

Tugas Besar 1 IF3170 Intelelegensi Artifisial

Semester I tahun 2024/2025

### Pencarian Solusi Diagonal Magic Cube dengan Local Search



Disusun oleh:

M. Zaidan Sa'dun Robbani 13522135

Farhan Raditya Aji 13522142

Rafif Ardhinto Ichwantoro 13522159

Rayhan Ridhar Rahman 13522160

**PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
2024/2025**

## DAFTAR ISI

DAFTAR ISI.....	2
BAB 1	
DESKRIPSI PERSOALAN.....	4
BAB 2	
PEMBAHASAN.....	6
2.1 Pemilihan Objective Function.....	6
2.2 Penjelasan Implementasi Algoritma Local Search.....	7
2.2.1 Hill Climb.....	7
2.2.2 Simulated Annealing.....	7
2.2.3 Genetic Algorithm.....	9
2.3 Hasil Eksperimen dan Analisis.....	14
2.3.1 Hill Climb.....	14
2.3.2 Simulated Annealing.....	14
2.3.3 Genetic Algorithm.....	14
BAB 3	
KESIMPULAN.....	15
LAMPIRAN.....	16
DAFTAR PUSTAKA.....	17

## **BAB 1**

### **DESKRIPSI PERSOALAN**

Tugas Besar I dalam kuliah IF3170 Intelelegensi Buatan bertujuan untuk memberi wawasan kepada peserta mengenai penerapan algoritma local search dalam mencari solusi untuk permasalahan Diagonal Magic Cube. Dalam konteks ini, Diagonal Magic Cube merupakan struktur kubus berukuran  $n \times n \times n$  yang terdiri dari angka 1 hingga  $n^3$  tanpa pengulangan. Setiap baris, kolom, tiang, dan diagonal dari kubus harus menghasilkan jumlah yang sama, yang disebut sebagai magic number. Untuk tugas ini, peserta diminta menyusun rencana implementasi algoritma local search khusus untuk kubus berukuran 5x5x5, dimulai dari konfigurasi acak angka-angka tersebut.

Dalam mendefinisikan objective function, kita akan mengukur seberapa dekat konfigurasi kubus dengan kondisi ideal yang ditentukan oleh magic number. Objective function ini dapat berupa selisih antara jumlah yang dihasilkan oleh setiap baris, kolom, tiang, dan diagonal dengan magic number. Semakin kecil nilai dari objective function, semakin baik konfigurasi tersebut. Proses pencarian solusi melibatkan beberapa algoritma local search, seperti Steepest Ascent Hill-climbing, Hill-climbing with Sideways Move, Random Restart Hill-climbing, Stochastic Hill-climbing, Simulated Annealing, dan Genetic Algorithm.

Di antara semua algoritma tersebut, Simulated Annealing dipandang sebagai pendekatan terbaik untuk menyelesaikan permasalahan ini. Simulated Annealing memiliki mekanisme unik yang memungkinkan penerimaan solusi yang kurang optimal dalam tahap awal pencarian untuk menghindari terjebak pada local optima. Dengan memanfaatkan prinsip penurunan suhu secara bertahap, algoritma ini mampu mengeksplorasi ruang pencarian secara lebih efektif, memberikan peluang untuk menemukan solusi yang mendekati optimal dalam struktur pencarian yang kompleks seperti Diagonal Magic Cube.

Simulated Annealing mengungguli metode lain dalam konteks ini karena fleksibilitasnya dalam menyeimbangkan eksplorasi dan eksloitasi ruang pencarian. Pendekatan ini

memungkinkan pergerakan keluar dari local optim dengan mempertimbangkan solusi suboptimal dalam jangka pendek, yang pada akhirnya meningkatkan kemungkinan menemukan solusi global optimal. Ini menjadikannya lebih handal dibandingkan dengan metode seperti Steepest Ascent Hill-climbing atau Random Restart Hill-climbing, yang cenderung terjebak pada solusi suboptimal.

