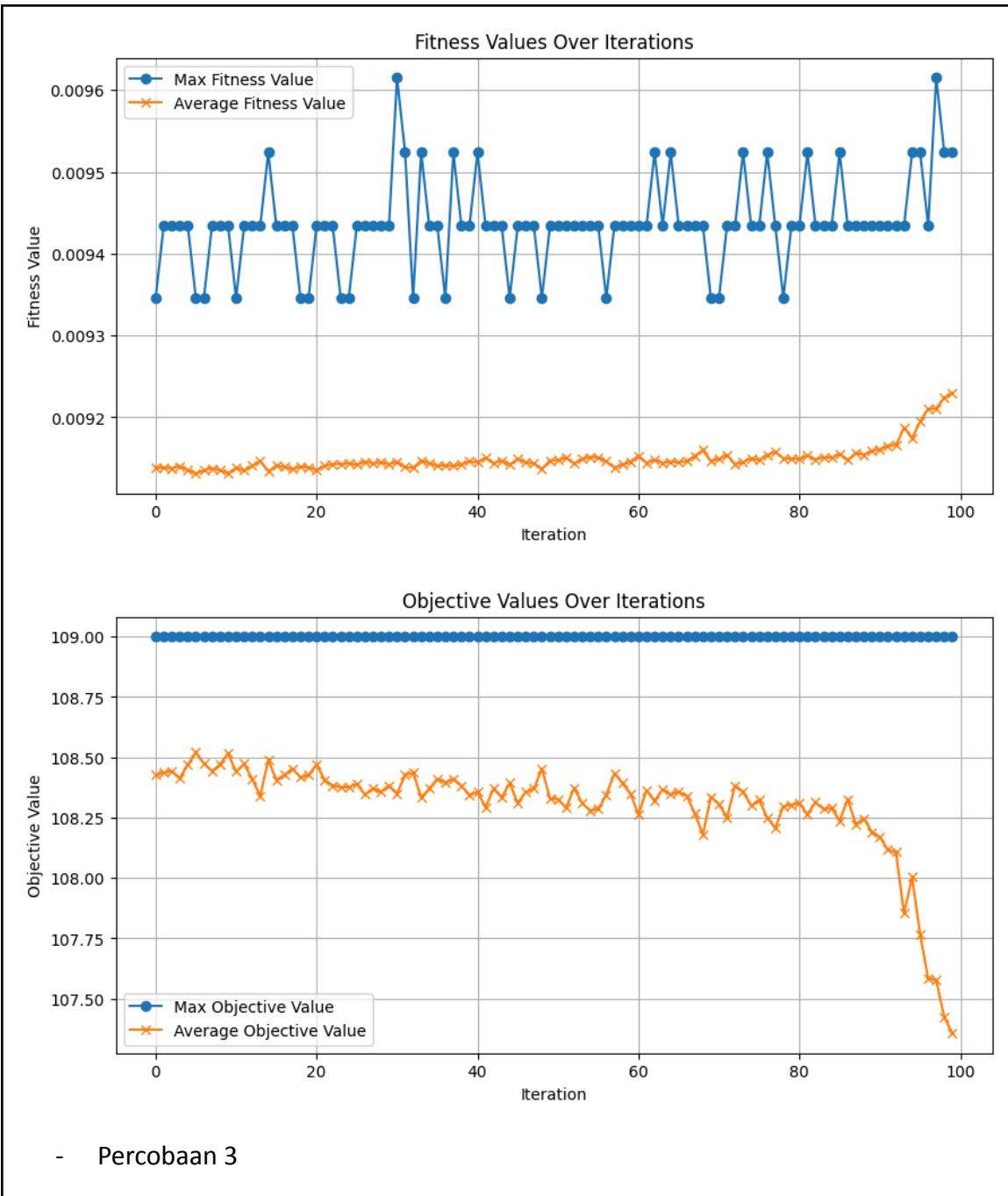
Variasi 1: 300, 100

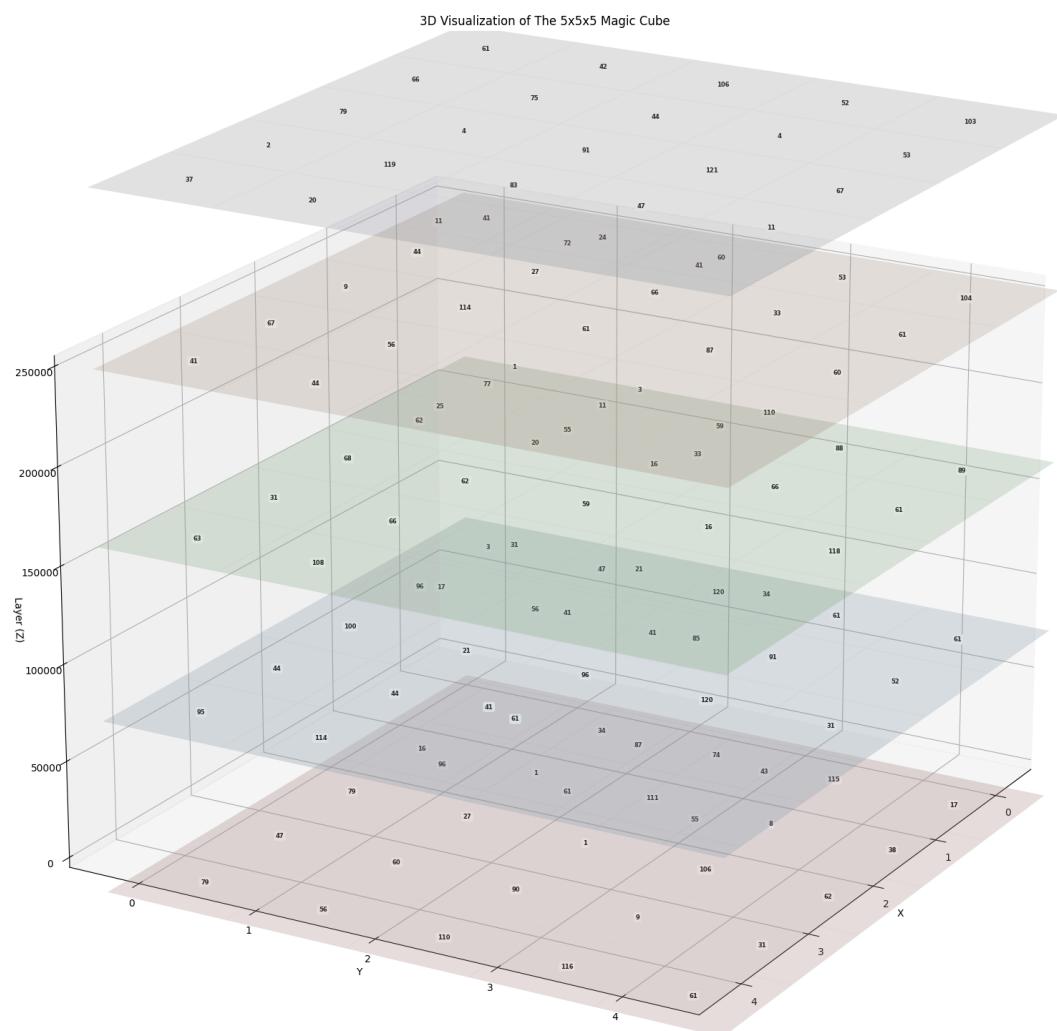
- Perocbaan 1



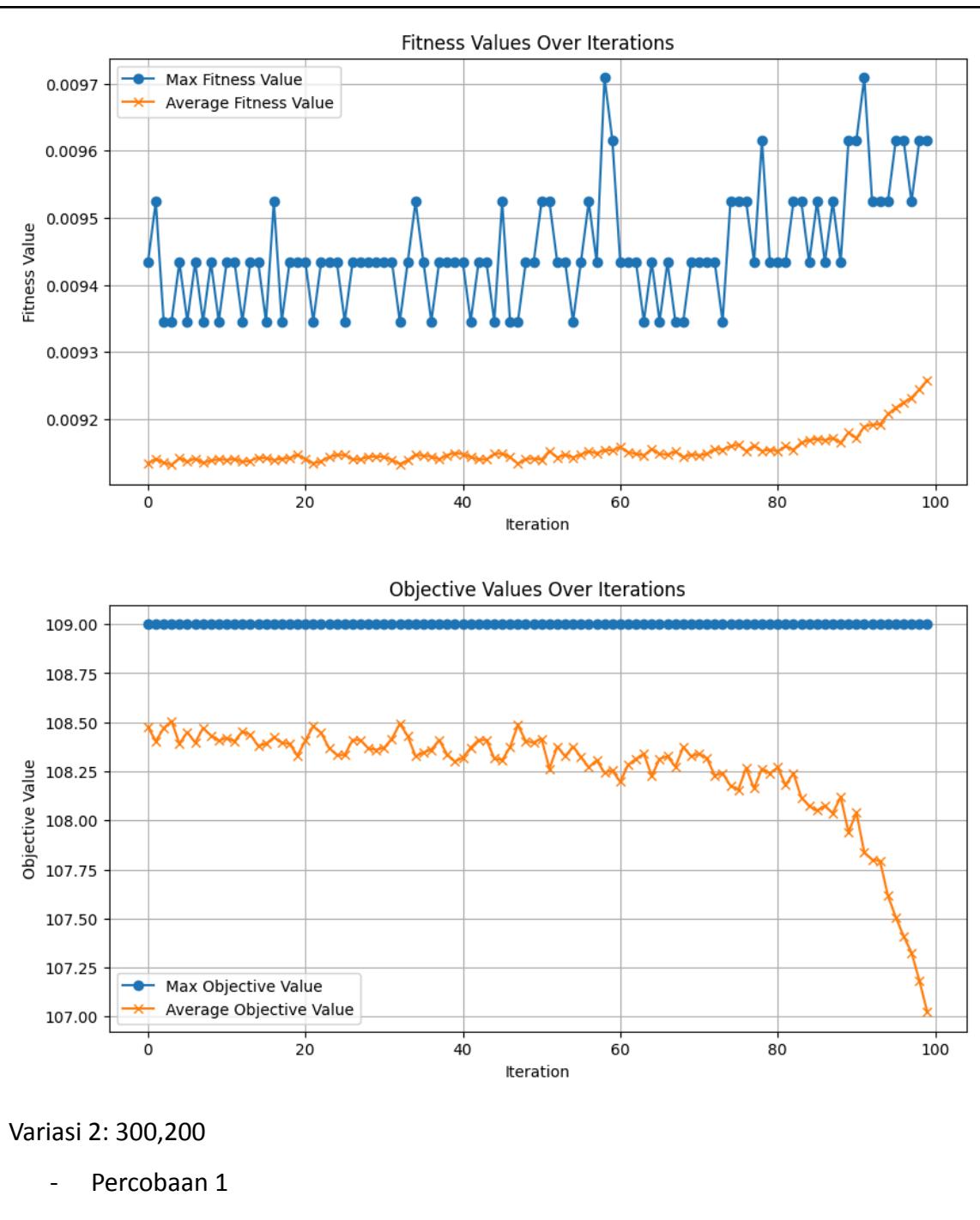


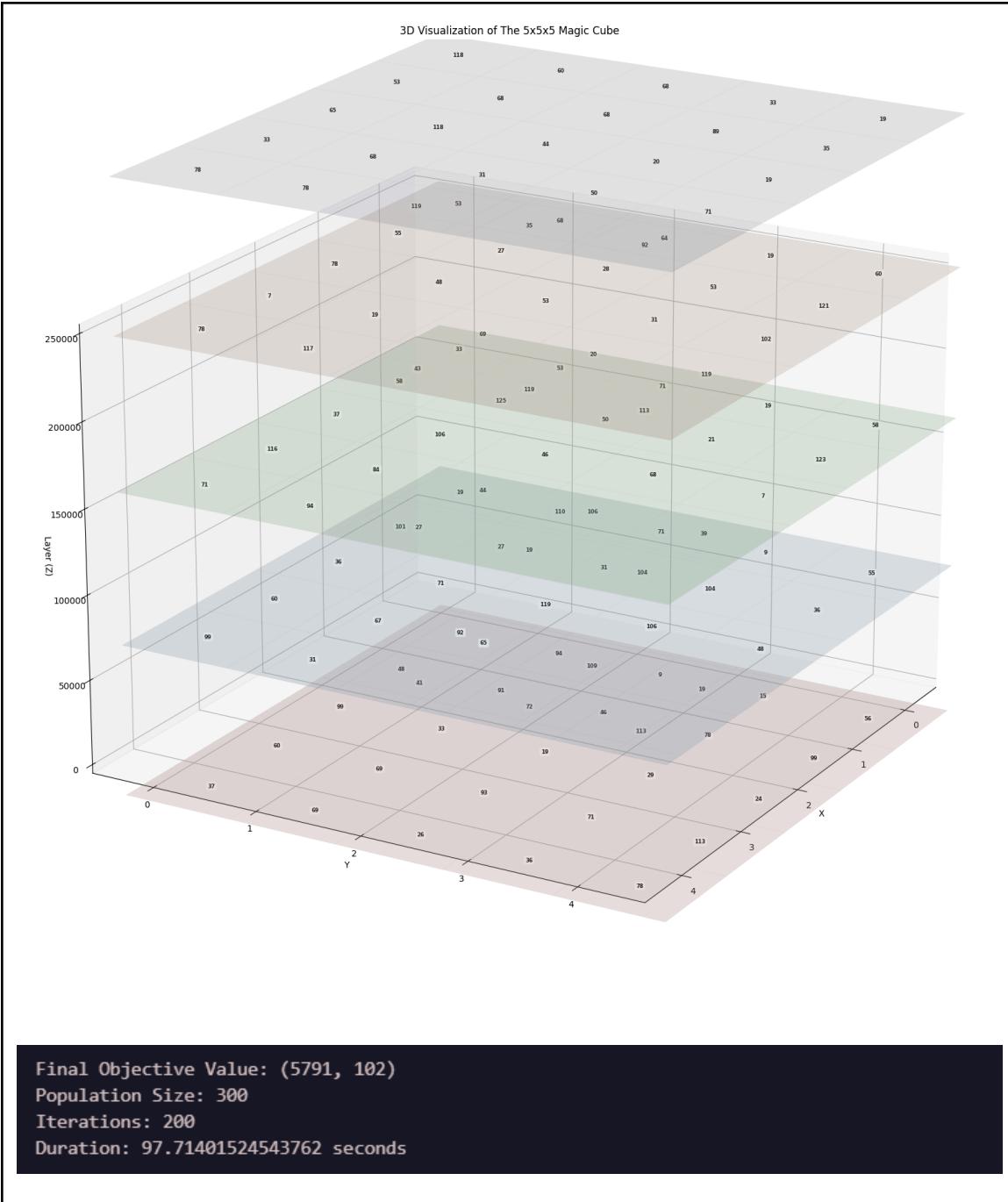


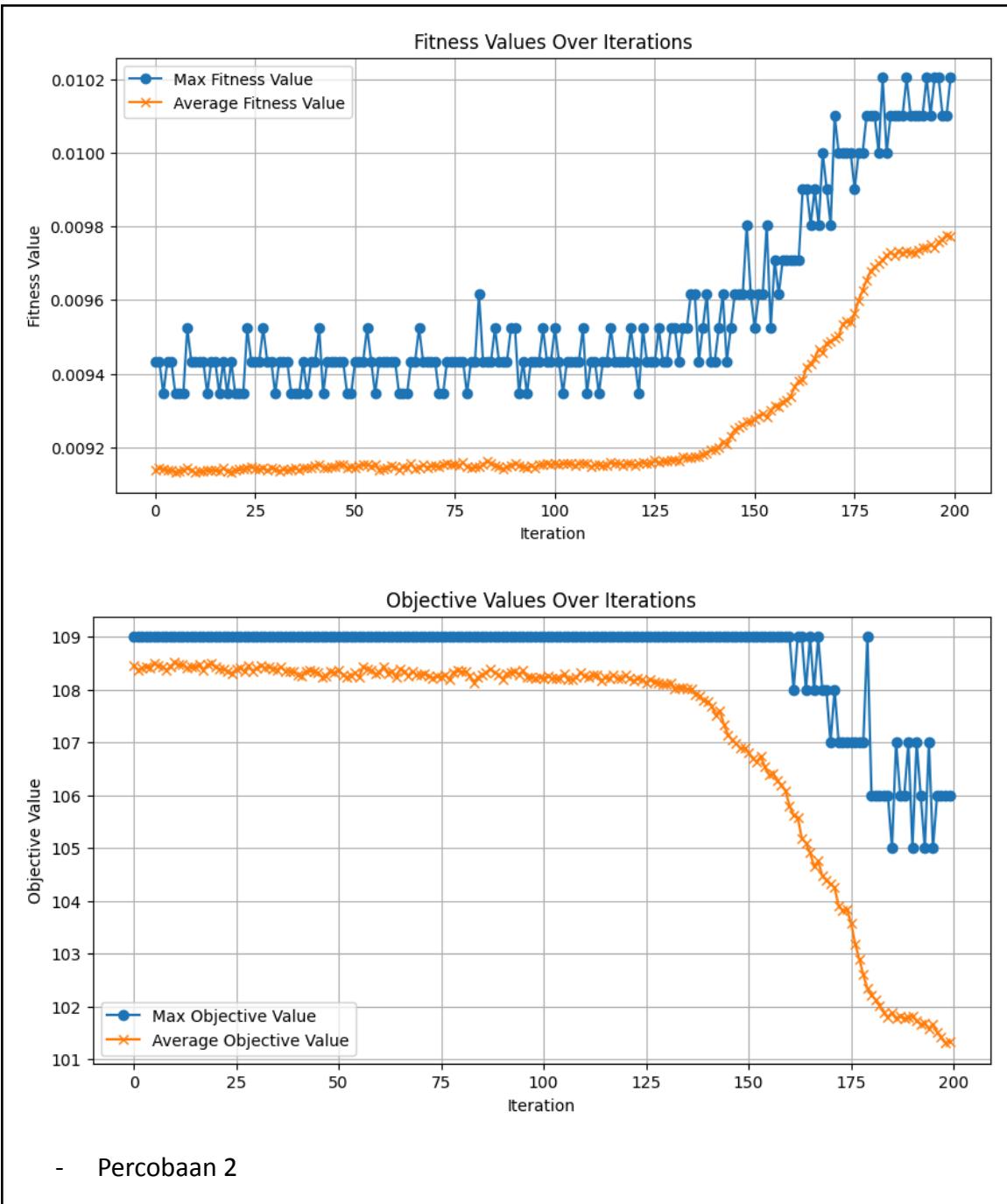


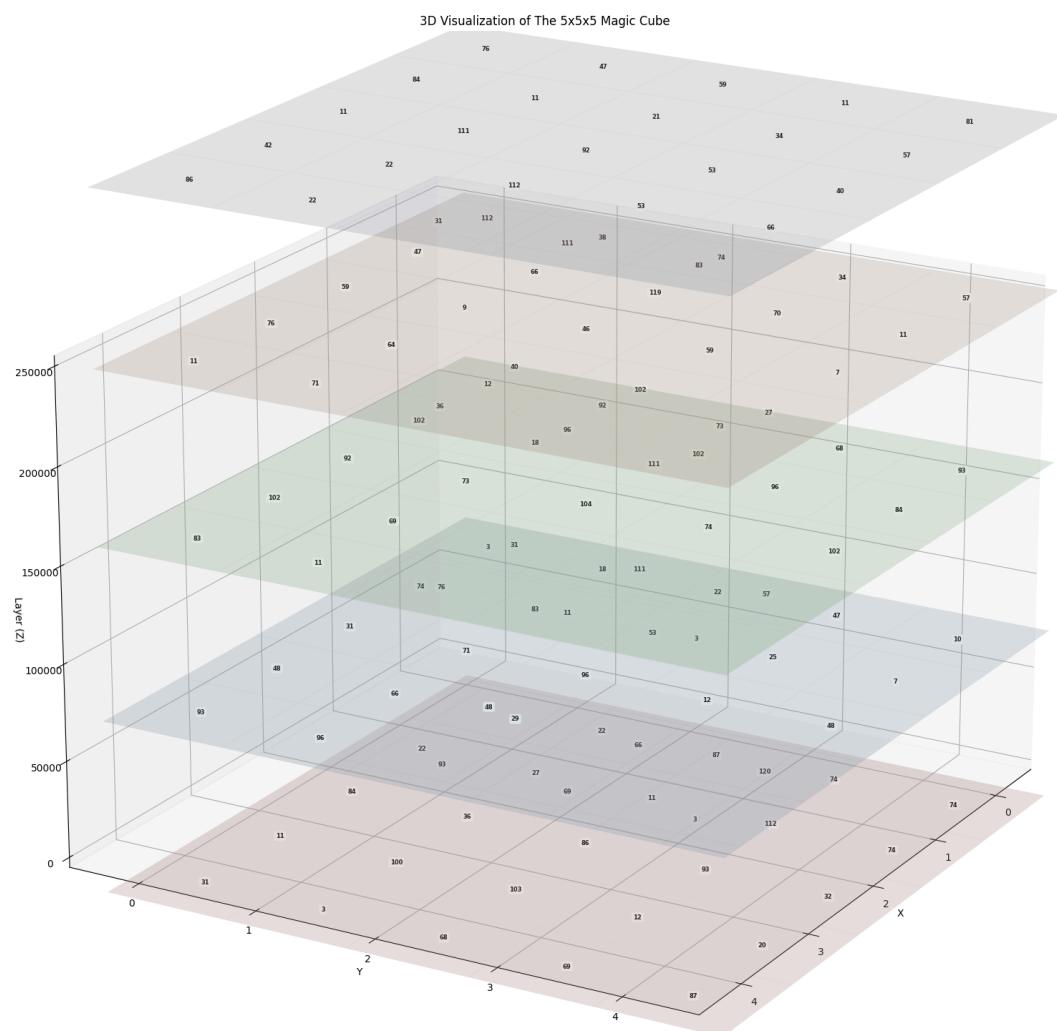


Final Objective Value: (7366, 107)
Population Size: 300
Iterations: 100
Duration: 56.720948934555054 seconds

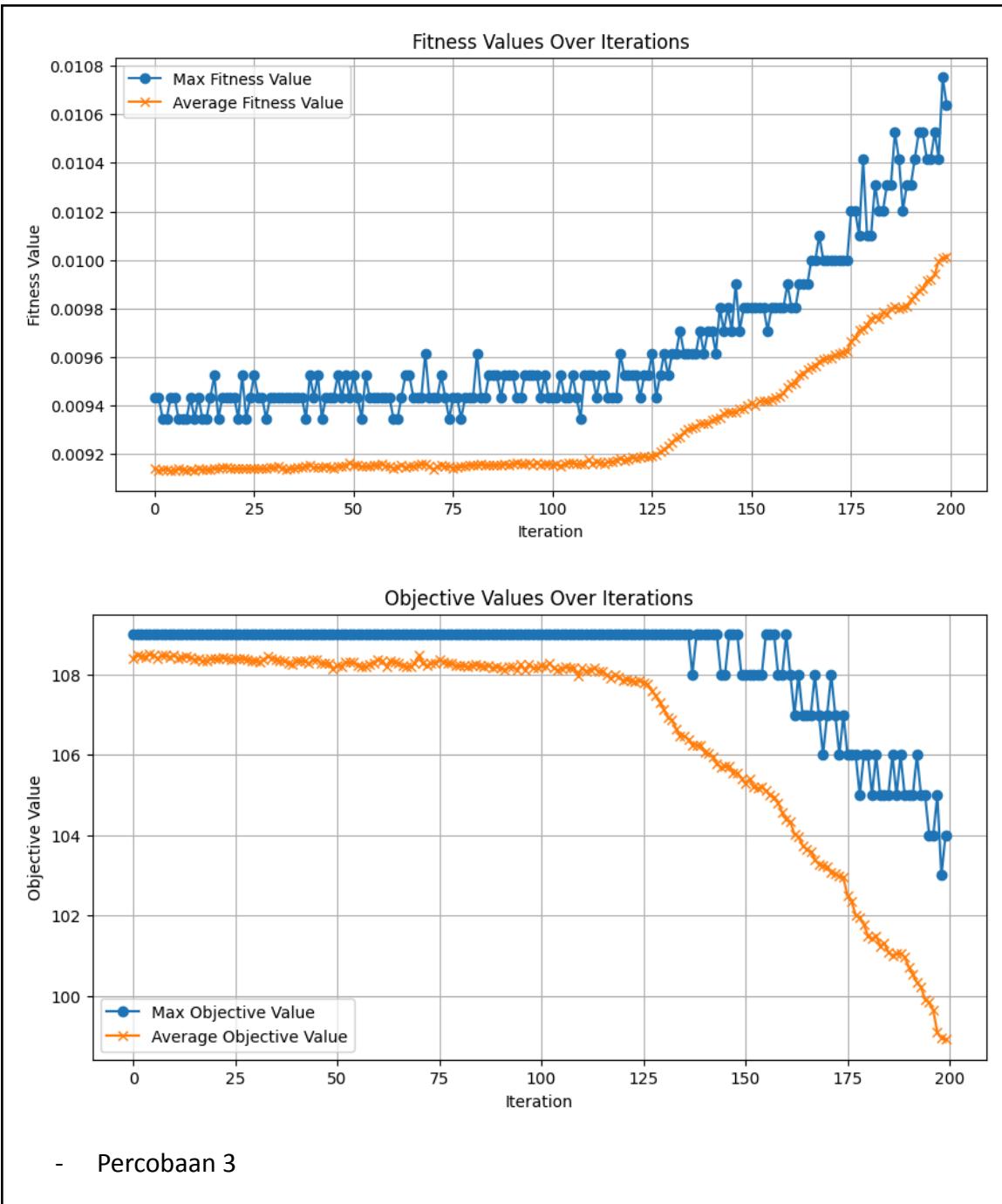


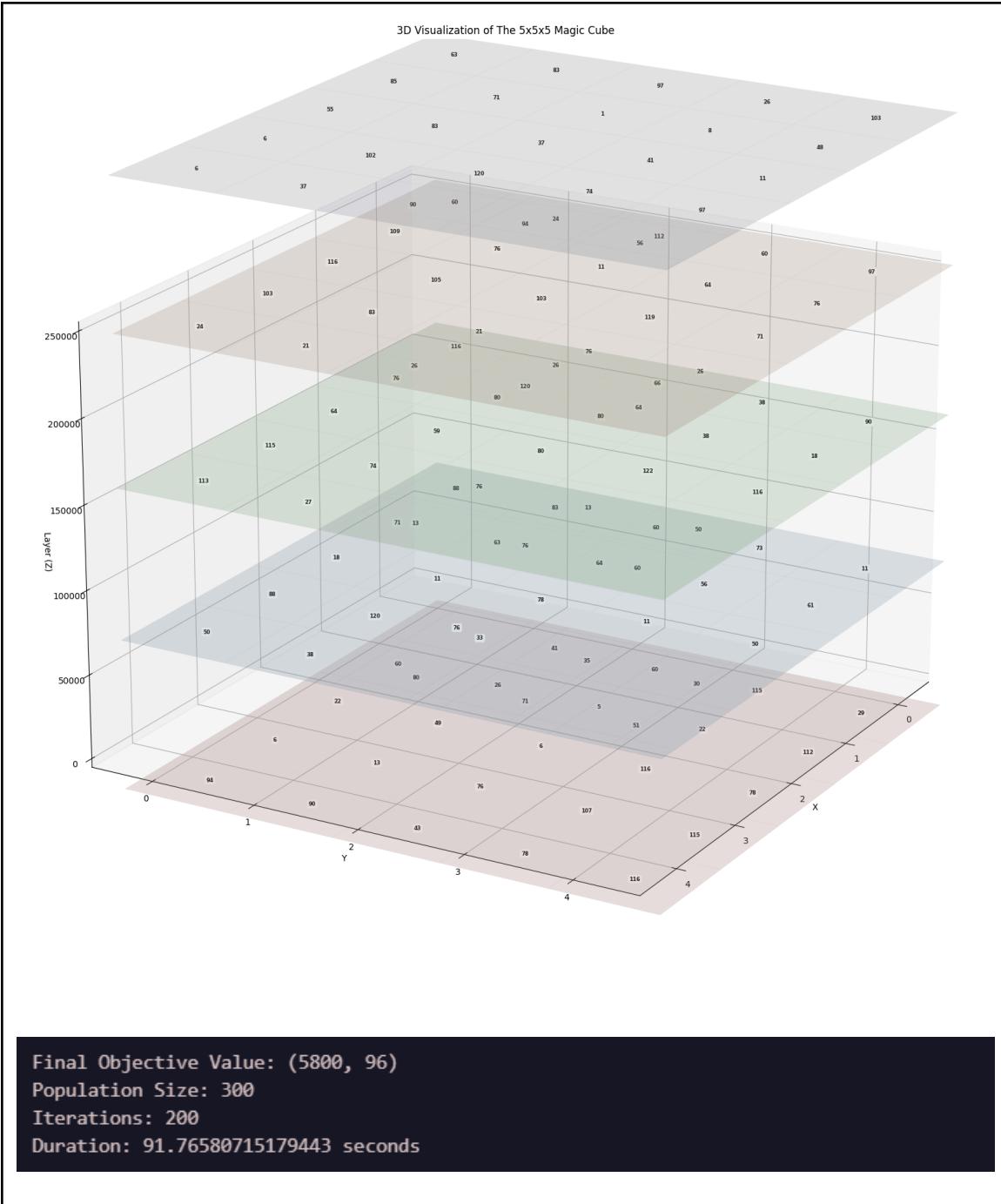


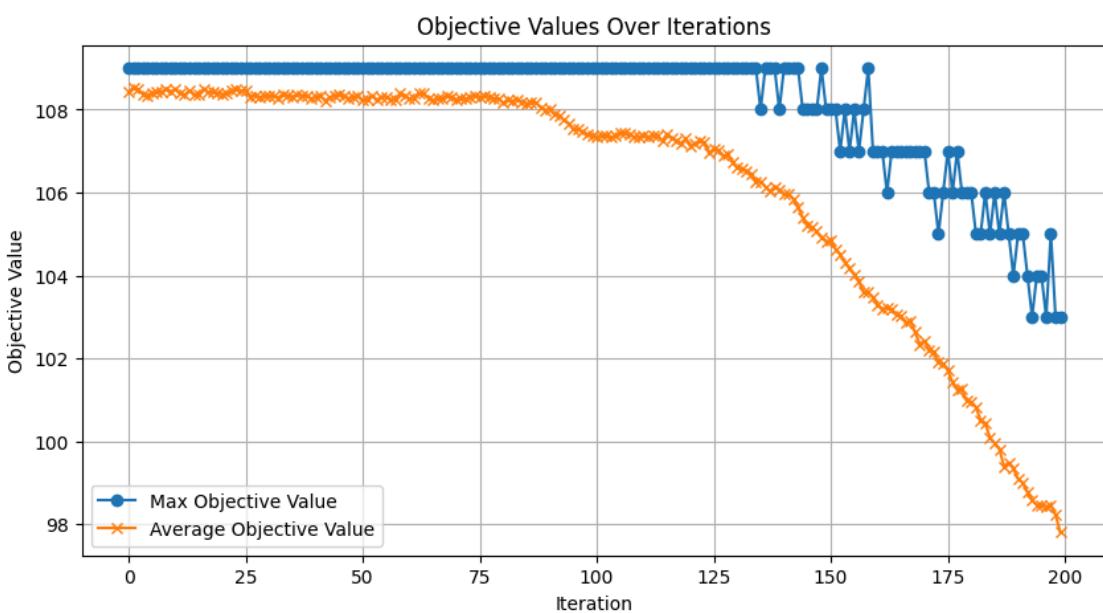
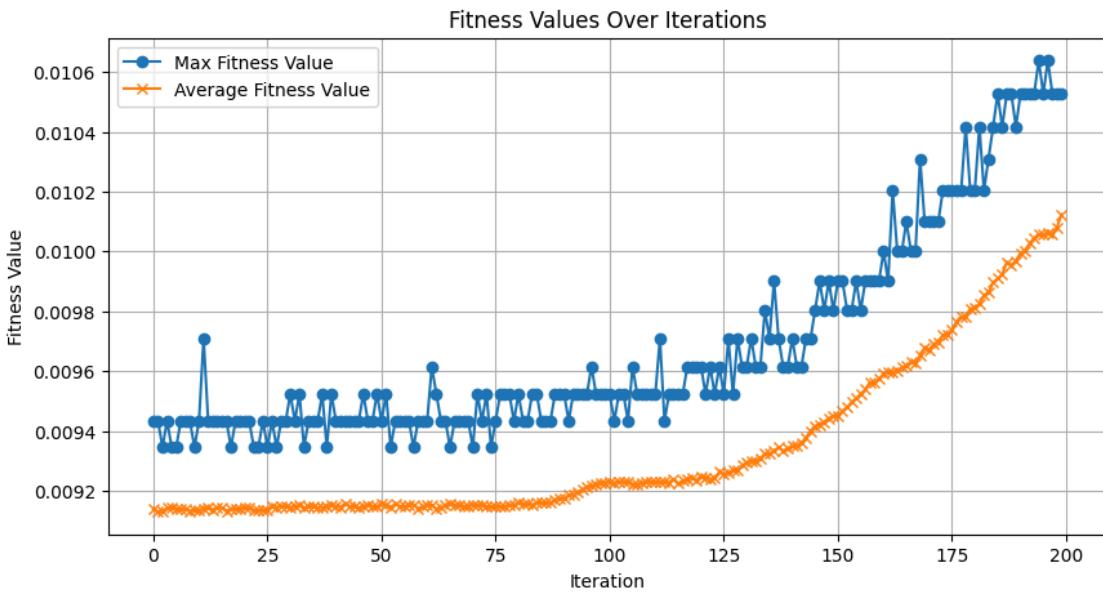




Final Objective Value: (5787, 99)
Population Size: 300
Iterations: 200
Duration: 101.90136241912842 seconds