Rencana implementasi algoritma-algoritma local search ini mencakup kelas utama seperti `MagicCube` untuk mengatur status kubus dan metode `evaluate` untuk menghitung nilai objective function. Selain itu, terdapat fungsi spesifik untuk setiap algoritma, seperti `steepest\_ascent`, `hill\_climbing\_with\_sideways\_move`, `random\_restart\_hill\_climbing`, `stochastic\_hill\_climbing`, `simulated\_annealing`, dan `genetic\_algorithm`. Setiap fungsi ini dirancang untuk menangani penukaran angka, evaluasi solusi, dan penerapan mekanisme pencarian yang sesuai dengan algoritma yang digunakan.

## BAB 2

### PEMBAHASAN

#### 2.1 Pemilihan Objective Function

Objective function yang digunakan dalam tugas ini adalah fungsi cost yang berusaha meminimalkan perbedaan antara jumlah dari setiap baris, kolom, tiang, dan diagonal kubus dengan nilai magic number yang diharapkan.

$$\text{Cost Function} = \sum_{i=1}^m |Sum_i - MagicNumber|$$

Keterangan:

1.  $Sum_i$  adalah jumlah dari elemen-elemen pada baris, kolom, tiang, diagonal, atau triagonal ke-i dalam kubus.
2.  $MagicNumber$  adalah jumlah yang diharapkan pada tiap dimensi.
3.  $m$  adalah total jumlah baris, kolom, tiang, dan diagonal yang perlu dicek.

Objective function ini digunakan karena cost function memberikan informasi seberapa jauh solusi saat ini dari solusi yang diinginkan. Semakin kecil nilai cost function, semakin dekat kita dengan solusi optimal. Dalam konteks Diagonal Magic Cube, kita ingin menemukan susunan angka sedemikian rupa sehingga setiap dimensi (baris, kolom, tiang, dan diagonal) memiliki jumlah yang sama dengan magic number, sehingga fungsi ini membantu mengukur keberhasilan setiap iterasi algoritma.

Source code program Objective function sebagai berikut:

```
def __get_objective_value(self):  
    objectiveSum = (self.__rows_value() + self.__cols_value() +  
    self.__pillars_value() + self.__diagonals_value() + self.__triagonals_value())  
    return objectiveSum  
  
def __rows_value(self):
```

```

        rowsValue = np.sum(np.abs(self.magicNumber - np.sum(self.cube[:, :, :],
        :, axis = 2)))
        return rowsValue

    def __cols_value(self):
        colsValue = np.sum(np.abs(self.magicNumber - np.sum(self.cube[:, :, :],
        :, axis = 1)))
        return colsValue

    def __pillars_value(self):
        pillarsValue = np.sum(np.abs(self.magicNumber - np.sum(self.cube[:, :, :],
        :, axis = 0)))
        return pillarsValue

    def __diagonals_value(self):
        diagonalsValue = 0

        indices = np.arange(5)

        for i in indices:
            d1Sum = np.sum(self.cube[i, indices, indices])
            d2Sum = np.sum(self.cube[i, indices, 4 - indices])
            diagonalsValue += abs(self.magicNumber - d1Sum) +
abs(self.magicNumber - d2Sum)

            for j in indices:
                d1Sum = np.sum(self.cube[indices, j, indices])
                d2Sum = np.sum(self.cube[indices, j, 4 - indices])
                diagonalsValue += abs(self.magicNumber - d1Sum) +
abs(self.magicNumber - d2Sum)

            for k in indices:
                d1Sum = np.sum(self.cube[indices, indices, k])
                d2Sum = np.sum(self.cube[indices, 4 - indices, k])
                diagonalsValue += abs(self.magicNumber - d1Sum) +
abs(self.magicNumber - d2Sum)

        return diagonalsValue

    def __triagonals_value(self):
        indices = np.arange(5)

        t1Sum = np.sum(self.cube[indices, indices, indices])
        t2Sum = np.sum(self.cube[indices, indices, 4 - indices])
        t3Sum = np.sum(self.cube[4 - indices, indices, indices])
        t4Sum = np.sum(self.cube[4 - indices, indices, 4 - indices])

        triagonalsValue = (abs(self.magicNumber - t1Sum) +
abs(self.magicNumber - t2Sum) + abs(self.magicNumber - t3Sum) +
abs(self.magicNumber - t4Sum))

        return triagonalsValue

```

Berikut adalah penjelasan dari fungsi-fungsi di atas:

- `__get_objective_value(self)`: Fungsi ini merupakan fungsi utama yang menghitung total nilai objektif dari magic cube dengan menjumlahkan semua nilai yang dihitung dari fungsi-fungsi helper lainnya. Fungsi ini mengombinasikan nilai dari baris (rows), kolom (columns), pilar (pillars), diagonal, dan triagonal. Hasil penjumlahan ini menunjukkan seberapa jauh cube tersebut dari kondisi "magic" yang ideal - semakin kecil nilainya, semakin mendekati solusi magic cube yang diinginkan.
- `__rows_value(self)`: Fungsi ini menghitung nilai objektif untuk semua baris dalam cube. Menggunakan numpy, fungsi ini menjumlahkan nilai-nilai sepanjang axis 2 (kedalaman cube) untuk setiap baris, kemudian menghitung selisih absolut antara jumlah tersebut dengan magic number yang diinginkan. Semakin besar perbedaan antara jumlah baris dengan magic number, semakin besar nilai yang dihasilkan, mengindikasikan seberapa jauh baris-baris tersebut dari kondisi magic yang diinginkan.
- `__cols_value(self)`: Fungsi ini menghitung nilai objektif untuk semua kolom dalam cube. Mirip dengan fungsi `rows_value`, tetapi menjumlahkan nilai-nilai sepanjang axis 1 (tinggi cube) untuk setiap kolom. Fungsi ini juga menghitung selisih absolut antara jumlah setiap kolom dengan magic number, memberikan indikasi seberapa jauh kolom-kolom tersebut dari kondisi magic yang diinginkan.
- `__pillars_value(self)`: Fungsi ini menghitung nilai objektif untuk semua pilar (garis vertikal yang menembus cube dari atas ke bawah) dalam cube. Fungsi menjumlahkan nilai-nilai sepanjang axis 0 untuk setiap pilar, kemudian menghitung selisih absolut antara jumlah tersebut dengan magic number. Hasil perhitungan menunjukkan seberapa jauh pilar-pilar tersebut dari kondisi magic yang diinginkan.
- `__diagonals_value(self)`: Fungsi ini menghitung nilai objektif untuk semua diagonal dalam cube. Fungsi ini lebih kompleks karena menghitung tiga jenis diagonal: diagonal pada setiap face yang sejajar dengan sumbu x, y, dan z. Untuk setiap face, dihitung dua diagonal (dari kiri atas ke kanan bawah dan dari kanan atas ke kiri bawah). Fungsi menggunakan indeks untuk mengakses elemen-elemen diagonal dan menghitung selisih absolut antara jumlah setiap diagonal dengan magic number.
- `__triagonals_value(self)`: Fungsi ini menghitung nilai objektif untuk empat triagonal utama cube (diagonal yang melewati cube dari satu pojok ke pojok yang

berlawanan). Fungsi menghitung empat kemungkinan triagonal dalam cube 3D, yaitu dari pojok depan-atas-kiri ke belakang-bawah-kanan, dari depan-atas-kanan ke belakang-bawah-kiri, dari depan-bawah-kiri ke belakang-atas-kanan, dan dari depan-bawah-kanan ke belakang-atas-kiri. Seperti fungsi lainnya, fungsi ini juga menghitung selisih absolut antara jumlah setiap triagonal dengan magic number yang diinginkan.