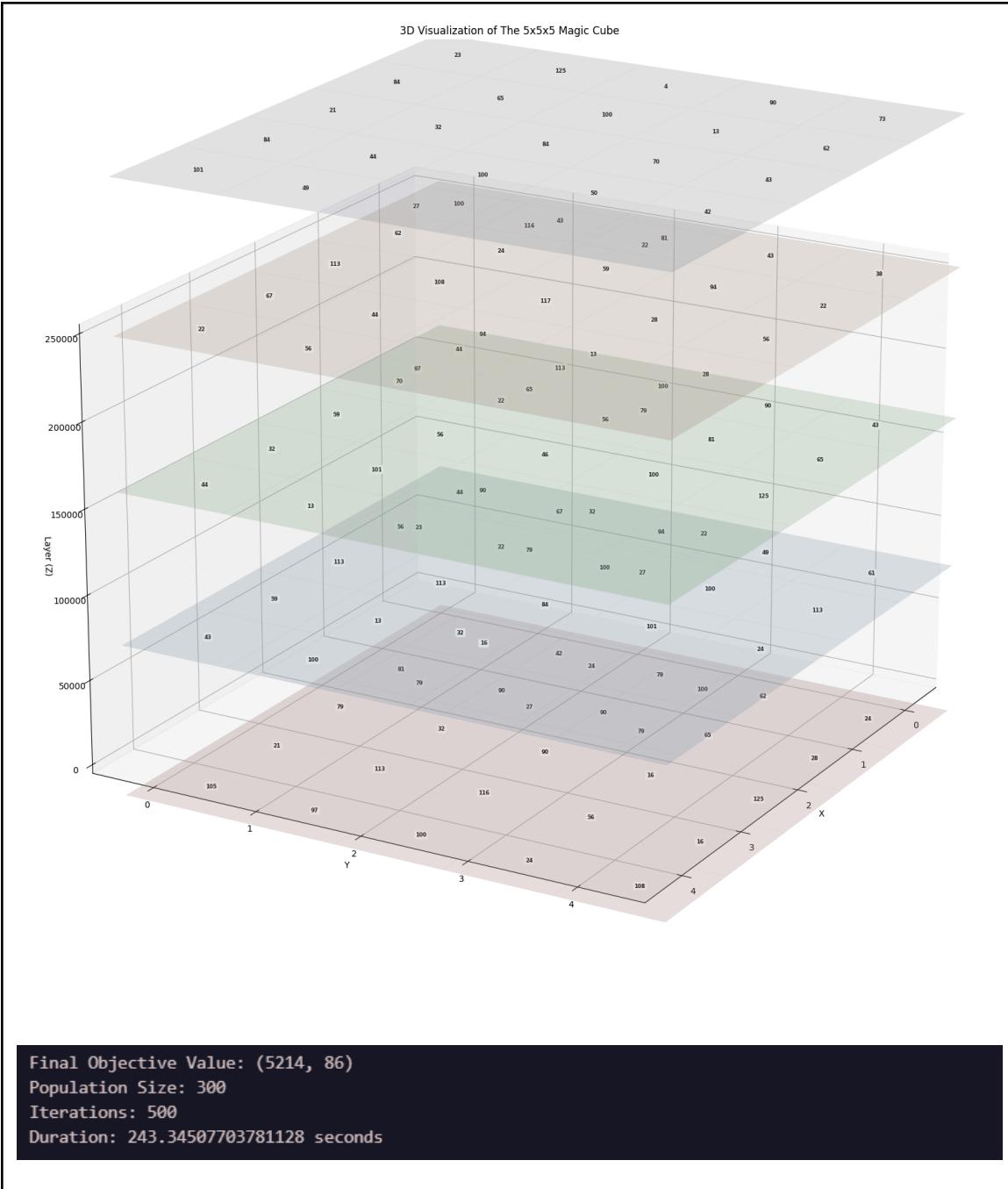


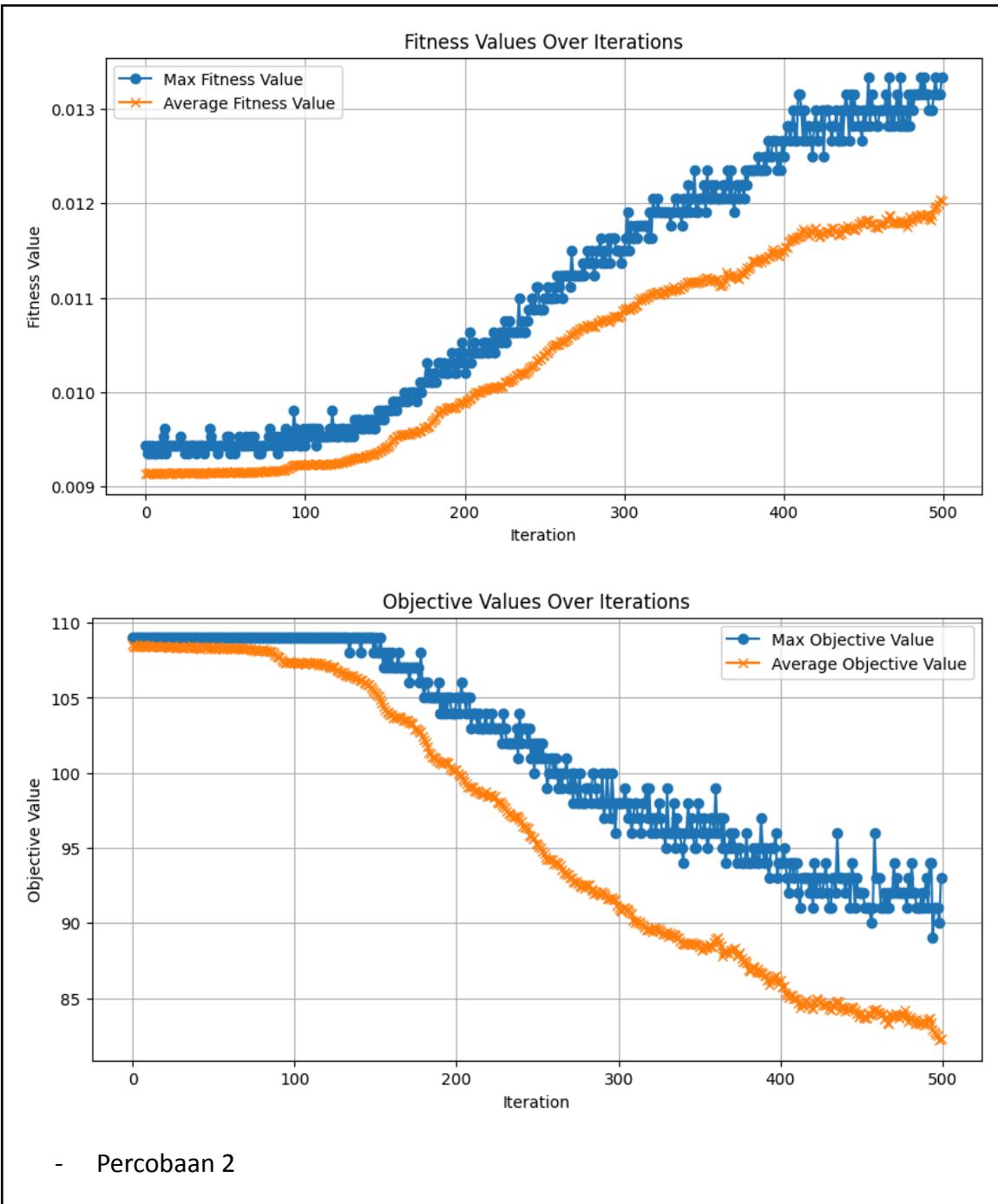


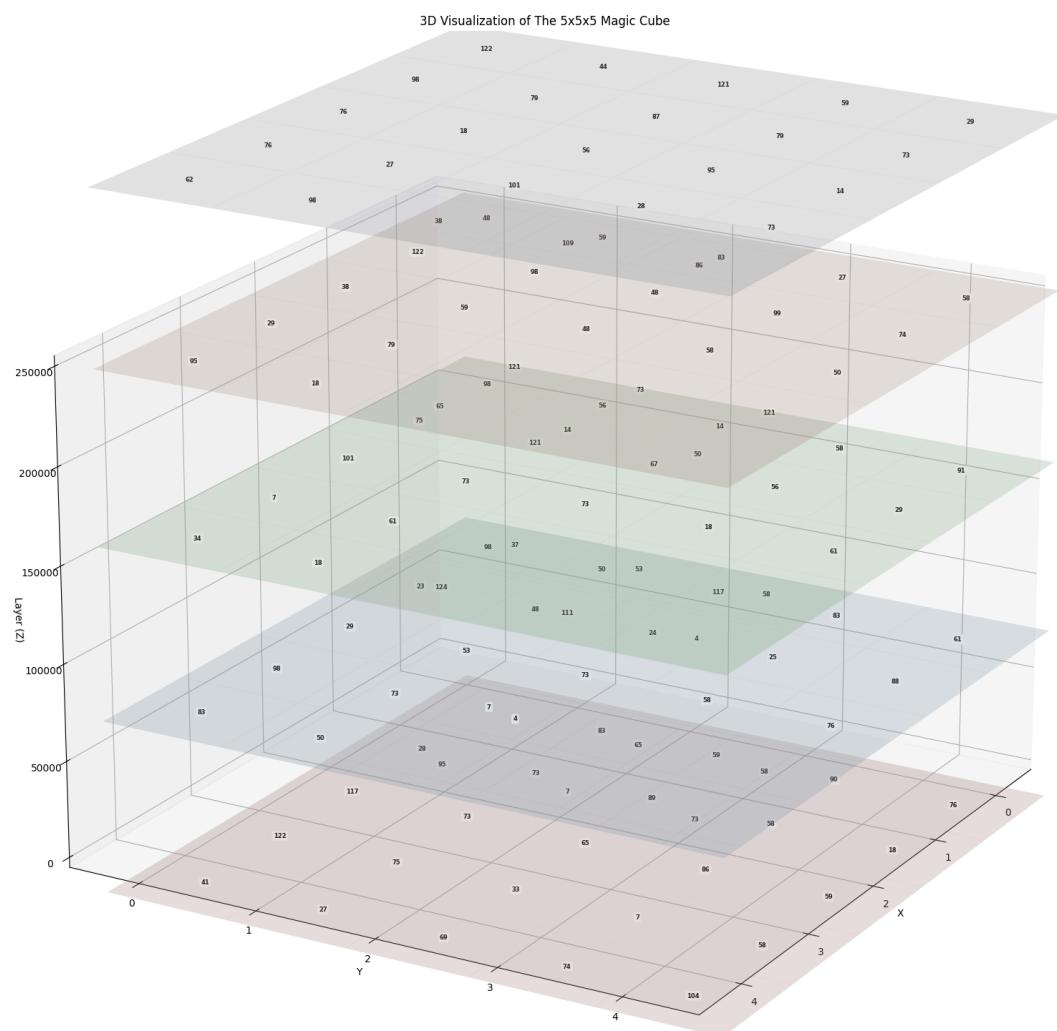


Variasi 3: 300, 500

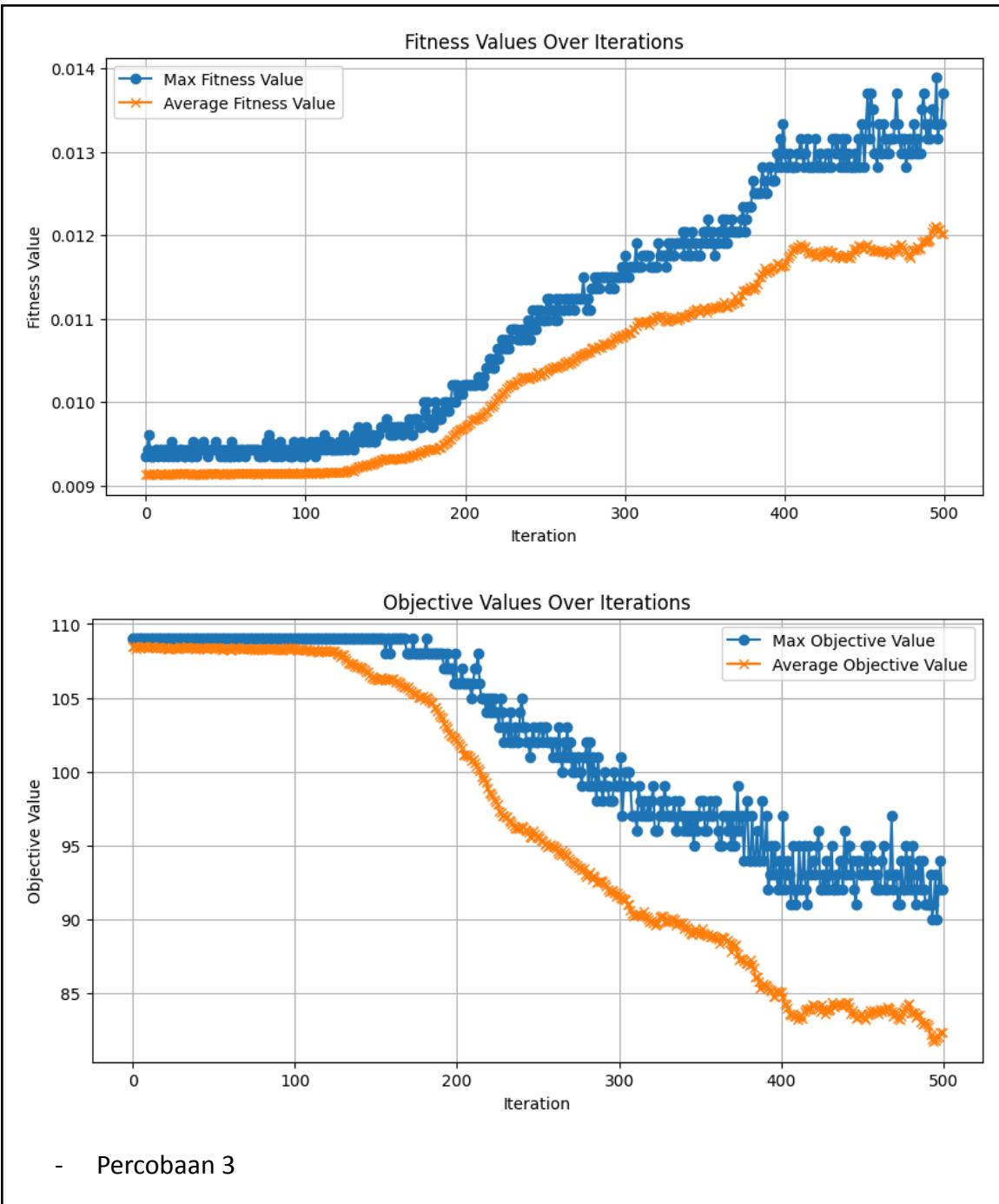
- Percobaan 1

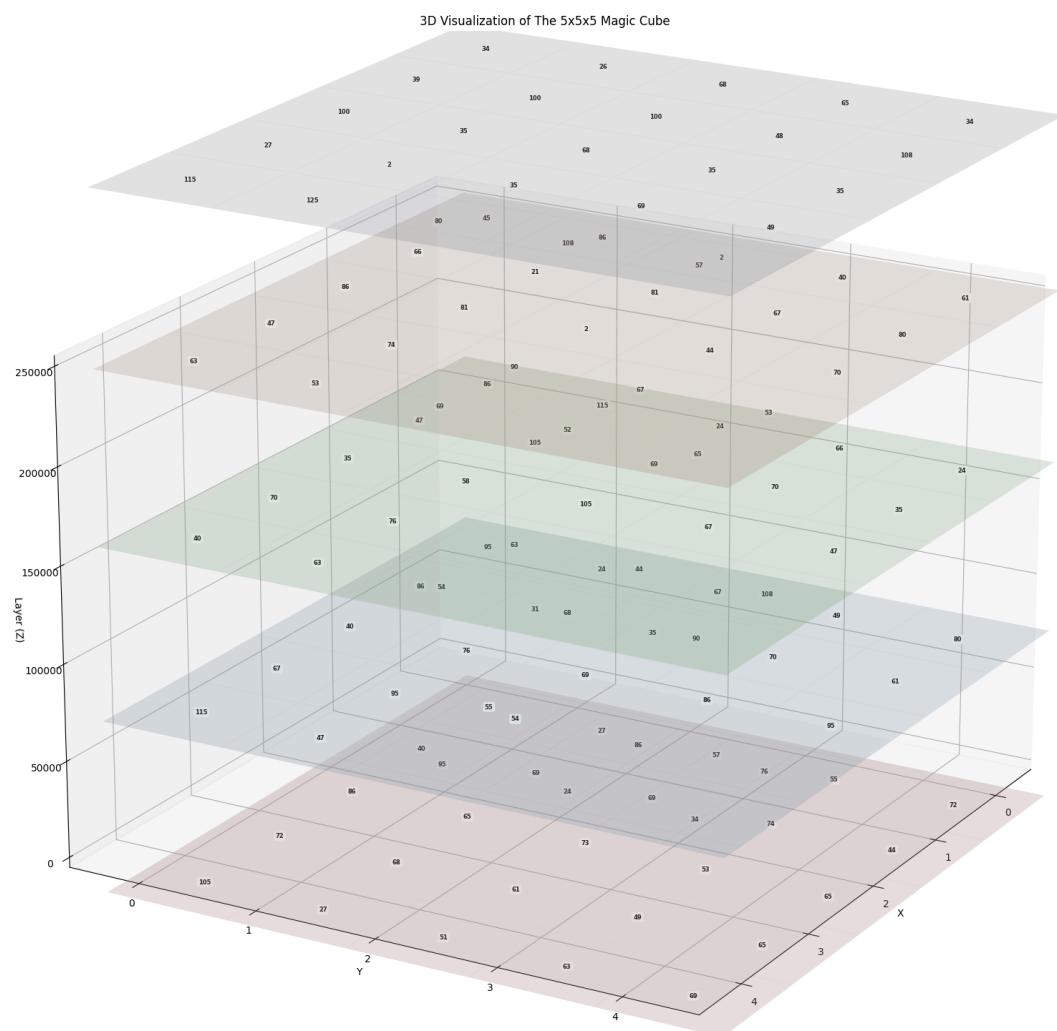




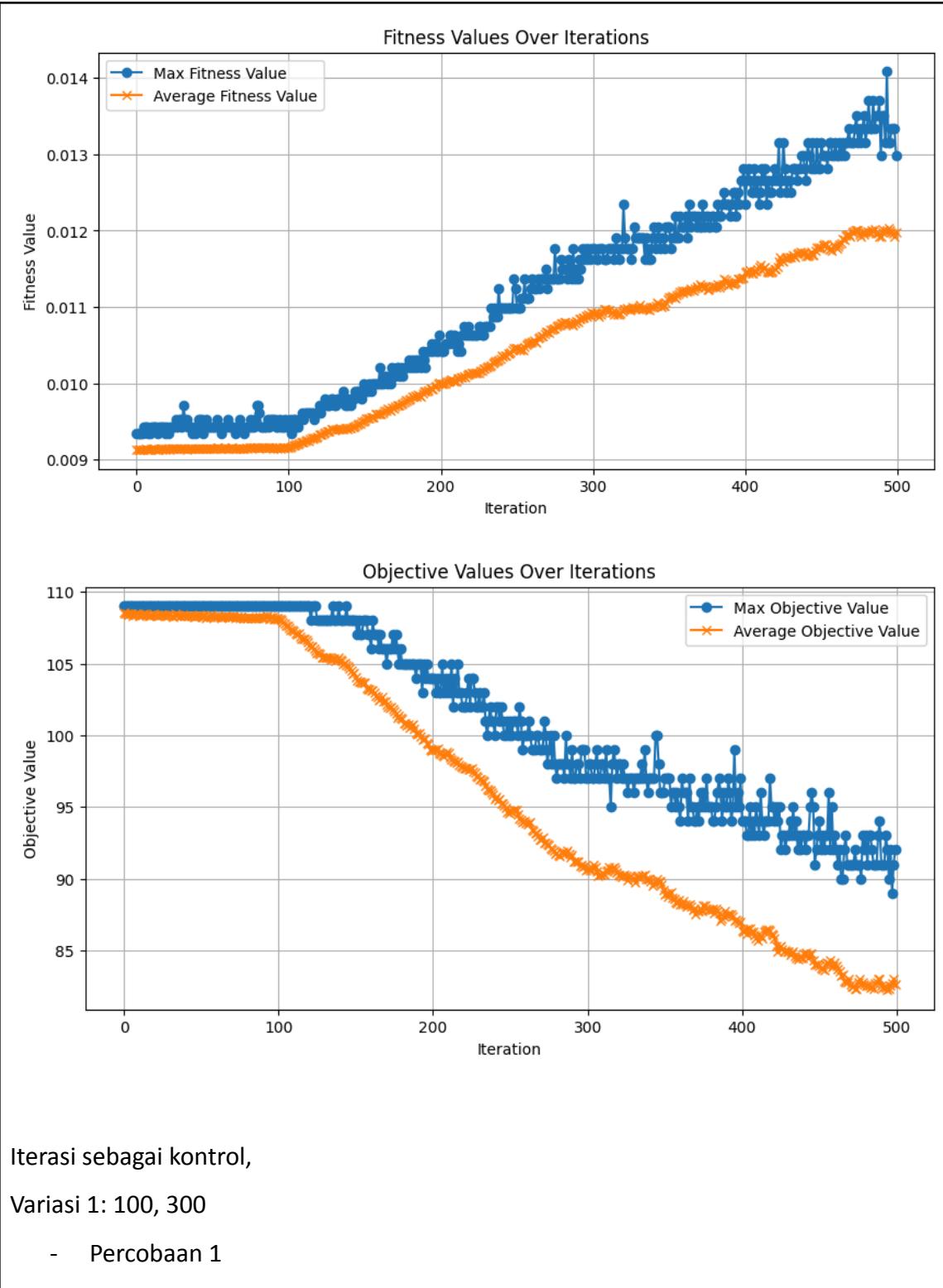


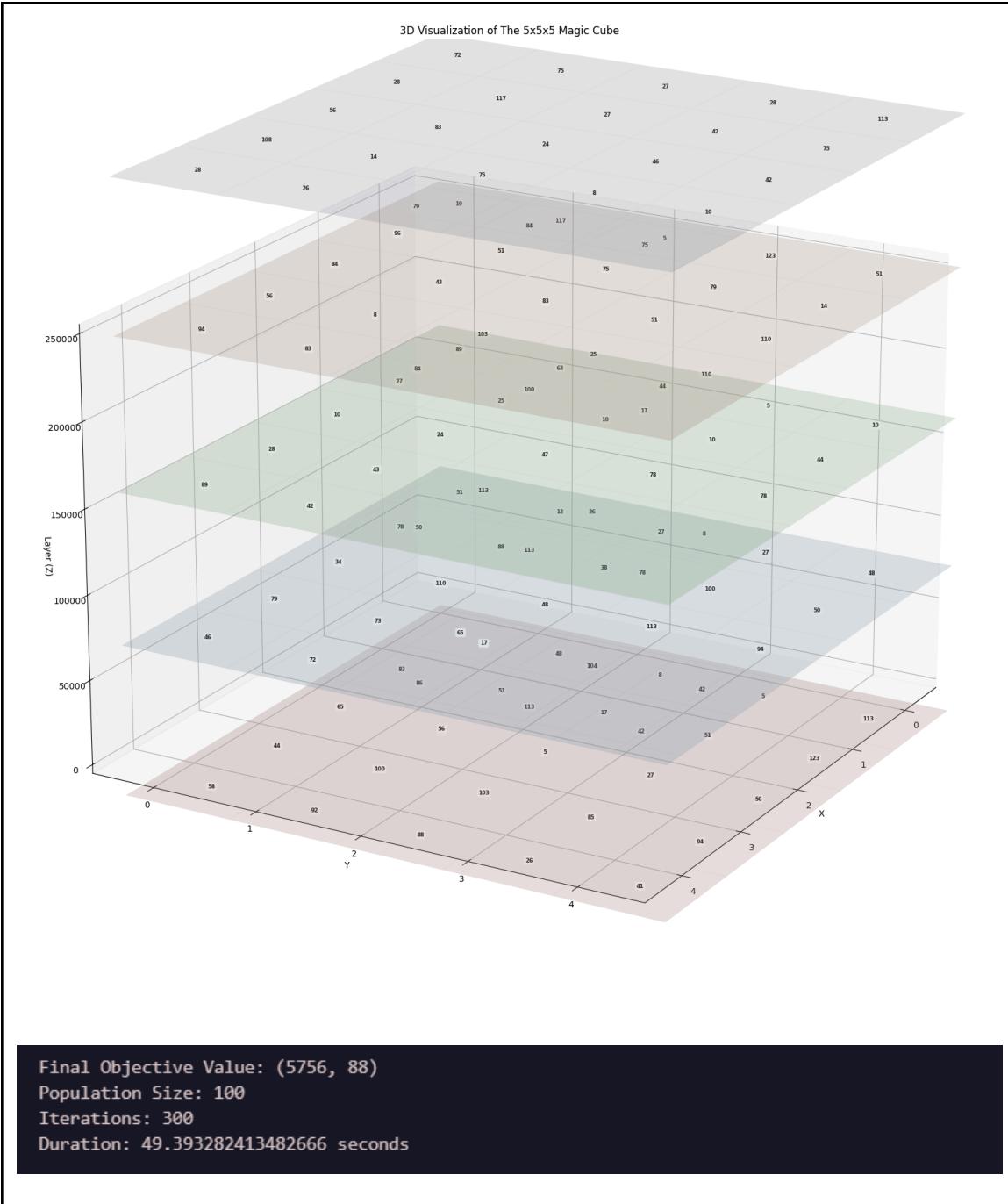
Final Objective Value: (4146, 78)
Population Size: 300
Iterations: 500
Duration: 239.5650634765625 seconds

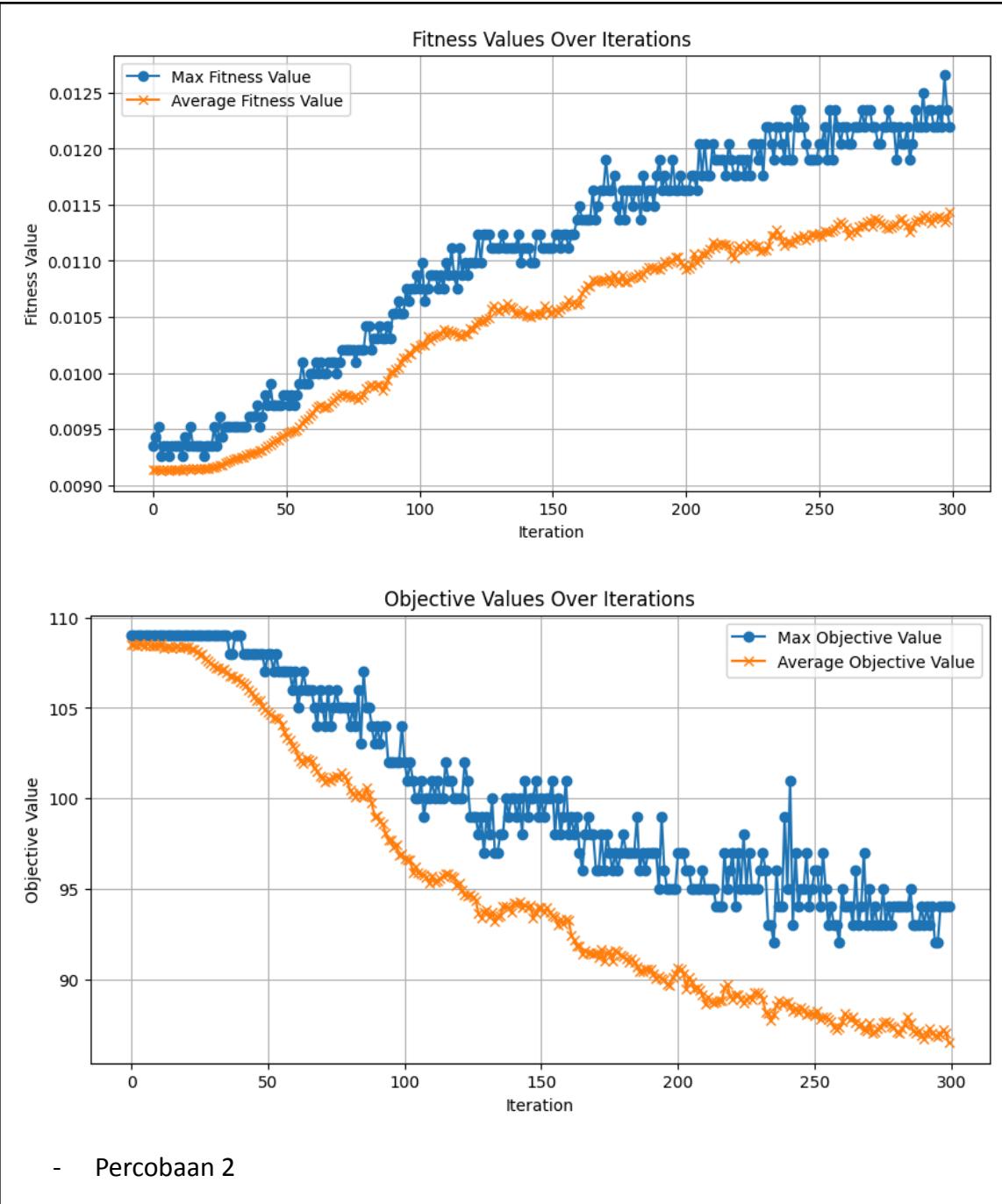


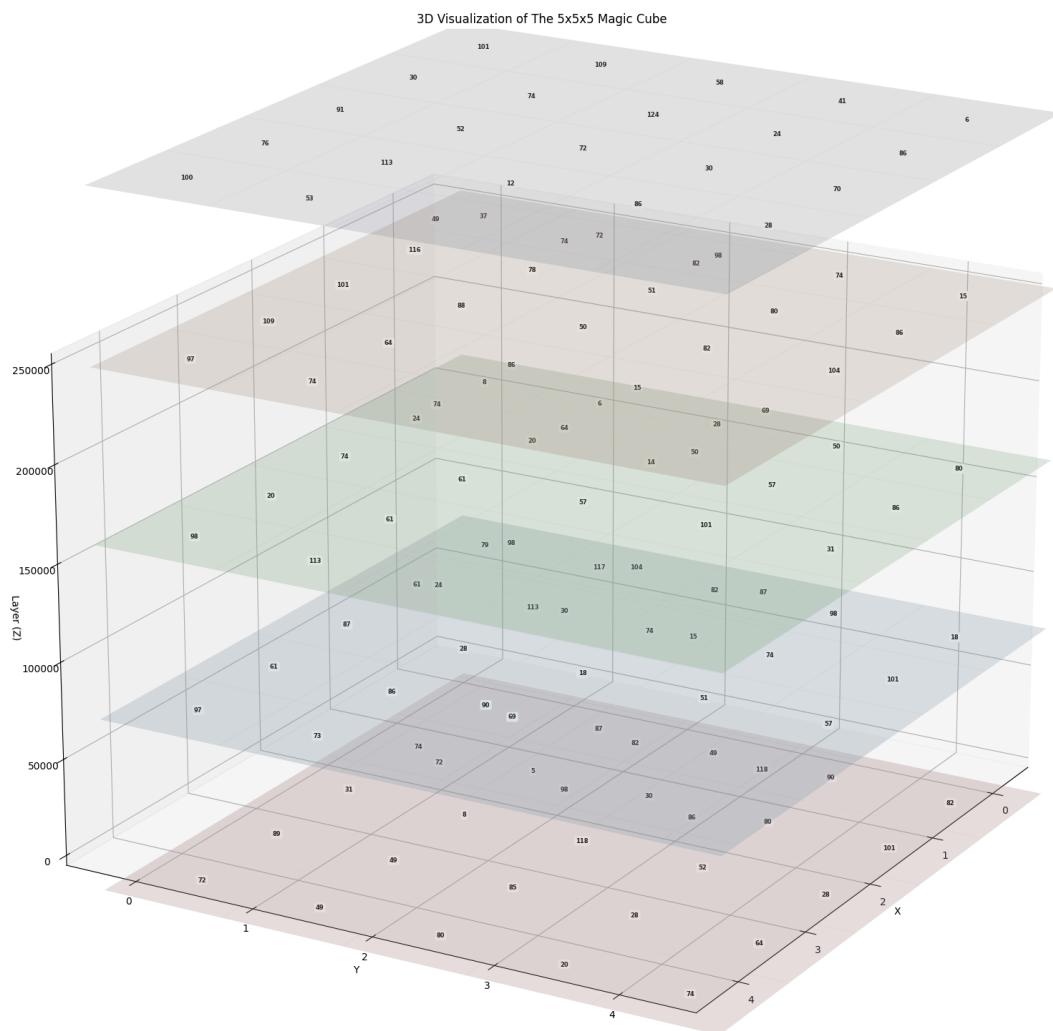


Final Objective Value: (3807, 79)
Population Size: 300
Iterations: 500
Duration: 261.3715353012085 seconds

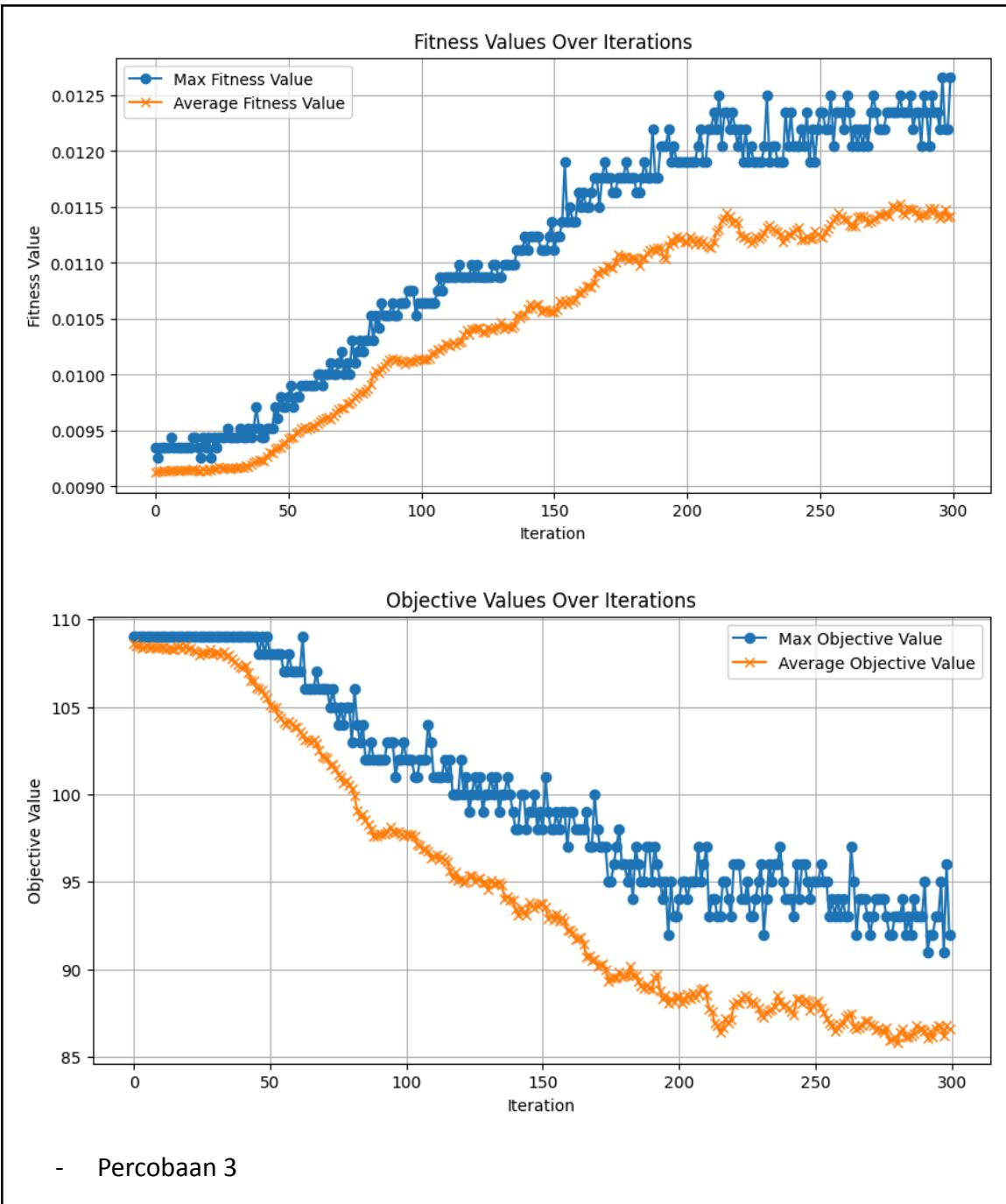


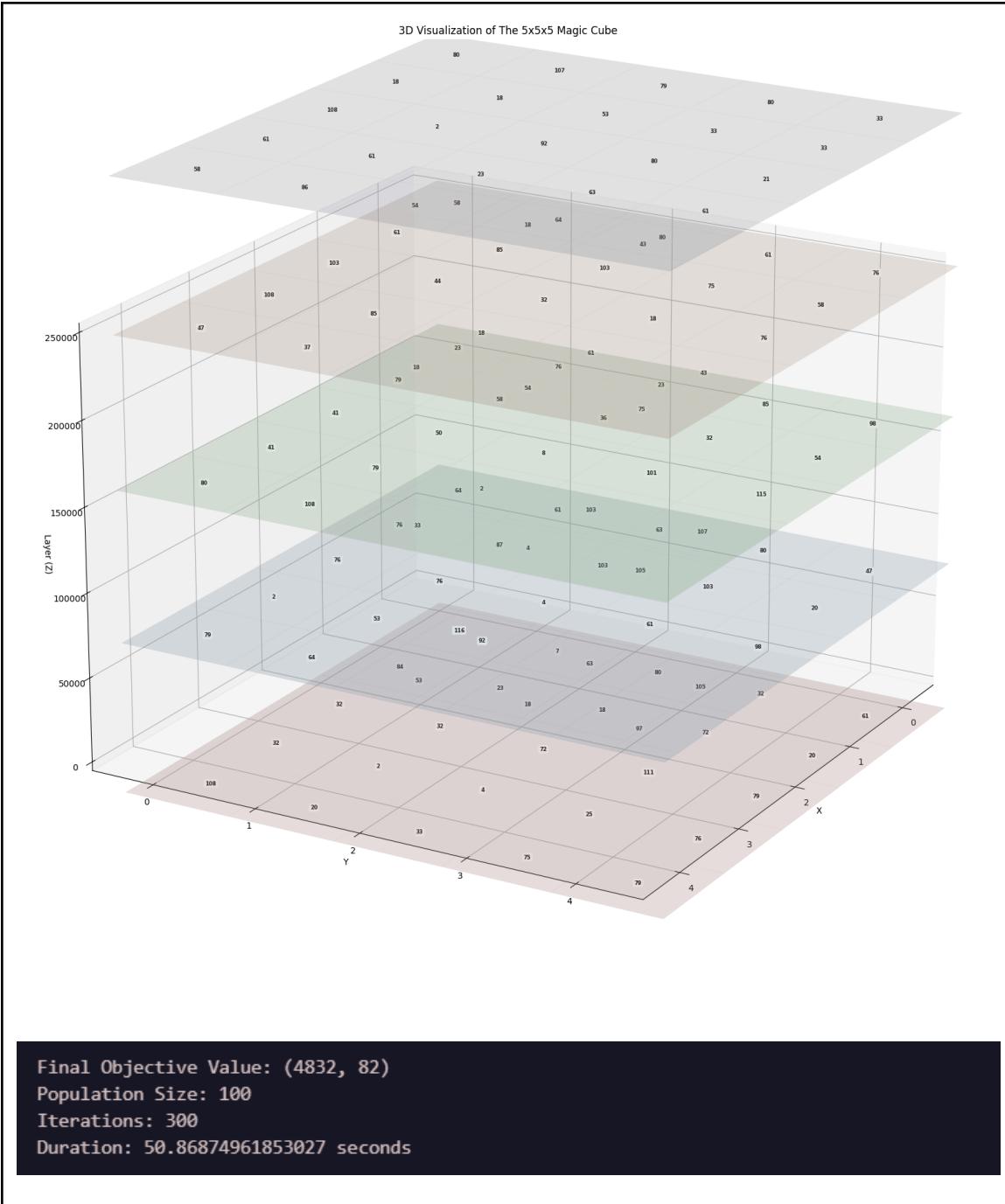


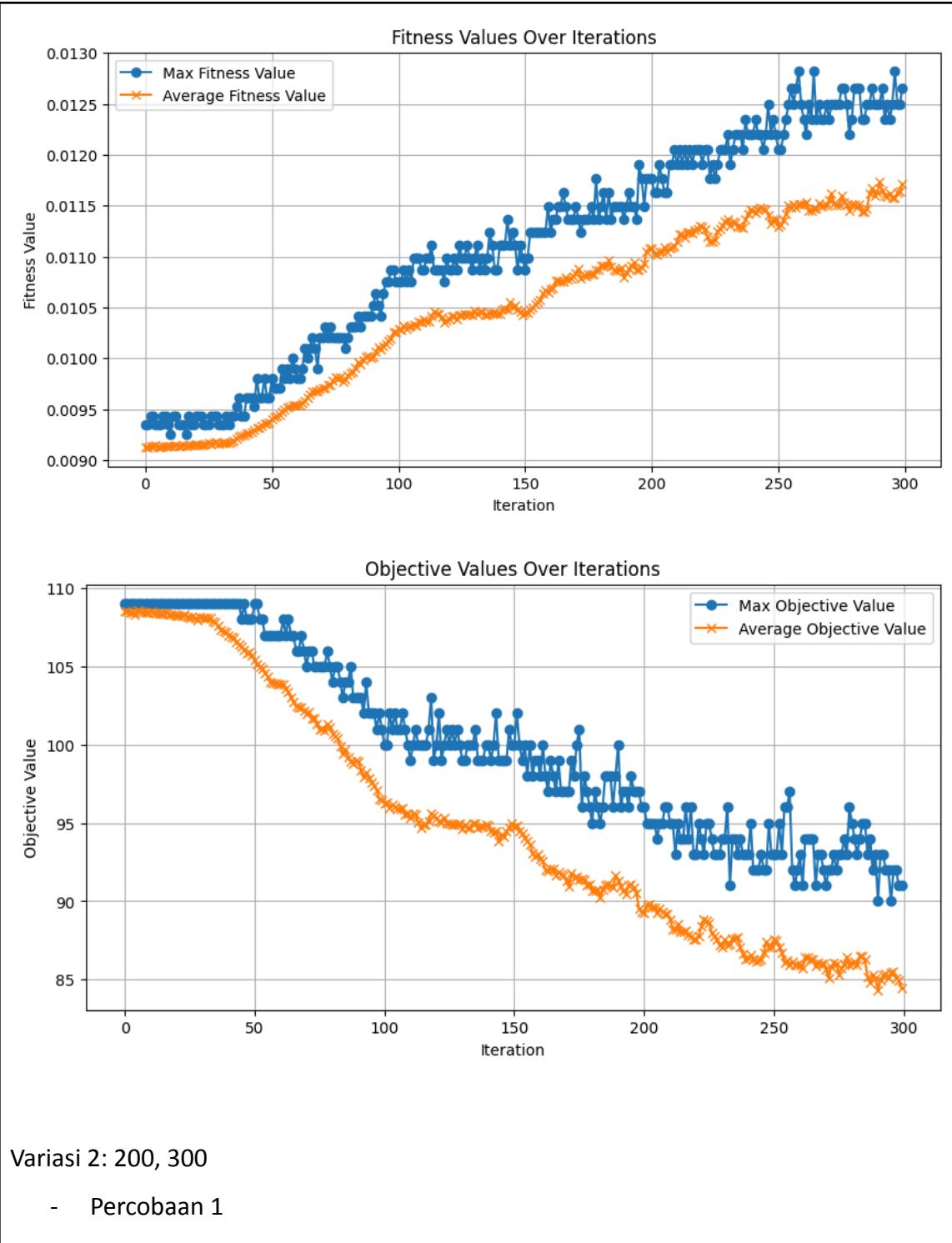


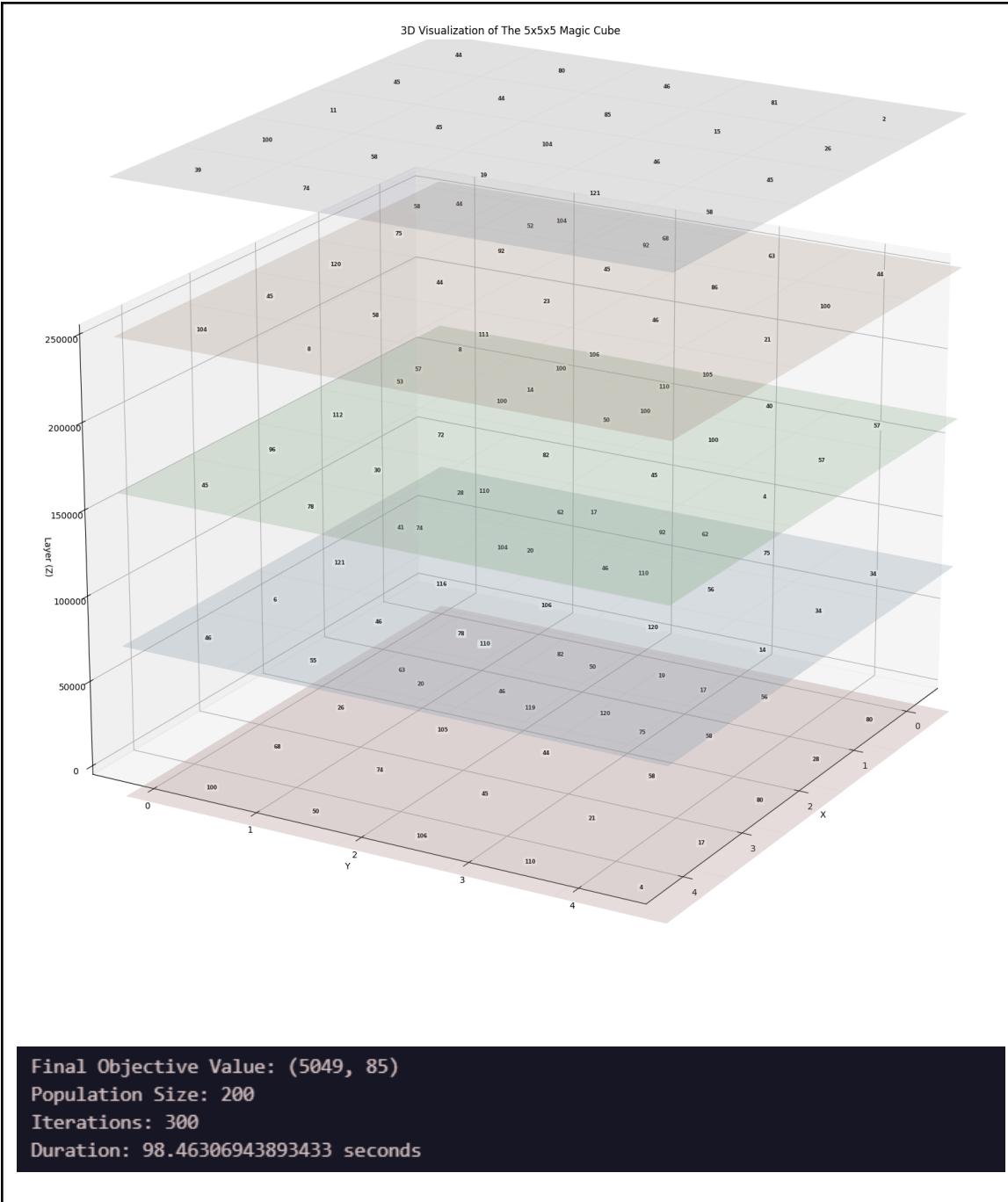


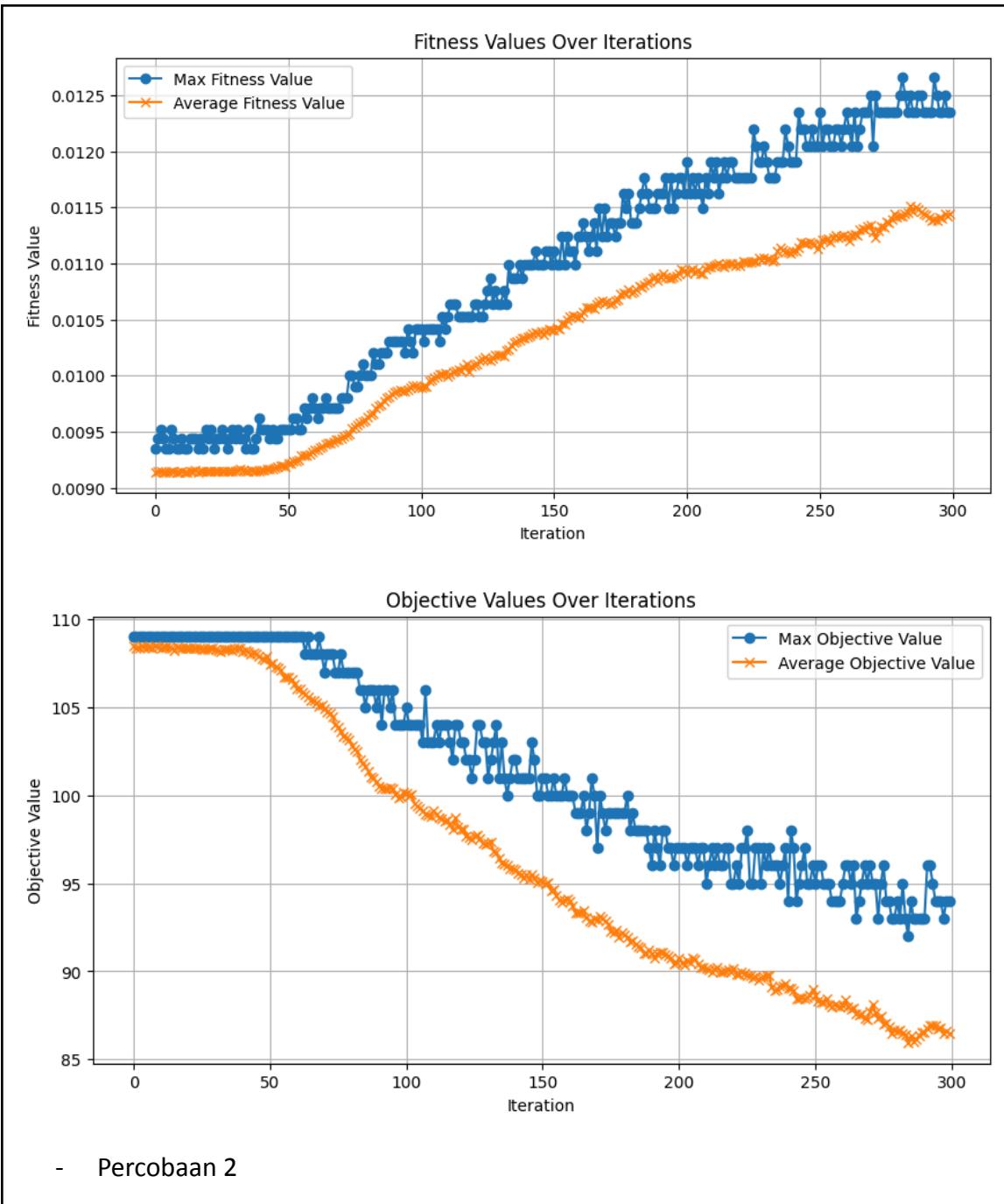
```
Final Objective Value: (5231, 88)
Population Size: 100
Iterations: 300
Duration: 54.4514536857605 seconds
```