## 2.2 Penjelasan Implementasi Algoritma Local Search

### 2.2.1 Hill Climbing

Algoritma *Hill-Climbing* adalah algoritma yang akan “mendaki” menuju state dengan nilai objektif terbaik (mendekati 0). Digunakan dua algoritma untuk mencari neighbor yaitu `find_best_neighbor()` dan `find_random_neighbor()`. Berikut merupakan kelas induknya:

```
class HillClimbing:  
    def __init__(self, initialState : Cube):  
        self.state = initialState  
        self.iteration = 0  
        self.objectiveValues = []  
  
    # Find best neighbor for steepest (Not stochastic)  
    def find_best_neighbor(self):  
        bestValue = float('inf')  
        bestNeighbor : Cube = None  
  
        for a in range(125):  
            i = a % 5  
            j = a // 5 % 5  
            k = a // 25  
  
            b = a + 1  
  
            while b != 125:  
                successor = self.state.cube.copy()  
  
                di = b % 5  
                dj = b // 5 % 5  
                dk = b // 25  
  
                successor[i,j,k], successor[di,dj,dk] = successor[di,dj,dk],  
successor[i,j,k]  
  
                newState = Cube(successor)  
  
                if newState.value < bestValue:  
                    bestNeighbor = newState  
                    bestValue = newState.value  
                elif newState.value == bestValue and random.randint(0,1):
```

```

        bestNeighbor = newState

        b += 1

    return bestNeighbor

    def find_random_neighbor(self):
        neighbor = self.state.cube.copy()
        x1, y1, z1 = random.randint(0, 4), random.randint(0, 4),
random.randint(0, 4)
        x2, y2, z2 = random.randint(0, 4), random.randint(0, 4),
random.randint(0, 4)
        neighbor[x1, y1, z1], neighbor[x2, y2, z2] = neighbor[x2, y2, z2],
neighbor[x1, y1, z1]

    return neighbor

    def search(self):
        raise NotImplementedError("Subclasses should implement this method")

```

- Fungsi `__init__()` : Melakukan inisialisasi atribut sebagai data yang akan disampaikan.
- Fungsi `find_best_neighbor()` : Mencari neighbor dengan nilai terbaik. Jika terdapat successor dengan nilai yang sama, ambil secara acak.
- Fungsi `find_random_neighbor()` : Mencari neighbor secara stochastic (acak)
- Fungsi `search()` : Dasar fungsi untuk kelas-kelas yang menjadi *children* dari kelas ini. Akan dipanggil untuk melakukan *local search* sesuai dengan metode hill-climbing subklasnya

#### A. Steepest Ascent

Steepest Ascent Hill-climbing merupakan algoritma yang selalu mencari neighbor terbaik untuk dijadikan current state selanjutnya. Jadi algoritma akan mencari pertukaran antara 2 potongan kubus yang akan menghasilkan value yang lebih baik dari current state.

```

class SteepestAscent(HillClimbing):
    def __init__(self, initial_state : Cube):
        super().__init__(initial_state)

    def search(self):
        success = True
        self.objectiveValues.append(self.state.value)

        while success:
            self.iteration += 1
            neighbor : Cube = self.find_best_neighbor()

            if self.state.value > neighbor.value:
                self.state = neighbor

```

```

        else:
            success = False

        self.objectiveValues.append(self.state.value)
    
```

## B. Steepest Ascent with Sideways Move

Variasi dari Steepest Ascent. Perbedaan dengan yang biasa adalah , algoritma ini membolehkan untuk pergi ke neighbor dengan state dengan nilai objektif yang sama dan pergerakan ini diberi batasan juga.

```

class SidewaysMovement(HillClimbing):
    def __init__(self, initial_state : Cube, maxSidewaysMoves = 4):
        super().__init__(initial_state)
        self.maxSidewaysMove = maxSidewaysMoves

    def search(self):
        success = True
        sidewaysMoves = 0
        self.objectiveValues.append(self.state.value)

        while success and sidewaysMoves < self.maxSidewaysMove:
            self.iteration += 1
            neighbor : Cube = self.find_best_neighbor()

            if self.state.value >= neighbor.value:
                if self.state.value == neighbor.value:
                    sidewaysMoves += 1
                else:
                    sidewaysMoves = 0
                    self.state = neighbor
            else:
                success = False

        self.objectiveValues.append(self.state.value)
    
```

- Terdapat atribut khusus maxSidewaysMoves sebagai jumlah pergerakan horizontal yang boleh berturut-turut

## C. Stochastic

Algoritma Hill-Climbing yang tidak mencari neighbor terbaik, tetapi neighbor yang acak. Dalam implementasi pada algoritma yang dibangun, terminasi algoritma ini terjadi ketika fungsi tidak dapat menemukan neighbor acak dengan nilai objektif yang lebih tinggi setelah n kali, pada algoritma diberi 100 kali.

```

class StochasticHC(HillClimbing):
    def __init__(self, initialState: Cube, nmax = 100):
        super().__init__(initialState)
        self.nmax = nmax
    
```

```

# Asumsi nmax adalah toleransi ketidaksesuaian nilai neighbor baru dari
state tertentu
def search(self):
    tolerance = self.nmax
    self.objectiveValues.append(self.state.value)

    while tolerance and self.state.value != 0:
        self.iteration += 1
        neighbor : Cube = self.find_random_neighbor()
        if self.state.value > neighbor.value:
            self.state = neighbor
            tolerance = self.nmax
        else:
            tolerance -= 1

    self.objectiveValues.append(self.state.value)

```

- Terdapat atribut khusus nmax, bukan merupakan parameter masukan dari aplikasi tetapi mengatur banyaknya kesalahan berturut-turut sebagai terminasi dari fungsi.

#### D. Random Restart

Variasi dari Steepest Ascent yang lainnya. Sama seperti Stochastic, ketika mencapai iterasi tertentu berdasarkan kemungkinan mencapai lokal maksimum (Dalam algoritma ini maksimal 30 iterasi). Jika masih belum sampai, pada algoritma yang dikembangkan, value terakhir disimpan dahulu, kemudian memperbarui state awal yang kemudian akan digunakan berikutnya, sampai mencapai nilai objektif 0 atau mencapai restart maksimum

```

class RandomRestart(HillClimbing):
    def __init__(self, initialState: Cube, maxRestart = 8):
        super().__init__(initialState)
        self.maxRestart = maxRestart
        self.iterationsPerRestart = []

    def search(self):
        bestOverall : Cube = None
        bestOverallValue = float('inf')
        self.objectiveValues.append(self.state.value)

        nmax = 50

        for i in range(self.maxRestart):
            if i > 0:
                self.state = Cube()

            currentIterations = 0

            success = True

            while success and currentIterations < nmax:
                currentIterations += 1
                neighbor : Cube = self.find_best_neighbor()

```

```

        if self.state.value > neighbor.value:
            self.state = neighbor
        else:
            success = False

        self.objectiveValues.append(self.state.value)

        self.iterationsPerRestart.append(currentIterations)

        if bestOverallValue > self.state.value:
            bestOverall = self.state
            bestOverallValue = self.state.value

        self.state = bestOverall
        self.iteration = sum(self.iterationsPerRestart)
    
```

- Terdapat dua atribut khusus yaitu maxRestart dan iterationsPerRestart. Sesuai dengan namanya, maxRestart adalah jumlah pengulangan maksimum sedangkan iterationsPerRestart merupakan list banyak iterasi yang dilakukan setiap pengulangan

### 2.2.2 Simulated Annealing

Simulated annealing adalah algoritma local search yang serupa dengan stochastic search tetapi dimungkinkan untuk mendapatkan neighbor dengan nilai fungsi objektif yang lebih buruk. Pada algoritma ini neighbor dipilih secara acak, Perpindahan ke solusi yang lebih buruk dapat dilakukan menggunakan fungsi  $e^{\Delta E/T}$  dengan merupakan perbedaan nilai objektif suksesor dengan solusi sekarang dan T merupakan hasil dari fungsi Schedule. Kemudian nilai yang didapat dari fungsi ini akan dibandingkan dengan nilai float konstan yang wajar atau nilai float acak (Dalam jangkauan 0 hingga 1). Jika hasil fungsi lebih besar, maka neighbor akan menjadi solusi.

Struktur algoritma dan source code Program, terdiri dari beberapa fungsi utama, yaitu:

1. Simulated\_annealing, yaitu fungsi yang mengelola proses iterasi pencarian algortma simulated\_annealing.

```

class SimulatedAnnealing:
    def __init__(self, initial_state: Cube, max_iter=100000,
initial_temp=10000, cooling_rate=0.77, threshold=0.5):
        self.state = initial_state
    
```

```

        self.max_iter = max_iter
        self.initial_temp = initial_temp
        self.cooling_rate = cooling_rate
        self.threshold = threshold
        self.best_state = initial_state.copy()
        self.best_value = self.state.value
        self.iteration = 0
        self.duration = 0
        self.objective_values = [self.state.value]

    def find_random_neighbor(self):
        # Generate a random neighbor by swapping two elements
        neighbor = self.state.cube.copy()
        x1, y1, z1 = np.random.randint(0, 5, 3)
        x2, y2, z2 = np.random.randint(0, 5, 3)

        # Ensure different positions for the swap
        while (x1, y1, z1) == (x2, y2, z2):
            x2, y2, z2 = np.random.randint(0, 5, 3)

        # Swap elements
        neighbor[x1, y1, z1], neighbor[x2, y2, z2] = neighbor[x2, y2, z2],
        neighbor[x1, y1, z1]
        return Cube(neighbor)

    def search(self):
        temperature = self.initial_temp
        start_time = time.time()

        for iteration in range(self.max_iter):
            self.iteration += 1
            neighbor = self.find_random_neighbor()
            neighbor_value = neighbor.value

            if neighbor_value < self.state.value:
                self.state = neighbor
                if neighbor_value < self.best_value:
                    self.best_state = neighbor
                    self.best_value = neighbor_value
            else:

                delta = neighbor_value - self.state.value
                acceptance_prob = np.exp(-delta / temperature)

                if acceptance_prob > self.threshold:
                    self.state = neighbor
                    if neighbor_value < self.best_value:
                        self.best_state = neighbor
                        self.best_value = neighbor_value
                else:
                    # Revert if not accepted
                    self.state = self.state

            self.objective_values.append(self.state.value)

            temperature *= self.cooling_rate

            if self.best_value == 0 or temperature < 1:
                break

```

```
end_time = time.time()
self.duration = end_time - start_time
```

- Def `__init__` : menginisialisasi attribute kelas Simulated annealing untuk menyimpan informasi-informasi yang diperlukan. Attribute-attribut yang disimpan yaitu, temperature, cooling\_rate, initial\_state, best\_state, best\_value, threshold, iteration, duration, dan array of objective value.
- Find\_random\_neighbor : mencari state neighbor dengan menukar 2 kubus. Jika kubus yang ditukar merupakan kubus yang sama, maka akan dicari ulang
- Search : melakukan pencarian dengan menggunakan algoritma simulated annealing, ketika didapatkan neighbor yang lebih kecil dari current state maka akan di bandingkan acceptace\_prob dengan threshold. Jika tidak melebihi threshold kubus ditukar lagi (pertukaran kubus tidak jadi dilakukan).

### 2.2.3 Genetic Algorithm

Algoritma genetika adalah algoritma optimasi yang terinspirasi dari proses seleksi alam. Pada algoritma ini, solusi awal ("populasi") diperbarui melalui proses seleksi, perkawinan silang (crossover), dan mutasi untuk menghasilkan generasi solusi baru dengan kualitas yang lebih baik. Setiap generasi diharapkan menghasilkan solusi yang mendekati atau mencapai tujuan optimasi.

Struktur Algoritma dan Source Kode Program.

```
import numpy as np
import random
from collections import Counter
from cube import *

class GeneticAlgorithm:
    def __init__(self, initial_state: Cube, n_population=100, n_generations=100,
                 elitism_rate=0.1):
        self.initial_state = initial_state
        self.n_population = n_population
        self.n_generations = n_generations
        self.elitism_rate = elitism_rate
        self.min_cost_per_generation = []
        self.avg_cost_per_generation = []
```

```

        self.population = self.__initialize_population()

    def __initialize_population(self):
        population = [Cube() for _ in range(self.n_population - 1)]
        population.append(self.initial_state)
        return population

    def __calculate_fitness(self, cubes):
        fitness = [1 / (1 + cube.value) for cube in cubes]
        return fitness

    def __selection(self, cumulative_probabilities, cubes):
        selected_cube = []
        for _ in range(2):
            r = random.random()
            for i, cp in enumerate(cumulative_probabilities):
                if r <= cp:
                    selected_cube.append(cubes[i].copy())
                    break
        return selected_cube

    def __crossover(self, selected_cubes):
        crossover_point = random.randint(0, 124)

        # Flatten to perform crossover
        cubel = np.array(selected_cubes[0].cube).flatten()
        cube2 = np.array(selected_cubes[1].cube).flatten()

        # Create children by swapping segments between the two crossover points
        child1 = np.concatenate((cubel[:crossover_point],
        cube2[crossover_point:])).reshape((5, 5, 5))
        child2 = np.concatenate((cube2[:crossover_point],
        cubel[crossover_point:])).reshape((5, 5, 5))

        return [Cube(child1), Cube(child2)]

    def __mutation(self, children, mutation_rate=0.1):
        mutated_children = []
        for child in children:
            if not self.is_cube_unique(child.cube):
                element_counts = Counter(child.cube.flatten())
                duplicates = [item for item, count in element_counts.items() if
count > 1]
                missing = [num for num in range(1, 126) if num not in
element_counts]

                for duplicate, miss in zip(duplicates, missing):
                    duplicate_index = np.where(child.cube == duplicate)
                    if len(duplicate_index[0]) > 0:
                        child.cube[duplicate_index[0][0], duplicate_index[1][0],
duplicate_index[2][0]] = miss

                if random.random() < mutation_rate:
                    # Randomly swap two positions
                    idx1, idx2 = random.sample(range(125), 2)
                    pos1 = np.unravel_index(idx1, (5, 5, 5))
                    pos2 = np.unravel_index(idx2, (5, 5, 5))
                    child.cube[pos1], child.cube[pos2] = child.cube[pos2],
child.cube[pos1]