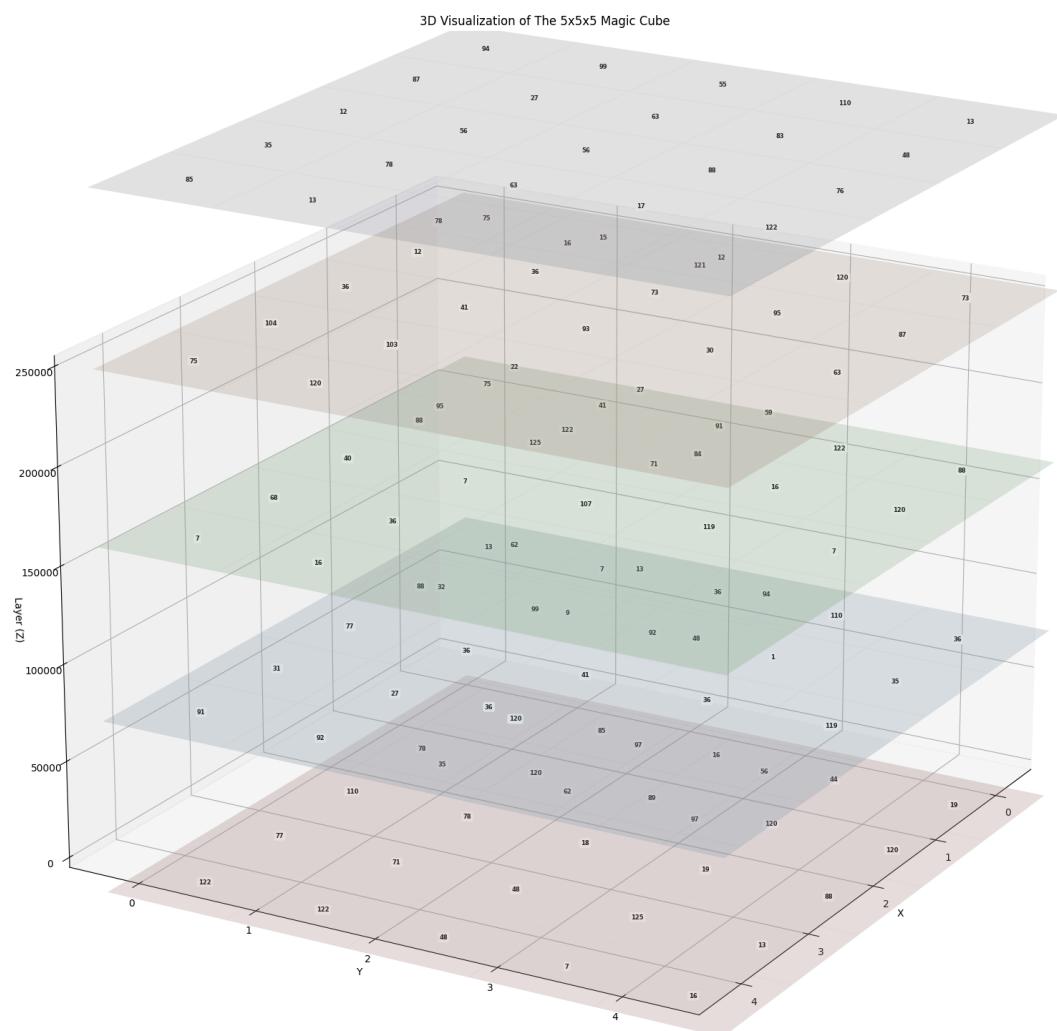




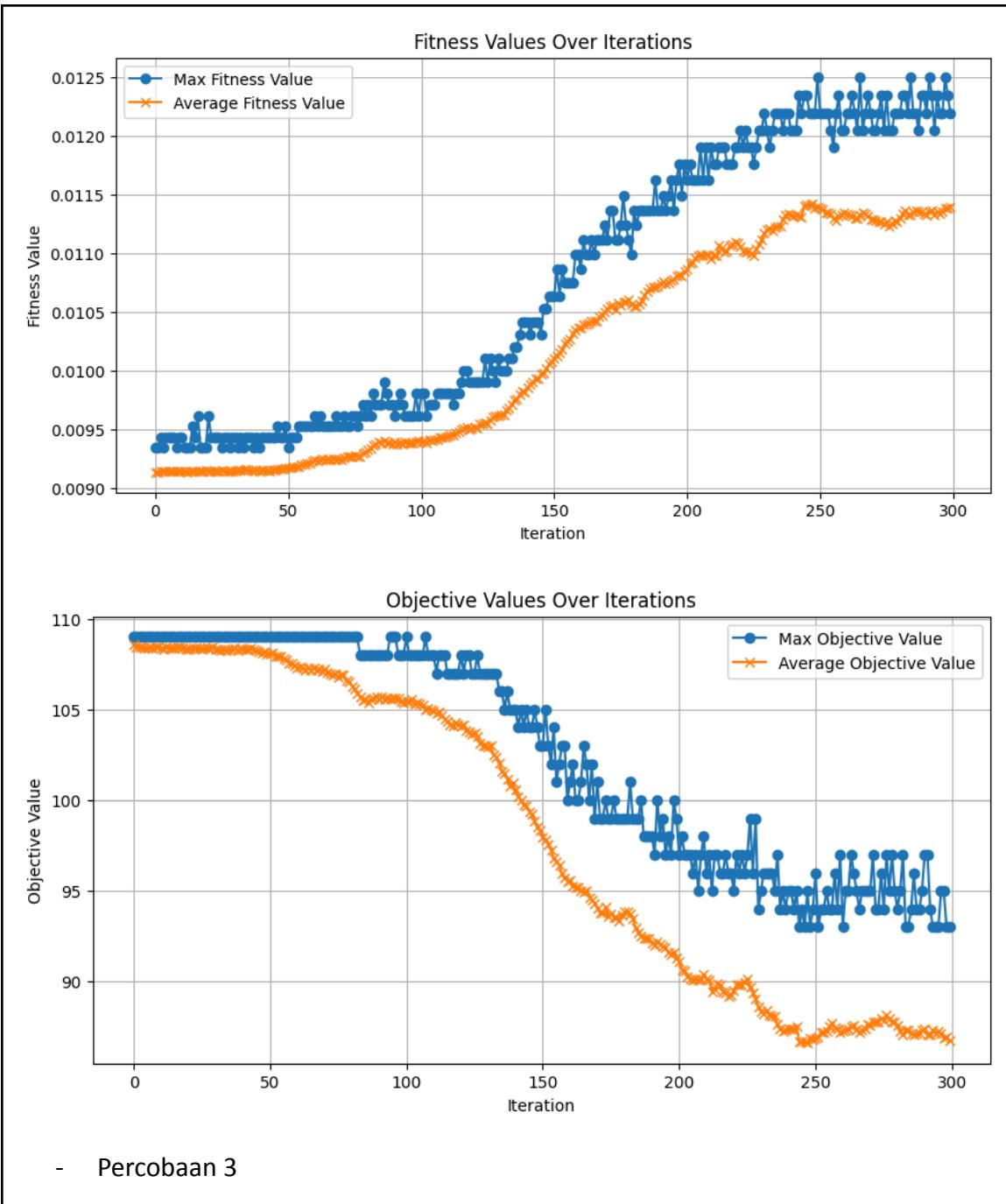


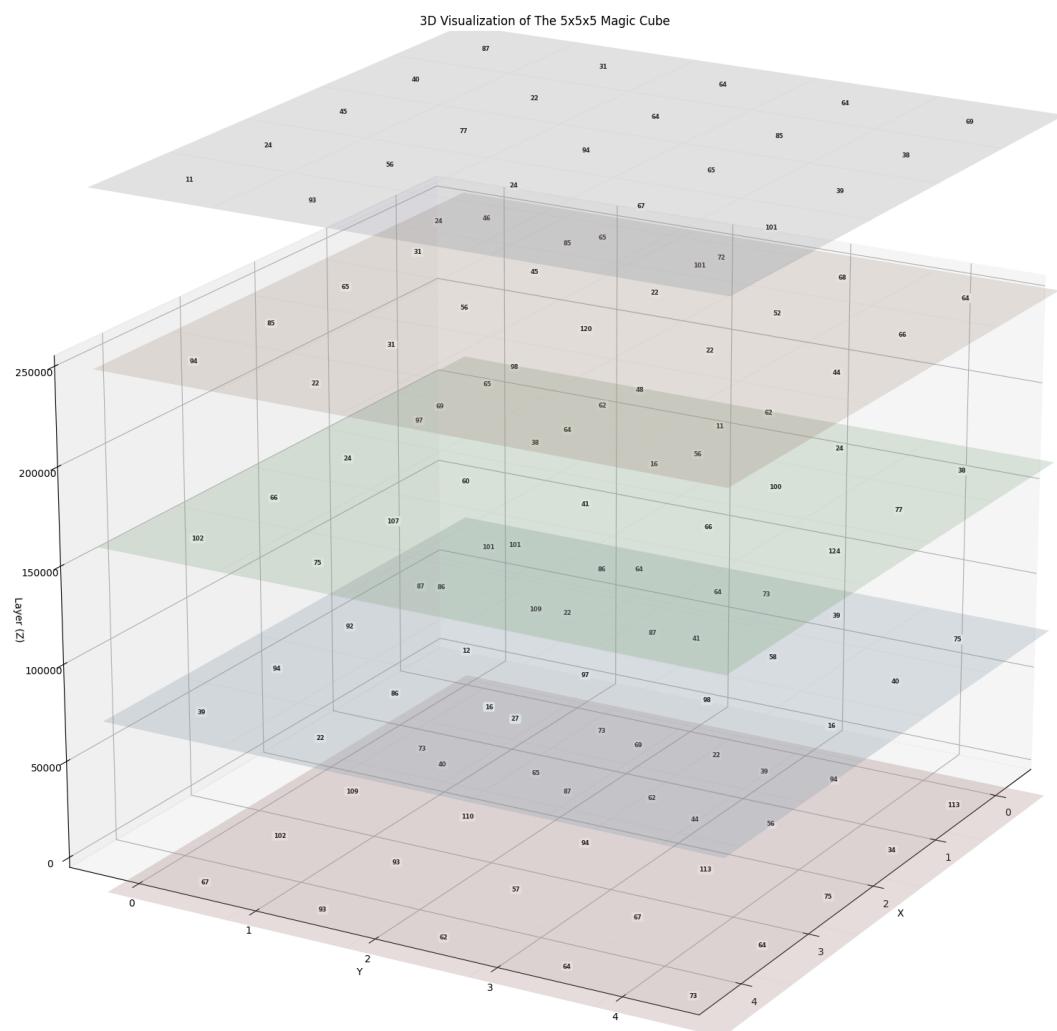




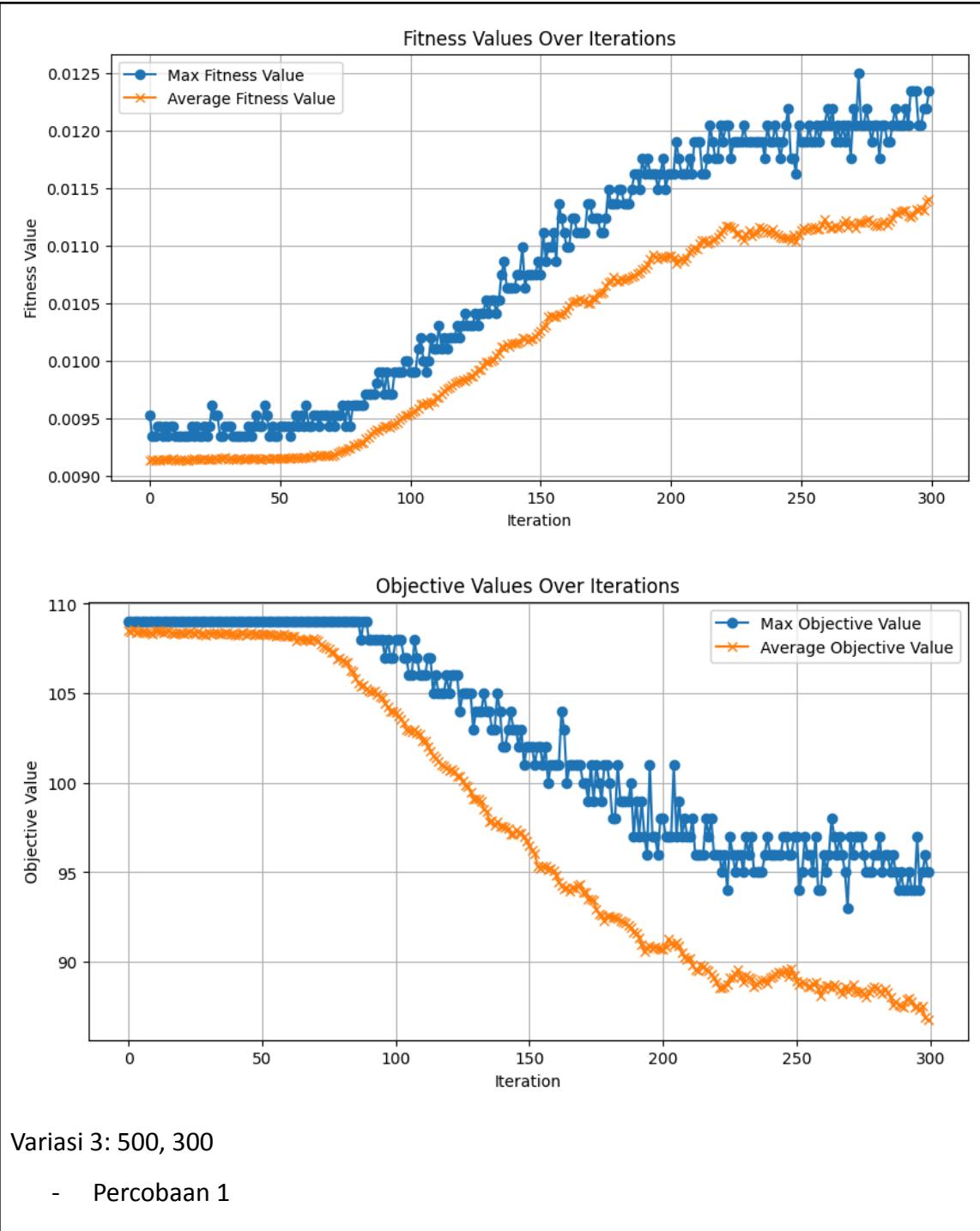


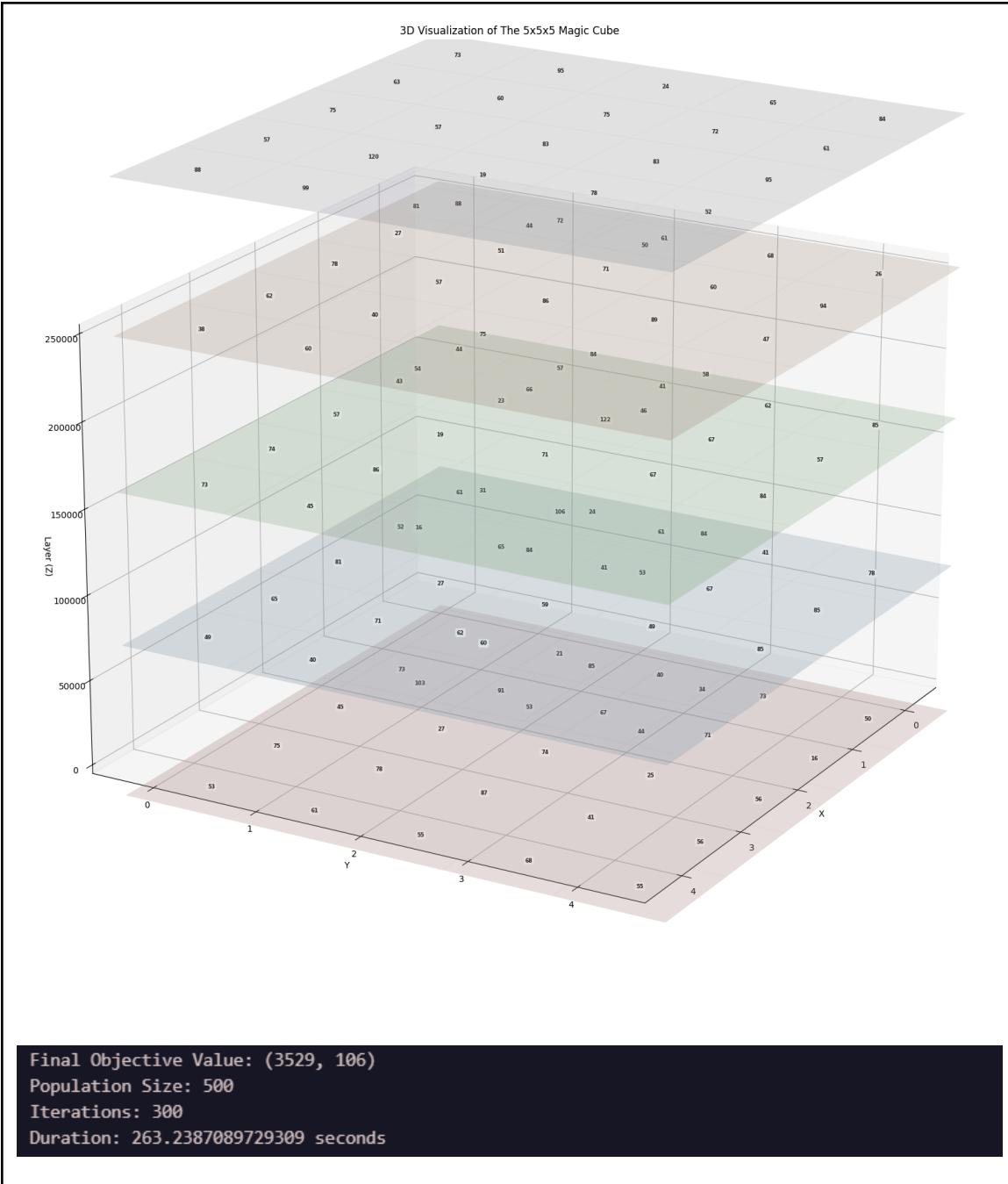
Final Objective Value: (5288, 86)
Population Size: 200
Iterations: 300
Duration: 102.96380186080933 seconds

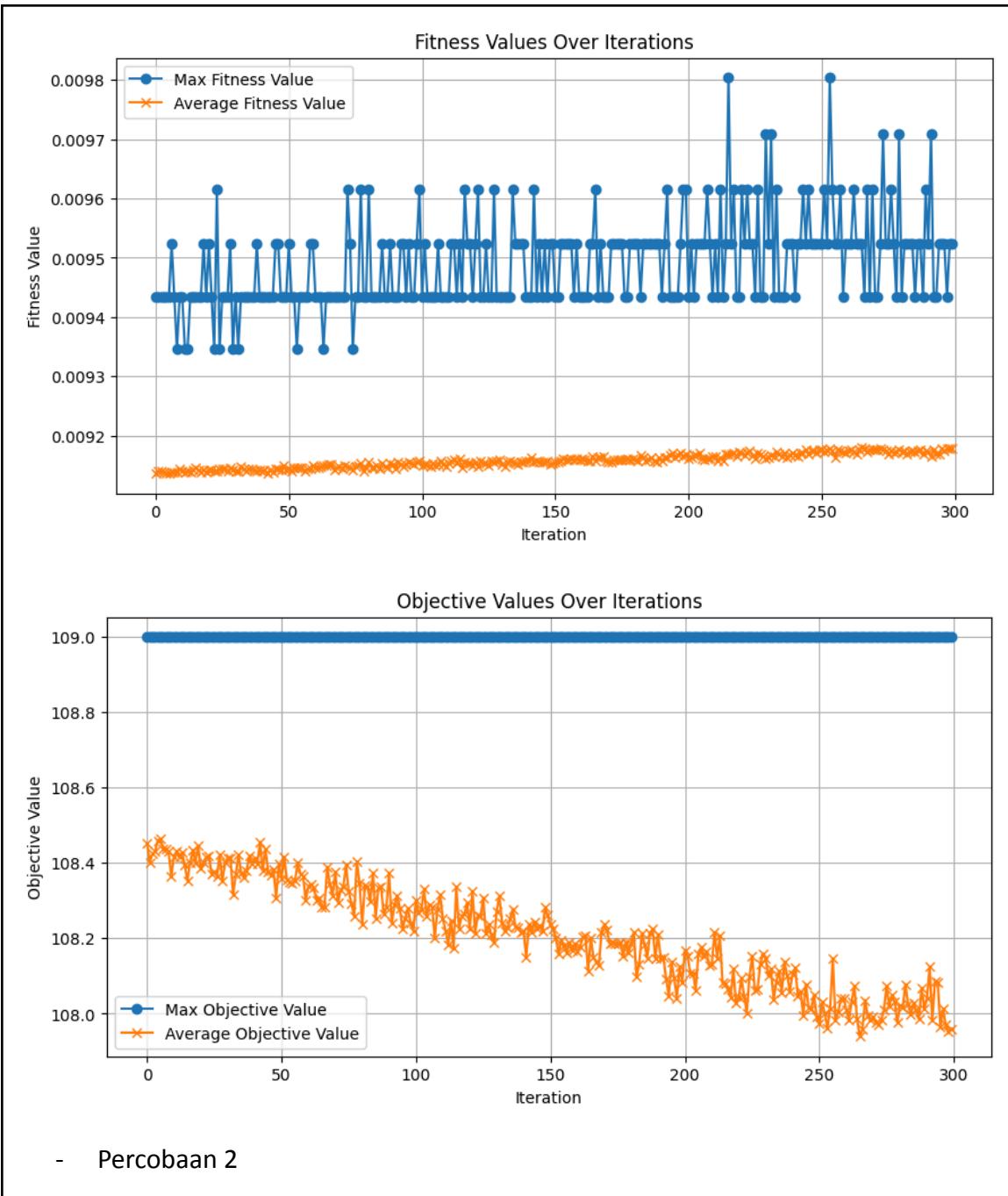


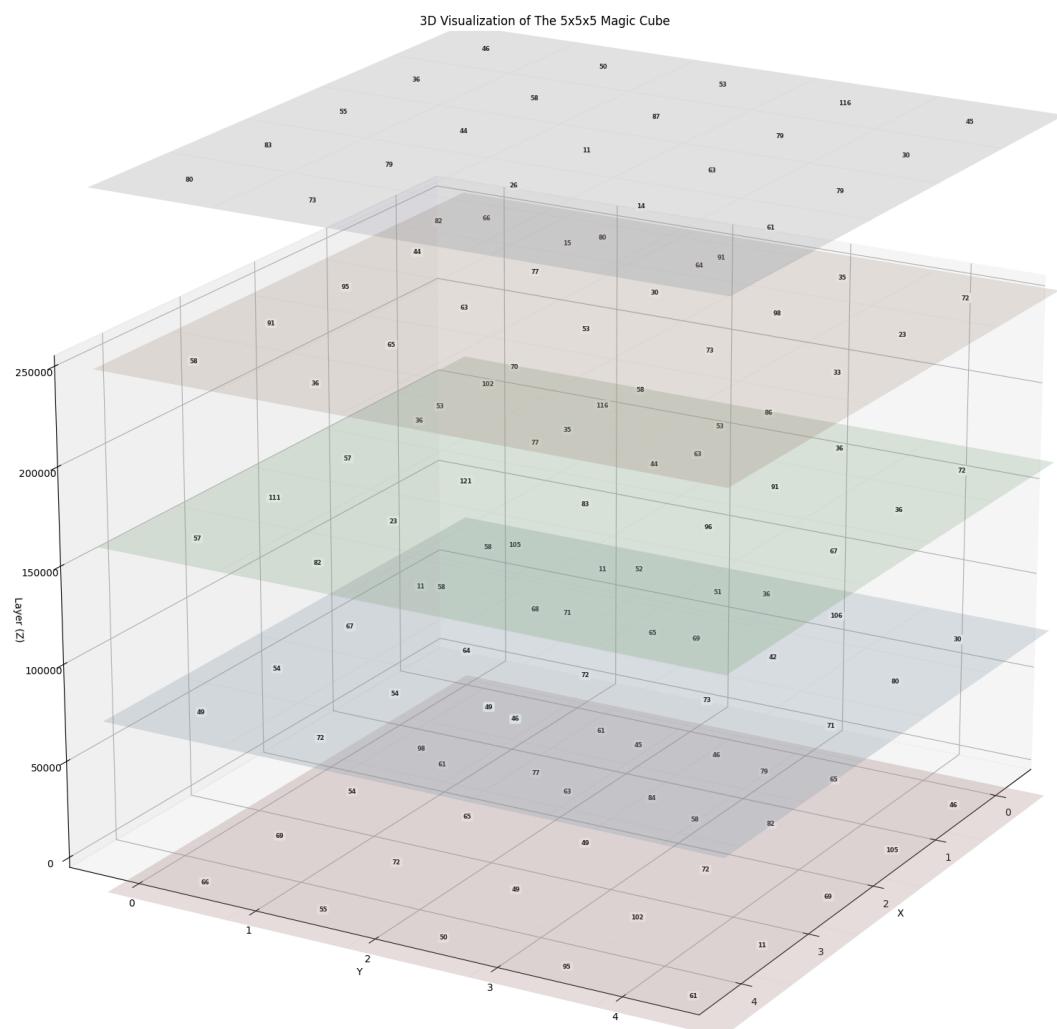


Final Objective Value: (4664, 88)
Population Size: 200
Iterations: 300
Duration: 94.4055700302124 seconds