```

```

        mutated_children.append(child)
    return mutated_children

def is_cube_unique(self, cube):
    # Check if all elements in the cube are unique
    flattened_cube = cube.flatten()
    return len(flattened_cube) == len(set(flattened_cube))

def run(self):
    for generation in range(self.n_generations):
        mutation_rate = max(0.05, 0.2 * (1 - generation / self.n_generations))
        fitness = self.__calculate_fitness(self.population)

        costs = [cube.value for cube in self.population]
        sorted_indices = np.argsort(costs)

        # Track min and average cost
        self.min_cost_per_generation.append(np.min(costs))
        self.avg_cost_per_generation.append(np.mean(costs))

        # Retain elites
        elite_size = max(1, int(self.n_population * self.elitism_rate * (1 -
generation / self.n_generations)))
        elites = [self.population[i] for i in sorted_indices[-elite_size:]]

        # Calculate cumulative probabilities
        probabilities = [f / sum(fitness) for f in fitness]
        cumulative_probabilities = np.cumsum(probabilities).tolist()

        # Select parents and generate children
        selected_cubes = self.__selection(cumulative_probabilities,
self.population)
        children = self.__crossover(selected_cubes)
        mutated_children = self.__mutation(children, mutation_rate)

        # Generate new population
        self.population = [Cube() for _ in range(self.n_population - elite_size
- 1)] + mutated_children + elites

        # Update fitness for new population
        fitness = self.__calculate_fitness(self.population)
        self.population = [self.population[i] for i in
np.argsort(fitness)[-self.n_population:]]

        # Select best cube
        best_cube = min(self.population, key=lambda cube: cube.value)

    return {
        "iterations": self.n_generations,
        "initial_state": self.initial_state(cube.tolist()),
        "final_state": best_cube(cube.tolist()),
        "objective_value": int(best_cube.value),
        "min_cost_per_generation": self.min_cost_per_generation,
        "avg_cost_per_generation": self.avg_cost_per_generation,
    }
}

```

Penjelasan :

### 1. Inisialisasi Parameter (Constructor: `__init__`)

Konstruksi kelas memerlukan beberapa parameter utama:

- `initial_state`: Objek Cube yang digunakan sebagai solusi awal dalam populasi.
- `n_population`: Jumlah individu (solusi) dalam populasi, default-nya 100.
- `n_generations`: Jumlah generasi (iterasi) yang akan dijalankan, default-nya 100.
- `elitism_rate`: Persentase elitisme, yakni proporsi individu terbaik yang akan dipertahankan pada setiap generasi untuk menjaga kualitas populasi.

Selain itu, ada beberapa atribut tambahan yang diinisialisasi untuk melacak perkembangan algoritma:

- `min_cost_per_generation` dan `avg_cost_per_generation`: List yang menyimpan nilai biaya minimum dan rata-rata di setiap generasi, untuk memantau perkembangan performa algoritma dari waktu ke waktu.

## 2. Membuat Populasi Awal (`__initialize_population`)

Metode ini menghasilkan populasi awal. Sebagian besar individu dalam populasi dibuat secara acak, kecuali satu individu yang merupakan `initial_state` yang diberikan. Hal ini memastikan bahwa kita memiliki solusi awal yang dapat dikembangkan lebih lanjut dalam proses evolusi.