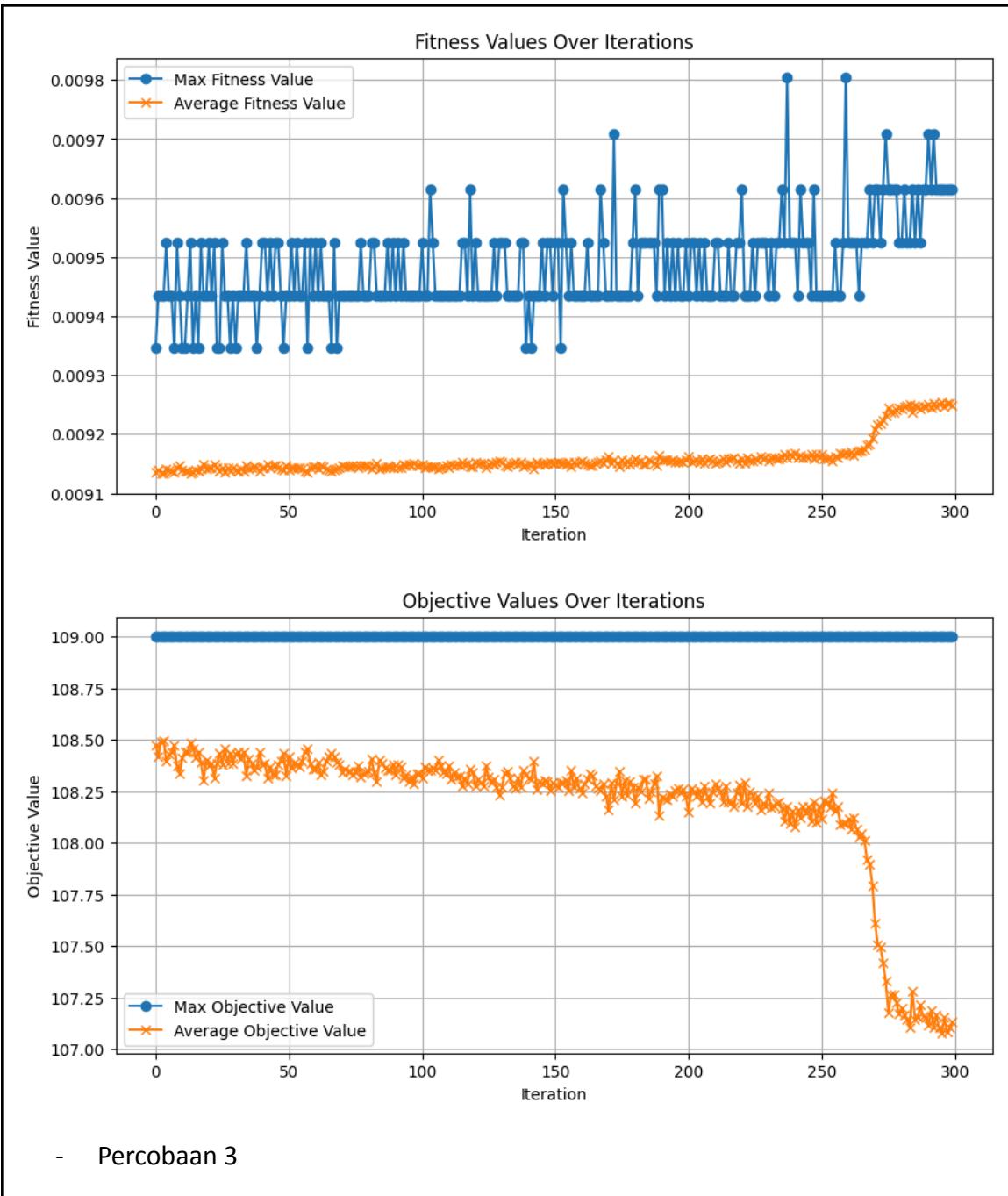


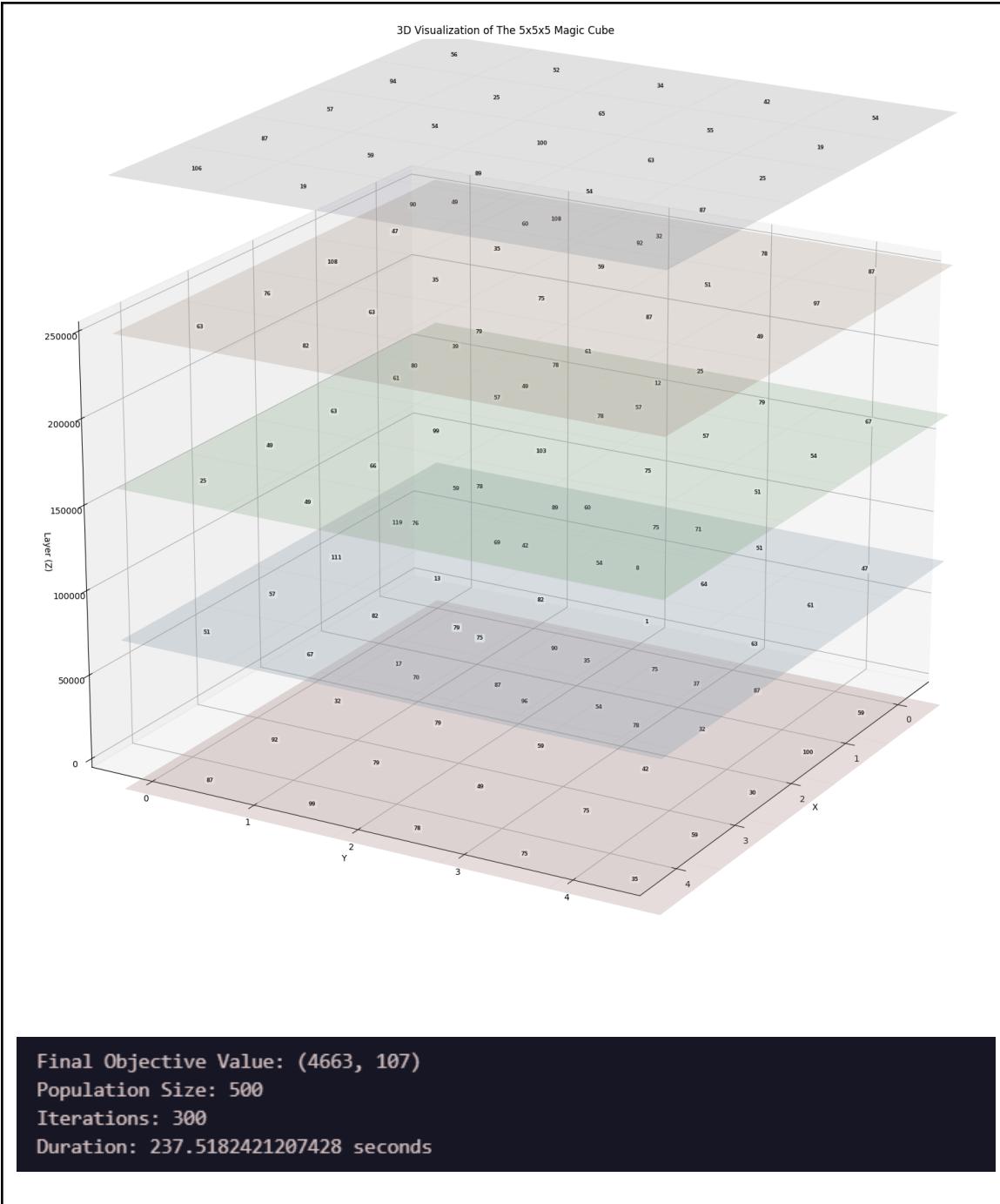


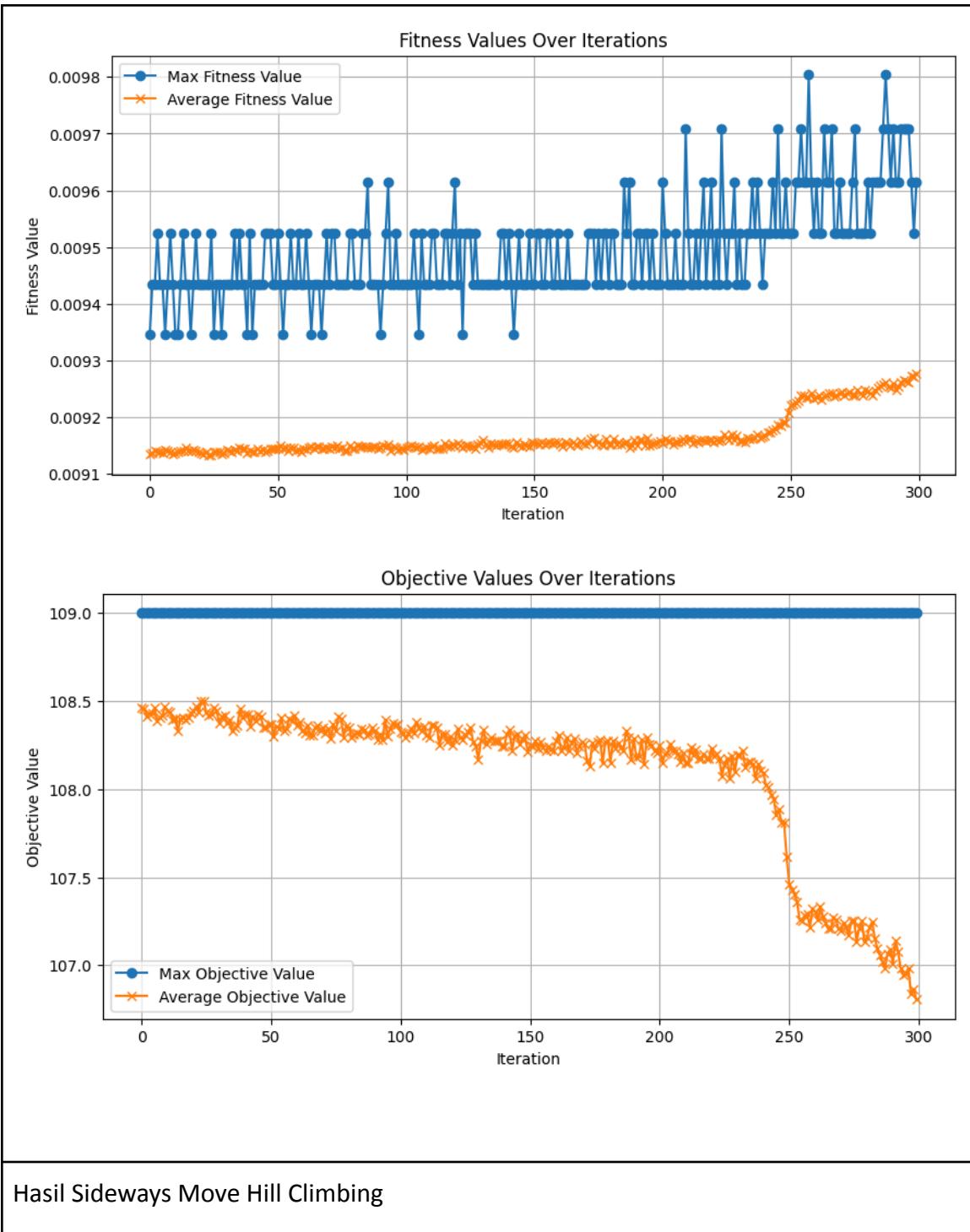


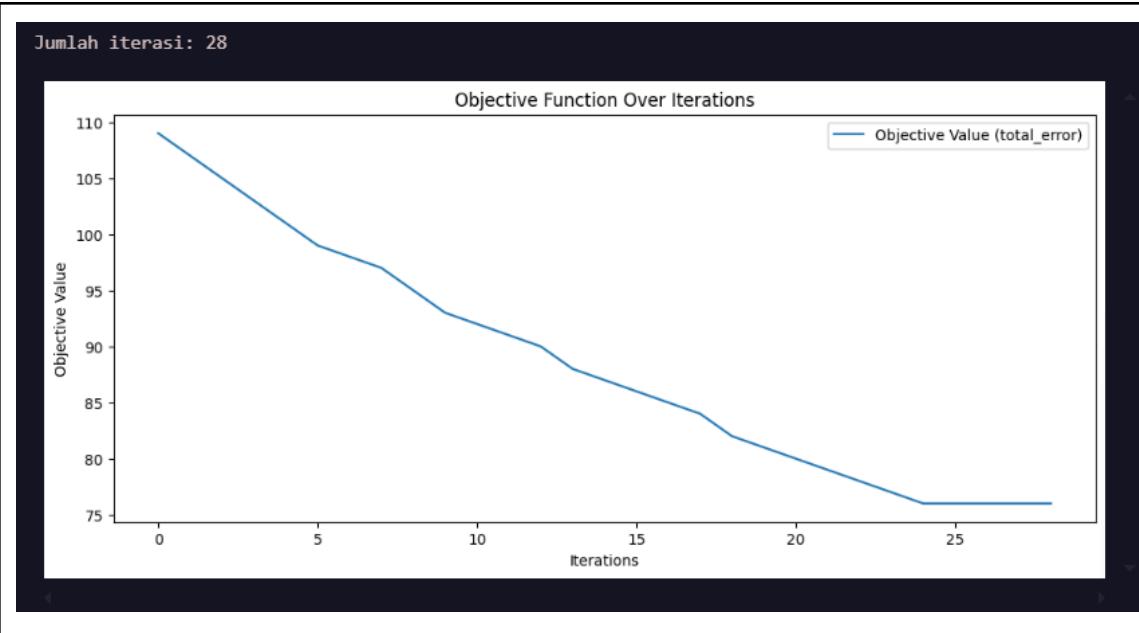


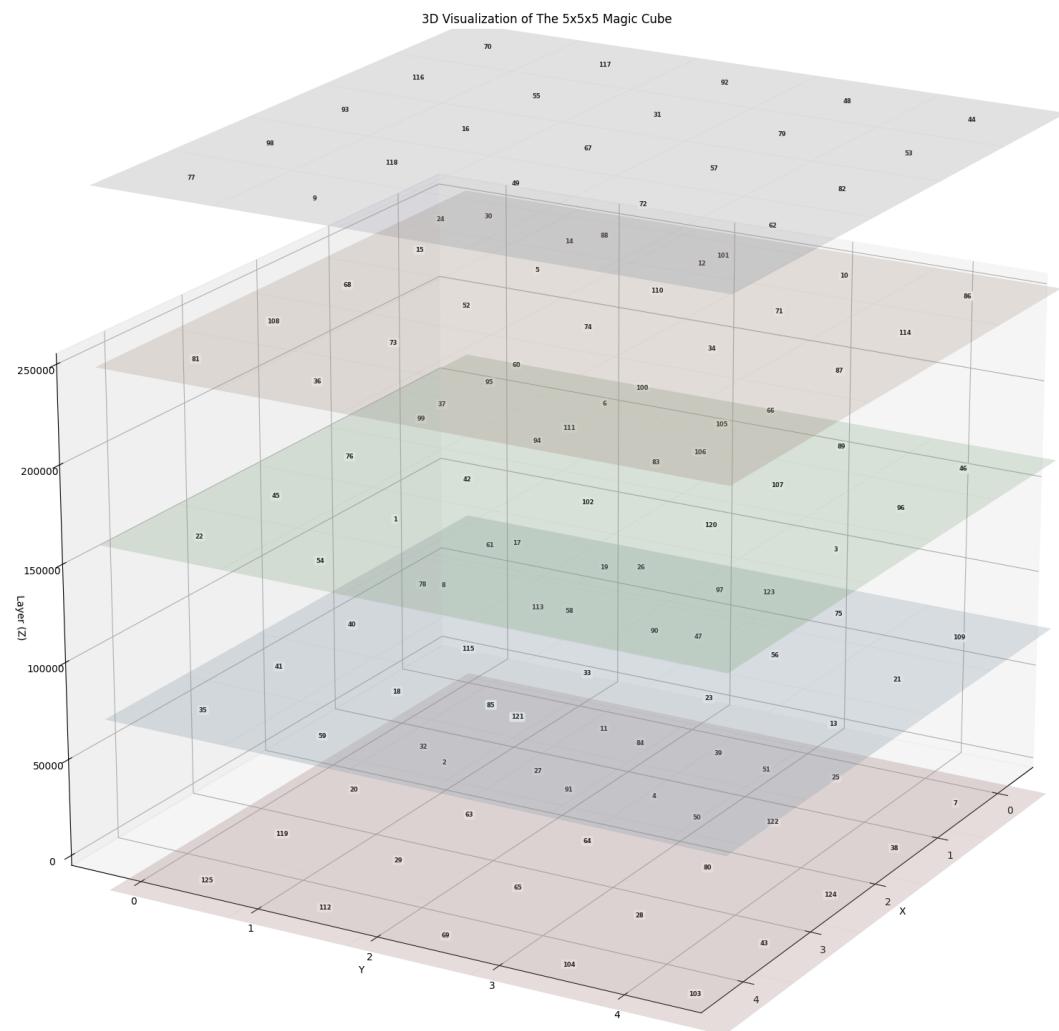
Final Objective Value: (3942, 107)
Population Size: 500
Iterations: 300
Duration: 273.8475058078766 seconds











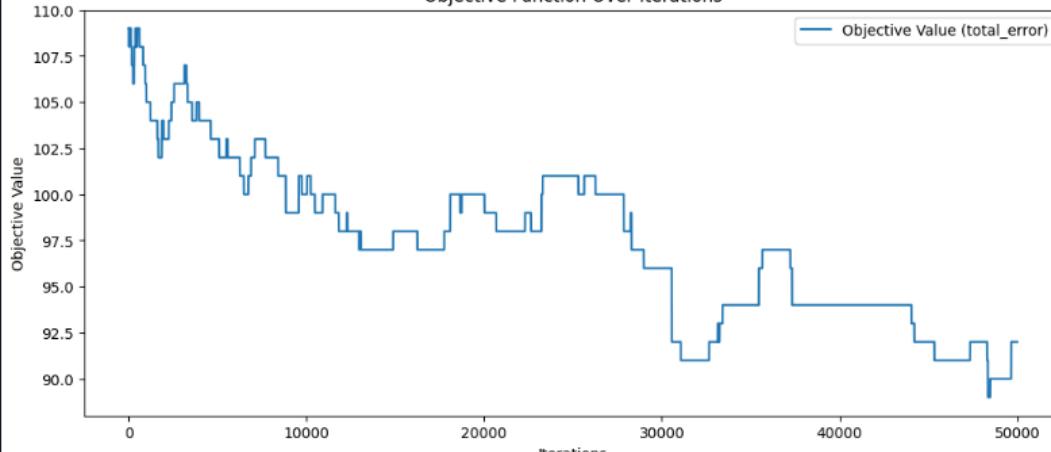
Best Cost: (4578, 76)
Duration: 315.11 seconds

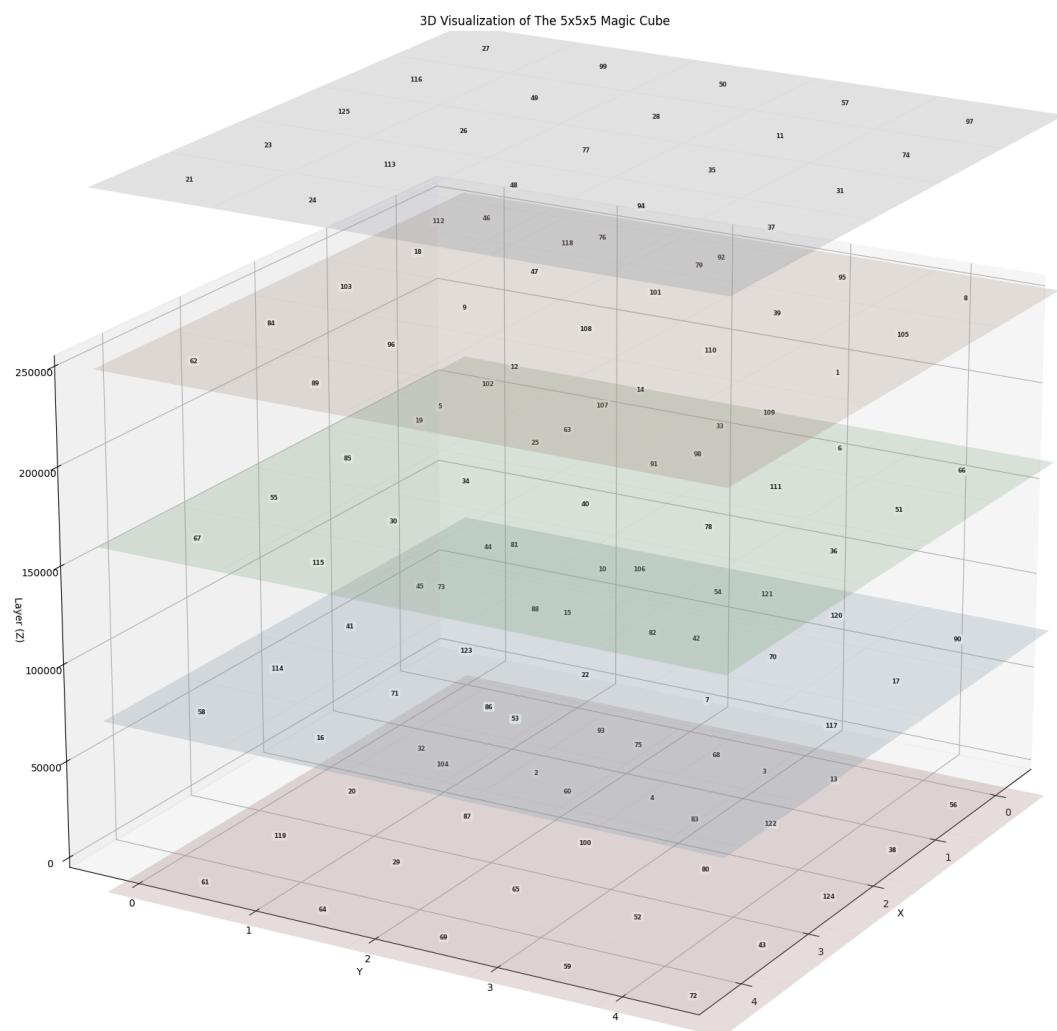
Hasil Stochastic Hill Climbing

Jumlah iterasi: 50000

Objective Function Over Iterations

— Objective Value (total_error)





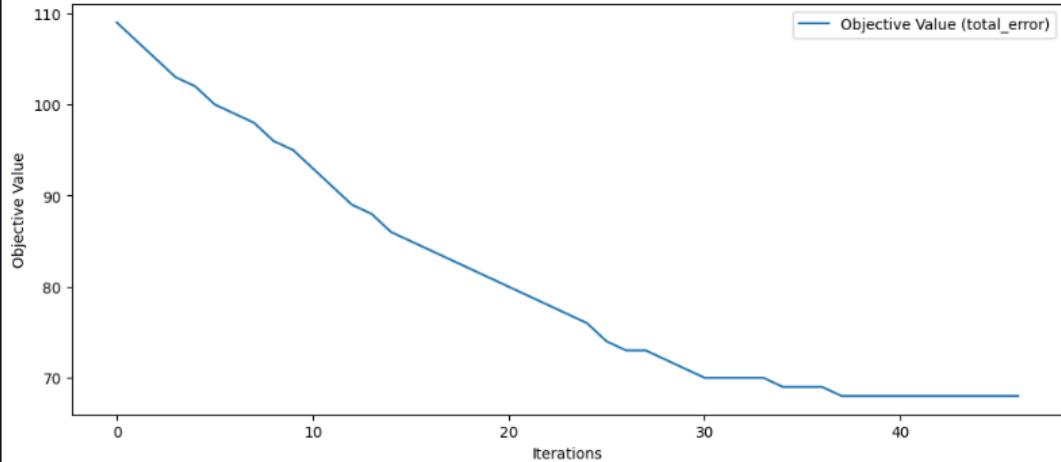
Best Cost: (985, 92)
Duration: 57.00 seconds

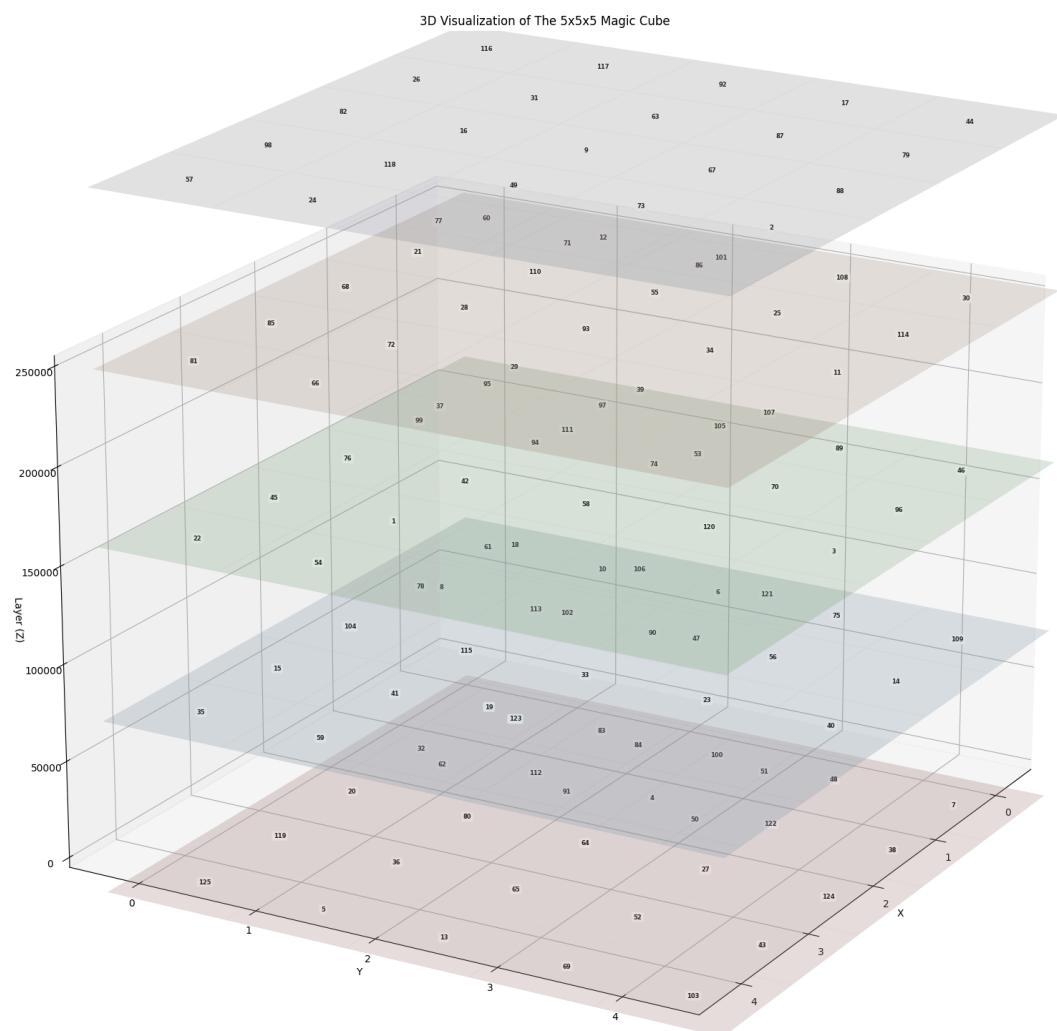
Hasil Steepest Ascent Hill Climbing

Jumlah iterasi: 46

Objective Function Over Iterations

Objective Value (total_error)

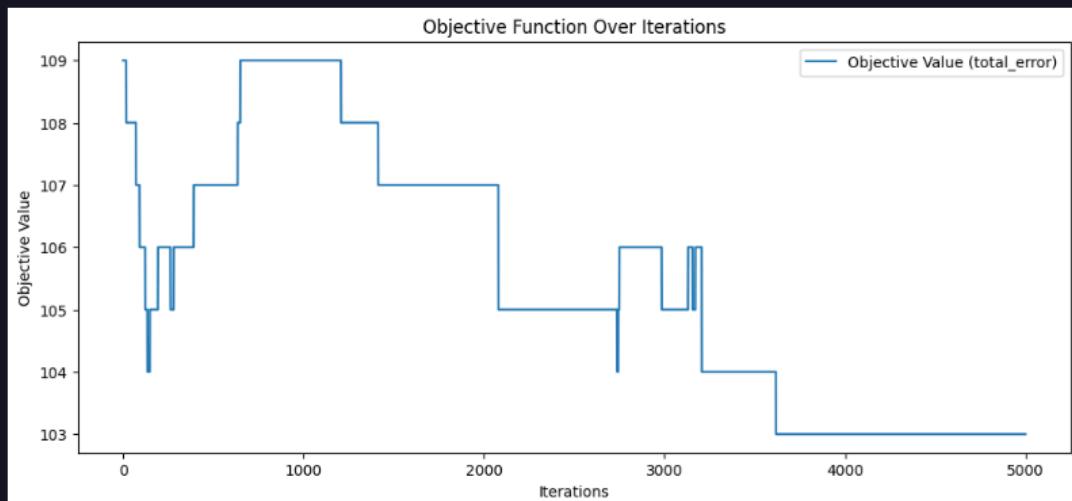




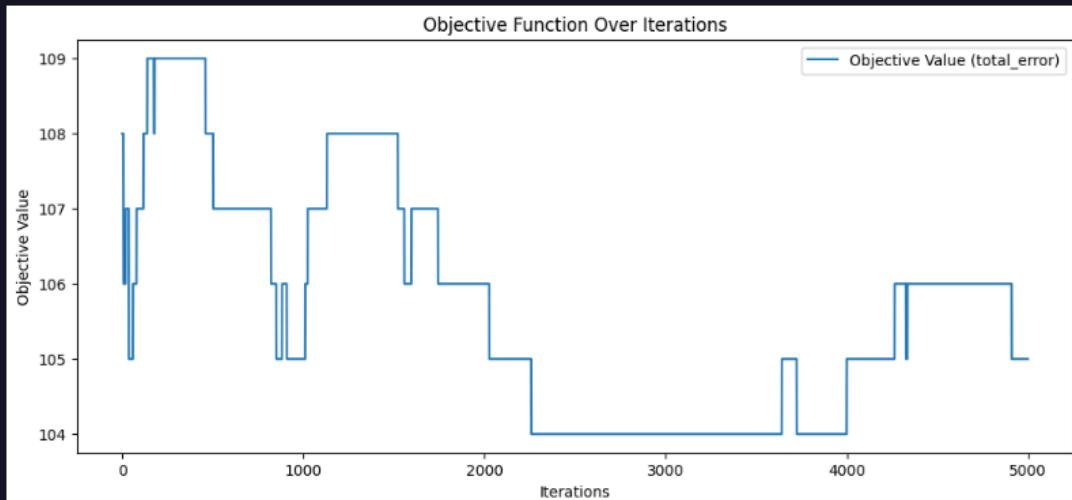
Best Cost: (2507, 68)
Duration: 456.13 seconds

Hasil Random Restart Hill Climbing

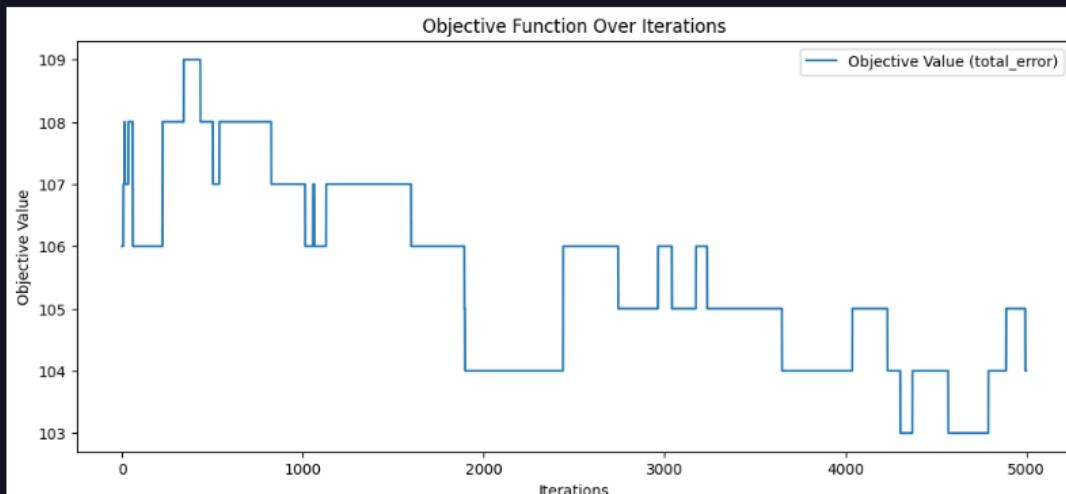
Restart 1:
Jumlah iterasi: 5000



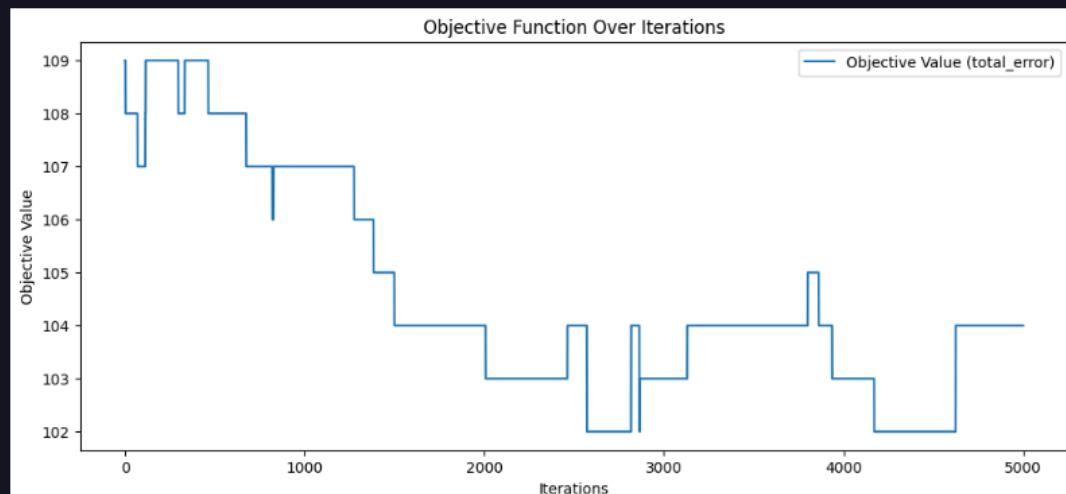
Restart 2:
Jumlah iterasi: 5000



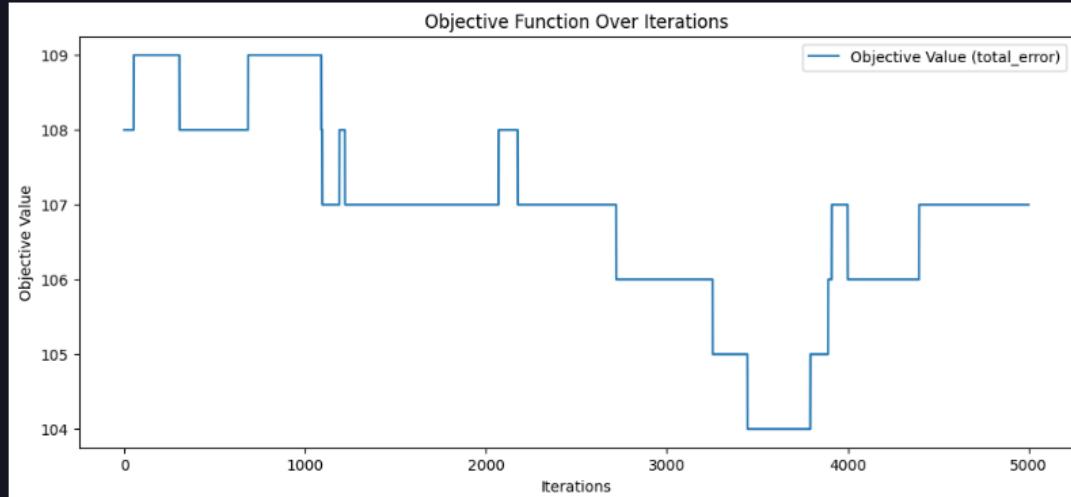
Restart 3:
Jumlah iterasi: 5000



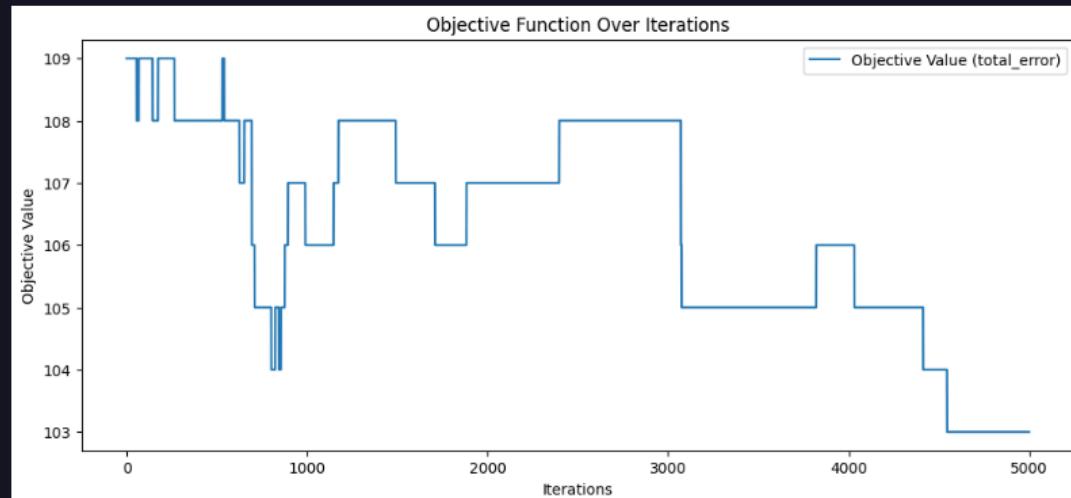
Restart 4:
Jumlah iterasi: 5000



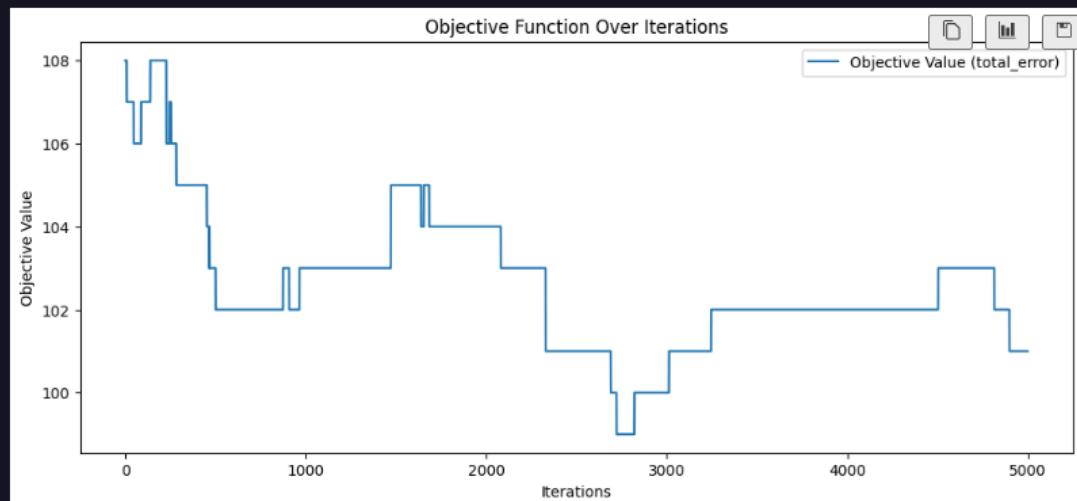
Restart 5:
Jumlah iterasi: 5000



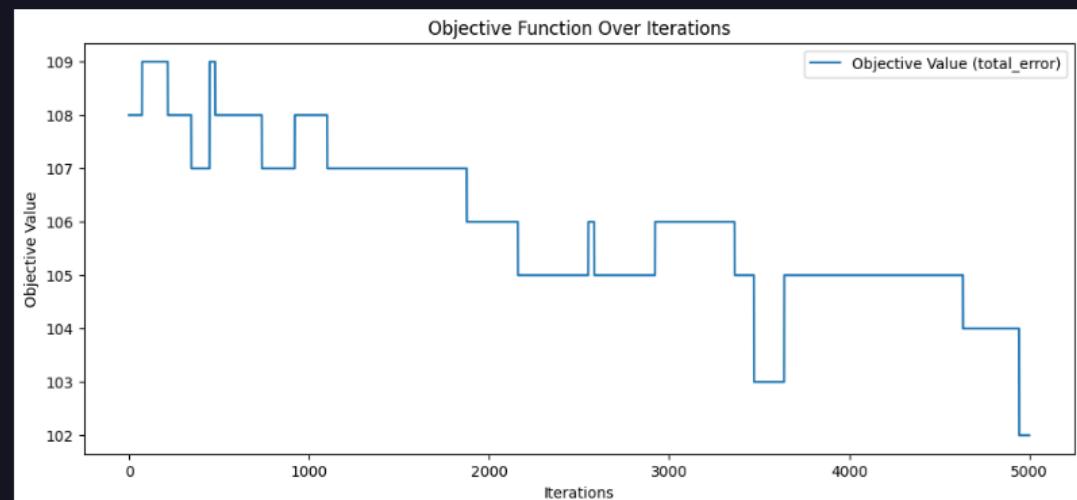
Restart 6:
Jumlah iterasi: 5000



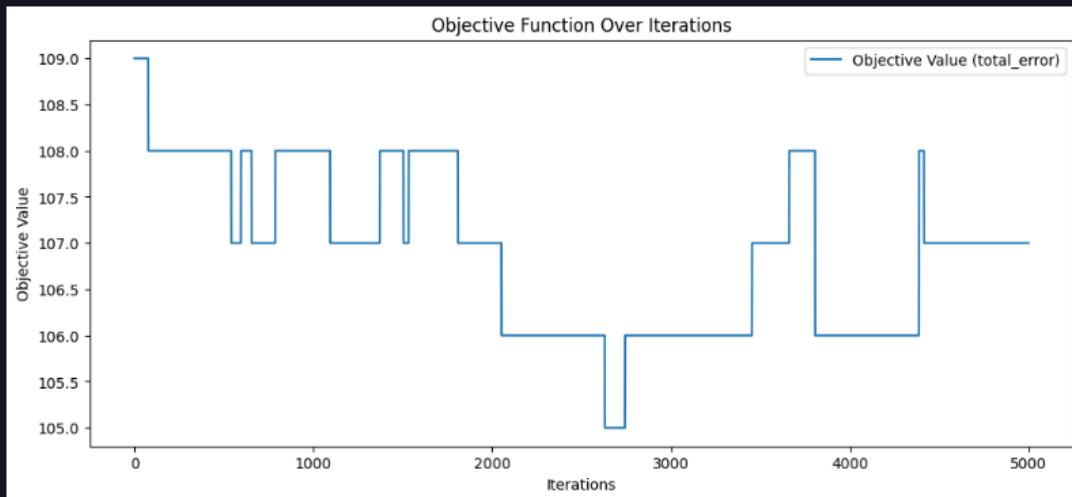
Restart 7:
Jumlah iterasi: 5000



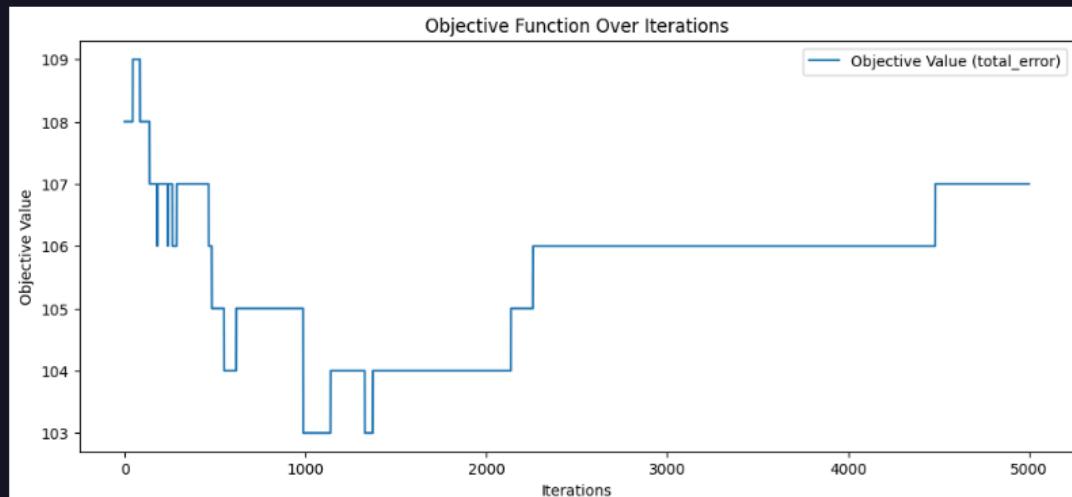
Restart 8:
Jumlah iterasi: 5000

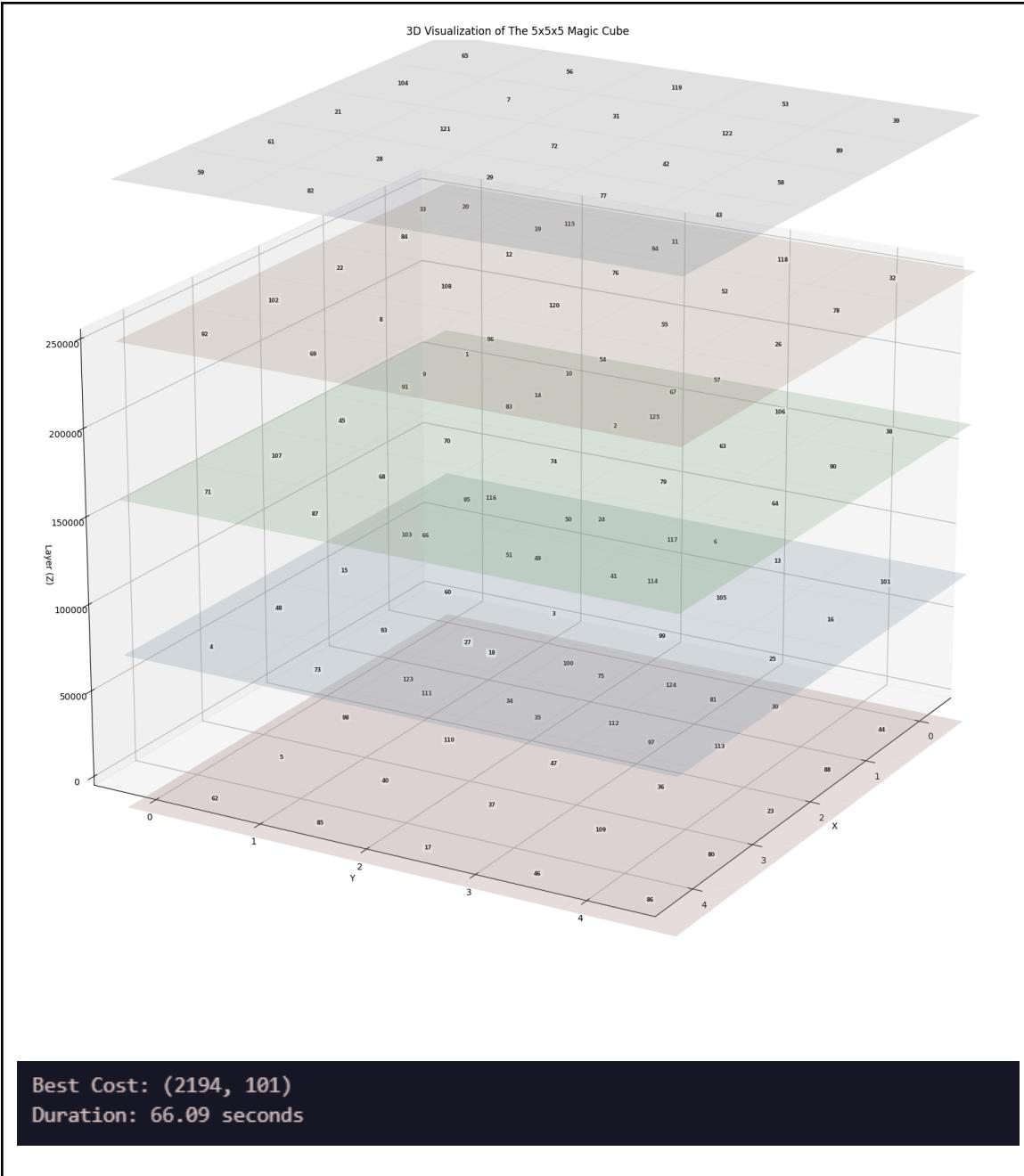


Restart 9:
Jumlah iterasi: 5000



Restart 10:
Jumlah iterasi: 5000





Analisis

Initial

Parameter	TC1	TC2	TC3
<i>Objective Value</i>	(6694, 109)	(7207, 108)	(6393, 109)

Simulated Annealing

Parameter	TC1	TC2	TC3
<i>Objective Value</i>	(6266, 108)	(7122, 106)	(6261, 107)
<i>Duration</i>	0.38s	1.13s	1.89s
<i>Stuck at</i>	2164	2176	2180

Genetic Algorithm

Populasi Sebagai Kontrol					
Variasi	Percobaan	Parameter	TC1	TC2	TC3
1	1	<i>Objective value</i>	(6610, 109)	(6456, 108)	(5829, 107)
		<i>Population size</i>	300		
		<i>Iteration</i>	100		
		<i>Duration</i>	14,44311s	35,30985s	48,148s
	2	<i>Objective value</i>	(6182, 108)	(5291, 109)	(5766, 108)
		<i>Population size</i>	300		
		<i>Iteration</i>	100		
		<i>Duration</i>	14,3221s	35,58704s	58,698s

	3	<i>Objective value</i>	(5434, 104)	(5493, 106)	(7366, 107)
		<i>Population size</i>	300		
		<i>Iteration</i>	100		
		<i>Duration</i>	14,5914s	35,28972s	56,721s
2	1	<i>Objective value</i>	(5391, 99)	(5506, 93)	(5791, 102)
		<i>Population size</i>	300		
		<i>Iteration</i>	200		
		<i>Duration</i>	28,966s	74,9968s	97,714s
	2	<i>Objective value</i>	(5567, 96)	(5749, 96)	(5787, 99)
		<i>Population size</i>	300		
		<i>Iteration</i>	200		
		<i>Duration</i>	29,3399s	72,34642s	101,901s
	3	<i>Objective value</i>	(6201, 93)	(5368, 96)	(5800, 96)
		<i>Population size</i>	300		
		<i>Iteration</i>	200		
		<i>Duration</i>	29,715s	71,38062s	91,766s
3	1	<i>Objective value</i>	(3732, 85)	(5090, 80)	(5214, 86)
		<i>Population size</i>	300		
		<i>Iteration</i>	500		
		<i>Duration</i>	73,6103s	182,0324s	243,345s
		<i>Objective value</i>	(5181, 87)	(4106, 79)	(4146, 78)

	2	<i>Population size</i>	300		
		<i>Iteration</i>	500		
		<i>Duration</i>	71,767s	179,81s	239,565s
		<i>Objective value</i>	(4074, 82)	(4702, 90)	(3807, 79)
	3	<i>Population size</i>	300		
		<i>Iteration</i>	500		
		<i>Duration</i>	72,164s	182,293s	261,372s
		Iterasi Sebagai Kontrol			
Variasi	Percobaan	Parameter	TC1	TC2	TC3
1	1	<i>Objective value</i>	(4304, 84)	(4882, 83)	(5756, 88)
		<i>Population size</i>	100		
		<i>Iteration</i>	300		
		<i>Duration</i>	14,386s	30,974s	49,393s
	2	<i>Objective value</i>	(4683, 87)	(4896, 85)	(5231, 88)
		<i>Population size</i>	100		
		<i>Iteration</i>	300		
		<i>Duration</i>	15,039s	31,694s	54,452s
	3	<i>Objective value</i>	(4890, 82)	(5613, 86)	(4832, 82)
		<i>Population size</i>	100		
		<i>Iteration</i>	300		
		<i>Duration</i>	14,93s	31,292s	50,869s

	1	<i>Objective value</i>	(3406, 84)	(4509, 83)	(5049, 85)
		<i>Population size</i>	200		
		<i>Iteration</i>	300		
		<i>Duration</i>	29,667s	72,583s	98,463s
2	2	<i>Objective value</i>	(4724, 84)	(4737, 87)	(5288, 86)
		<i>Population size</i>	200		
		<i>Iteration</i>	300		
		<i>Duration</i>	30,016s	63,627s	102,964s
3	3	<i>Objective value</i>	(5458, 91)	(4333, 86)	(4664, 88)
		<i>Population size</i>	200		
		<i>Iteration</i>	300		
		<i>Duration</i>	29,904s	74,756s	94,406s
3	1	<i>Objective value</i>	(4582, 108)	(4391, 109)	(3529, 106)
		<i>Population size</i>	500		
		<i>Iteration</i>	300		
		<i>Duration</i>	73,821s	169,582s	263,239s
	2	<i>Objective value</i>	(4542, 109)	(3508, 106)	(3942, 107)
		<i>Population size</i>	500		
		<i>Iteration</i>	300		
		<i>Duration</i>	73,623s	157,006s	273,848s
		<i>Objective value</i>	(4660, 108)	(4341, 108)	(4663, 107)

3	<i>Population size</i>	500		
	<i>Iteration</i>	300		
	<i>Duration</i>	74,501s	166,383s	237,518s

Sideways Move

Parameter	TC1	TC2	TC3
Iterasi	26	29	28
<i>Best cost</i>	(4945, 74)	(4246, 72)	(4578, 76)
<i>Duration</i>	70,78s	186,97s	315,11s

Stochastic

Parameter	TC1	TC2	TC3
Iterasi	50000	50000	50000
<i>Best Cost</i>	(1919, 95)	(2172, 89)	(985, 92)
<i>Duration</i>	16,22s	37,67s	57,00s

Steepest Ascent

Parameter	TC1	TC2	TC3
Iterasi	35	42	46
<i>Best Cost</i>	(4290, 68)	(3410, 67)	(2507, 68)
<i>Duration</i>	96,17s	254,54s	456,13s

Random Restart

Parameter	TC1	TC2	TC3
Jumlah <i>restart</i>	10	10	10
Iterasi	5000/restart	5000/restart	5000/restart
<i>Best cost</i>	(2357, 100)	(2228, 99)	(2194, 101)
<i>Duration</i>	16,69s	40,30s	66,09s

Urutan **objective value** pada setiap test case untuk seluruh algoritma dari yang paling baik

TC1: SA(4290, 68) > SW(4945, 74) > GI2(3406, 84) > CH(1919, 95) > RR(2357, 100) > SIM(6266, 108)

TC2: SA(3410, 67) > SW(4246, 72) > GP3(4106, 79) > CH(2172, 89) > RR(2228, 99) > SIM(7122, 106)

TC3: SA(2507, 68) > SW(4578, 76) > GI1(4832, 82) > CH(985, 92) > RR(2194, 101) > SIM(6261, 107)

Keterangan :

- CH : Stochastic Hill Climbing
- RR : Random Restart Hill Climbing
- GP : Genetic Algorithm Population Based
- GI : Genetic Algorithm Iteration Based
- SA : Steepest Ascent Hill Climbing
- SW : Hill Climbing with Sideways Move
- SIM : Simulated Annealing

1. Seberapa dekat tiap-tiap algoritma bisa mendekati global optima dan mengapa hasilnya demikian?

- Dari hasil eksperimen, terlihat bahwa algoritma Steepest Ascent Hill Climbing dan Hill Climbing with Sideways Move mampu mencapai nilai objektif terendah pada beberapa *test case*, yaitu sekitar 68 untuk Steepest Ascent Hill Climbing dan 74 untuk Hill Climbing with Sideways Move. Kedua algoritma tersebut mampu memiliki nilai objektif tertinggi karena mengeksplorasi semua *neighbor* dan memilih yang terbaik. Algoritma Hill Climbing lainnya seperti Stochastic Hill Climbing dan Random Restart Hill Climbing tidak memiliki *objective value* seperti SA dan SW karena jumlah *neighbor* untuk persoalan ini sangat banyak (sekitar 5^6 pilihan) sehingga kemungkinan memilih *neighbor* yang baik kecil.
- *Genetic Algorithm* menunjukkan hasil yang nilainya berada di tengah algoritma lain untuk populasi besar dan iterasi tinggi, dengan hasil terbaik 79 pada TC2 dengan 500 iterasi dan populasi sebesar 300. Algoritma ini lebih baik dalam menjaga variasi solusi melalui seleksi, crossover, dan mutasi, namun membutuhkan lebih banyak iterasi dan ukuran populasi untuk mencapai hasil optimal.
- *Simulated Annealing*, meskipun secara teori diharapkan bisa lebih unggul karena memiliki probabilitas penerimaan solusi lebih buruk yang menurun bertahap, menunjukkan hasil suboptimal dalam eksperimen ini. Hal ini kemungkinan disebabkan oleh pengaturan parameter suhu yang mungkin tidak ideal, sehingga algoritma ini terjebak di sekitar solusi suboptimal.

2. Bagaimana perbandingan hasil pencarian tiap-tiap algoritma dengan algoritma local search yang lain?

- *Steepest Ascent Hill Climbing* dan *Hill Climbing with Sideways Move* menunjukkan performa yang lebih baik dibandingkan algoritma local search lainnya dalam hal *objective value*. Kedua algoritma ini cenderung lebih efisien dalam mencapai solusi berkualitas tinggi karena metode pergerakan mereka. *Steepest Ascent* memilih tetangga terbaik dalam setiap iterasi, sementara *Sideways Move* memungkinkan pergerakan ke

solusi tetangga meskipun tidak memperbaiki nilai objective secara langsung, yang memberi fleksibilitas tambahan untuk keluar dari lokal optimum.

- *Stochastic Hill Climbing* dan *Random Restart Hill Climbing* memanfaatkan pencarian *neighbor* secara acak untuk eksplorasi lebih luas. *Stochastic* secara acak memilih tetangga baru, memberikan kesempatan yang lebih besar untuk keluar dari local optimum, meskipun tidak seefektif *Steepest Ascent* dan *Sideways Move* dalam beberapa kasus. *Random Restart*, dengan strategi restart, mengatasi masalah *local optimum* dengan memulai pencarian dari titik awal baru setelah mengalami kebuntuan, meningkatkan kemungkinan mencapai hasil yang lebih optimal. Hasil kedua algoritma tersebut cenderung sedikit di bawah *Steepest Ascent* dan *Sideways Move* dalam pengujian ini.
 - *Genetic Algorithm* kompetitif dengan *Random Restart* dan *Stochastic Hill Climbing* terutama pada populasi besar dan iterasi tinggi, di mana variasi yang lebih banyak membantu algoritma ini mendekati global optimum. Namun, *Genetic Algorithm* membutuhkan durasi lebih lama.
 - *Simulated Annealing*, meski efektif pada beberapa kasus, kalah dari *Random Restart* dan *Stochastic Hill Climbing* dalam mencapai global optimum. Pada percobaan ini, hal tersebut terjadi karena jumlah iterasi untuk *simulated annealing* kurang banyak untuk mencapai solusi optimal
3. **Bagaimana perbandingan durasi proses pencarian tiap algoritma relatif terhadap algoritma lainnya?**
- *Simulated Annealing* menunjukkan durasi yang paling singkat dengan rata-rata sekitar 1-2 detik, tetapi tidak menghasilkan solusi terbaik.
 - *Stochastic Hill Climbing* memiliki durasi sedang dengan rata-rata sekitar 16-57 detik, namun menghasilkan nilai objektif yang lebih baik.
 - *Steepest Ascent Hill Climbing* membutuhkan waktu cukup lama dengan rata-rata durasi 96-456 detik karena menelusuri setiap *neighbor* yang ada.

- *Random Restart Hill Climbing* memiliki durasi yang efisien sekitar 16-66 detik karena merupakan *Stochastic Hill Climbing* yang dilakukan secara berulang (yang lebih cepat daripada *Steepest Ascent Hill Climbing*).
- *Genetic Algorithm* menunjukkan peningkatan durasi signifikan seiring peningkatan iterasi dan ukuran populasi. Durasi mencapai 30-102 detik untuk iterasi kecil dan 73-263 detik untuk iterasi besar.
- *Hill Climbing with Sideways Move* membutuhkan durasi sedang sekitar 70-315 detik, algoritma ini memakan waktu yang cukup lama karena seperti *Steepest Ascent Hill Climbing*, algoritma ini menelusuri setiap *neighbor* yang ada.

4. Seberapa konsisten hasil akhir yang didapatkan dari tiap-tiap eksperimen yang dilakukan?

- *Steepest Ascent Hill Climbing* konsisten menghasilkan hasil yang optimal pada ketiga test case.
- *Hill Climbing with Sideways Move* juga memberikan hasil yang baik dan relatif konsisten di setiap test case.
- *Genetic Algorithm* cukup konsisten tetapi sangat bergantung pada populasi dan iterasi. Dengan populasi dan iterasi lebih tinggi, hasilnya lebih mendekati global optima dan variasi antar percobaan cenderung menurun.
- *Stochastic Hill Climbing* dan *Random Restart Hill Climbing* menunjukkan hasil yang bervariasi namun tetap baik pada setiap percobaan karena randomisasi.
- *Simulated Annealing* kurang konsisten dibandingkan algoritma berbasis randomisasi lainnya. Hal ini mungkin karena sensitivitas terhadap pengaturan suhu dan cooling rate, sehingga pada parameter tertentu, algoritma ini bisa berhenti di solusi yang jauh dari global optimum.

5. Bagaimana pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada Genetic Algorithm?

- Pada *Genetic Algorithm*, iterasi lebih banyak dan populasi lebih besar memberikan hasil lebih optimal. Hasil eksperimen menunjukkan bahwa semakin tinggi iterasi (500 iterasi) dan populasi (300), semakin rendah nilai objektif yang dihasilkan. Hal ini menunjukkan

bahwa variasi solusi pada setiap generasi membantu algoritma menghindari lokal optimum dan mendekati global optimum.

- Ukuran populasi yang besar memungkinkan algoritma mengeksplorasi berbagai konfigurasi solusi di awal pencarian, meningkatkan peluang untuk menemukan solusi yang lebih optimal seiring berjalannya iterasi. Namun, kekurangannya adalah semakin meningkat iterasi dan populasinya maka durasinya akan lebih lama juga, misalnya hingga 263 detik untuk TC3 pada iterasi 500.

6. **Mengapa *Simulated Annealing* tidak menjadi yang paling efektif, meskipun secara teori lebih baik?**

- Dalam eksperimen ini, *Simulated Annealing* mungkin tidak optimal karena ketergantungan pada parameter suhu awal, *cooling rate*, dan *stopping temperature* yang mempengaruhi durasi pencarian dan efektivitas eksplorasi. Jika suhu awal terlalu rendah atau cooling rate terlalu cepat, algoritma akan berhenti terlalu dini sebelum mencapai global optimum. Di sisi lain, pendinginan lambat bisa membuat algoritma mengeksplorasi solusi lebih lama pada biaya komputasi yang tinggi tanpa penurunan nilai objektif yang signifikan.

7. **Mengapa efek randomisasi pada *Stochastic Hill Climbing* dan *Random Restart* kurang membantu mencapai nilai objektif terendah secara *error count*, tetapi terendah secara *error sum*?**

- Pada implementasi yang telah dibuat, seperti yang dijelaskan pada bagian *objective function*, *objective function* diukur dengan dua parameter, yaitu jumlah selisih semua bagian kubus dengan *magic number (sum)* dan jumlah bagian kubus yang tidak sesuai dengan *magic number (count)*. *Objective function* akan meminimalkan *count* terlebih dahulu dan meminimalkan *sum* ketika *count* tidak bisa diminimalkan lagi.
- Karena *Stochastic Hill Climbing* mengambil *neighbor* secara acak, kemungkinan algoritma tersebut memilih *neighbor* dengan *count* lebih kecil sangat rendah (karena rentang nilai untuk parameter ini 1-100 an), namun kemungkinan algoritma tersebut memilih *neighbor* dengan *sum* lebih kecil masih relatif tinggi (karena rentang nilai untuk

parameter ini 1-10000an). Oleh karena itu, *neighbor* yang sering dipilih adalah *neighbor* dengan *count* yang sama, namun *sum* lebih rendah.

III. KESIMPULAN DAN SARAN

1. Kesimpulan

Dari hasil eksperimen tersebut, dapat disimpulkan bahwa *Steepest Ascent Hill Climbing* dan *Hill Climbing with Sideways Move* adalah algoritma yang paling efektif dalam mencapai nilai objektif rendah di sebagian besar test case. *Steepest Ascent* mencapai kinerja optimalnya dengan memilih tetangga terbaik di setiap iterasi, sementara *Hill Climbing with Sideways Move* memungkinkan perpindahan ke tetangga dengan nilai yang sama untuk menghindari jebakan local optimum. Kedua algoritma ini secara konsisten mendekati solusi berkualitas tinggi dalam setiap test case dibandingkan algoritma local search lainnya.

Stochastic Hill Climbing dan *Random Restart Hill Climbing*, yang mengandalkan eksplorasi randomisasi, juga menunjukkan kinerja cukup baik dalam menjangkau solusi global meskipun hasilnya sedikit di bawah *Steepest Ascent* dan *Sideways Move*. Kedua algoritma tersebut justru lebih unggul dalam meminimalkan jumlah selisih antara bagian kubus dengan *magic number*. Randomisasi memberi fleksibilitas tinggi untuk mengeksplorasi ruang solusi yang kompleks, sementara *Random Restart* secara efektif mengatasi local optimum dengan restart yang meningkatkan peluang menemukan solusi yang lebih baik.

Pada sisi lain, *Genetic Algorithm* berada di posisi menengah dan menunjukkan performa yang baik jika populasi besar dan iterasi tinggi, namun membutuhkan waktu komputasi lebih lama. *Simulated Annealing* memiliki hasil yang cenderung lebih suboptimal karena sensitivitasnya terhadap pengaturan parameter suhu dan *cooling rate*. Meskipun secara teori mampu menghindari *local optimum* melalui penerimaan solusi yang menurun bertahap, pengaturan yang kurang tepat dapat membuat algoritma ini berhenti sebelum mencapai global optimum.

2. Saran

Berdasarkan hasil analisis ini, beberapa saran yang dapat diberikan untuk pengembangan lebih lanjut adalah sebagai berikut:

1. Optimasi Parameter Algoritma Simulated Annealing

Disarankan untuk melakukan pengujian lebih lanjut pada parameter *Simulated Annealing*, seperti suhu awal dan laju pendinginan. Pengaturan yang tepat dapat meningkatkan kemampuan algoritma ini dalam mengeksplorasi solusi lebih baik dan menghindari jebakan local optimum.

2. Eksperimen dengan Dataset Beragam

Melakukan uji coba pada dataset yang lebih beragam dan kompleks dapat memberikan wawasan tambahan tentang efektivitas dan performa tiap algoritma dalam skenario berbeda. Hal ini juga membantu mengidentifikasi pengaturan optimal yang sesuai dengan variasi kompleksitas data.

3. Kombinasi Algoritma untuk Pendekatan Hybrid

Mengombinasikan algoritma dapat meningkatkan efisiensi pencarian solusi. Sebagai contoh, *Genetic Algorithm* dapat digunakan untuk mengeksplorasi solusi awal, kemudian diikuti oleh *Hill Climbing with Sideways Move* untuk menyempurnakan solusi tersebut. Pendekatan hybrid ini diharapkan mampu mencapai solusi optimal dengan lebih cepat.

4. Peningkatan Efisiensi Waktu Eksekusi Genetic Algorithm

Mengingat durasi yang lebih lama dalam eksekusi *Genetic Algorithm*, disarankan untuk fokus pada teknik pemrograman yang lebih efisien atau menerapkan pemrosesan paralel. Peningkatan ini akan mempercepat waktu eksekusi, terutama pada iterasi dan populasi besar.

5. Analisis Konsistensi Hasil Algoritma

Melakukan analisis mendalam terhadap konsistensi hasil yang dicapai tiap algoritma dapat membantu mengidentifikasi faktor yang mempengaruhi variasi hasil. Dengan memahami faktor ini, dapat dikembangkan strategi untuk meningkatkan konsistensi dalam pencarian solusi, terutama pada algoritma yang

menggunakan randomisasi, seperti *Stochastic Hill Climbing* dan *Random Restart Hill Climbing*.

IV. PEMBAGIAN TUGAS

Berikut adalah tabel rincian mengenai pendistribusian kerja oleh anggota kelompok:

No	NIM Anggota	Nama Anggota	Deskripsi
1	13522063	Shazya Audrea Taufik	Algoritma Simulated Annealing, Laporan
2	13522070	Marzuli Suhada M	Algoritma Genetic, Laporan
3	13522085	Zahira Dina Amalia	Algoritma Hill-Climbing, Laporan
4	13522108	Muhammad Neo Cicero Koda	Algoritma Hill-Climbing, Laporan

V. REFERENSI

- [Features of the magic cube - Magisch vierkant](#)
- [Perfect Magic Cubes \(trump.de\)](#)
- [Magic cube - Wikipedia](#)
- [Handbook of Evolutionary Computation: May 97 release](#)