## 3. Menghitung Nilai Fitness (`__calculate_fitness`)

Fungsi fitness digunakan untuk mengevaluasi "kualitas" setiap individu dalam populasi. Dalam konteks ini, fitness dihitung sebagai  $1 / (1 + \text{cube.value})$ , di mana `cube.value` adalah nilai biaya dari objek Cube. Semakin rendah nilai biaya `cube.value`, semakin tinggi fitness-nya, sehingga individu dengan biaya lebih rendah lebih cenderung dipilih untuk reproduksi.

## 4. Seleksi Individu untuk Crossover (`__selection`)

Metode seleksi ini memilih dua individu berdasarkan probabilitas kumulatif yang dihitung dari fitness. Dalam metode ini:

- `cumulative_probabilities`: Probabilitas kumulatif dari seluruh individu, berdasarkan fitness mereka.

- Dengan teknik ini, individu dengan fitness lebih tinggi (biaya lebih rendah) lebih sering terpilih, yang berarti individu yang lebih baik memiliki peluang lebih besar untuk berkembang.

#### 5. Crossover (Rekombinasi) (crossover)

Crossover adalah proses pertukaran informasi antara dua individu yang terpilih untuk menghasilkan dua "anak". Metode ini dilakukan dengan cara:

- Memilih titik crossover secara acak.
- Membagi array kubus (dalam format 3D) dari dua orang tua pada titik tersebut dan menukar segmen kubus di antara titik tersebut.
- Anak-anak yang dihasilkan akan memiliki karakteristik campuran dari kedua orang tua, memungkinkan variasi dalam populasi dan eksplorasi solusi yang lebih luas.

#### 6. Mutasi (mutation)

Mutasi dilakukan pada individu baru untuk menjaga keberagaman genetik dalam populasi. Mutasi dilakukan dengan:

- Menyesuaikan posisi elemen yang duplikat untuk memastikan tidak ada pengulangan nilai dalam kubus.
- Secara acak menukar dua posisi dalam kubus berdasarkan mutation\_rate.

Mutation Rate adalah probabilitas suatu individu akan mengalami mutasi. Nilai ini biasanya diturunkan seiring berjalannya generasi untuk meningkatkan stabilitas populasi saat mendekati solusi optimal. Dengan mutasi, algoritma genetika bisa menghindari terjebak di solusi lokal.

#### 7. Memastikan Keunikan Elemen dalam Kubus (is\_cube\_unique)

Fungsi ini memeriksa apakah kubus memiliki elemen unik (tidak ada duplikasi). Keunikan ini perlu dijaga agar konfigurasi kubus tetap valid.

#### 8. Proses Utama Algoritma (run)

Metode run adalah inti dari algoritma genetika yang mengulangi siklus generasi dan mutasi untuk meningkatkan solusi secara bertahap.

- Elitisme: Elitisme mempertahankan persentase tertentu (`elitism_rate`) dari individu terbaik di setiap generasi, memungkinkan mereka untuk langsung dibawa ke generasi berikutnya. Elitisme mengurangi risiko hilangnya solusi optimal yang sudah ditemukan.
- Menghitung Fitness: Pada setiap generasi, fitness populasi dihitung untuk mengetahui kualitas masing-masing individu.
- Crossover dan Mutasi: Setelah seleksi, individu-individu mengalami crossover dan mutasi sesuai dengan aturan yang ditetapkan.
- Populasi Baru: Populasi baru kemudian dibentuk dari individu hasil mutasi dan individu elit dari generasi sebelumnya.
- Pembaruan Statistik: Biaya minimum dan rata-rata di setiap generasi disimpan untuk melacak kemajuan dari algoritma.

Di akhir proses, algoritma mengembalikan informasi dari solusi terbaik yang ditemukan, termasuk:

- `iterations`: Jumlah generasi yang dijalankan.
- `initial_state`: Kondisi awal dari kubus.
- `final_state`: Kondisi akhir dari kubus terbaik yang ditemukan.
- `objective_value`: Nilai tujuan dari solusi optimal (nilai `cube.value` yang paling rendah).
- `min_cost_per_generation` dan `avg_cost_per_generation`: Statistik minimum dan rata-rata biaya di setiap generasi.

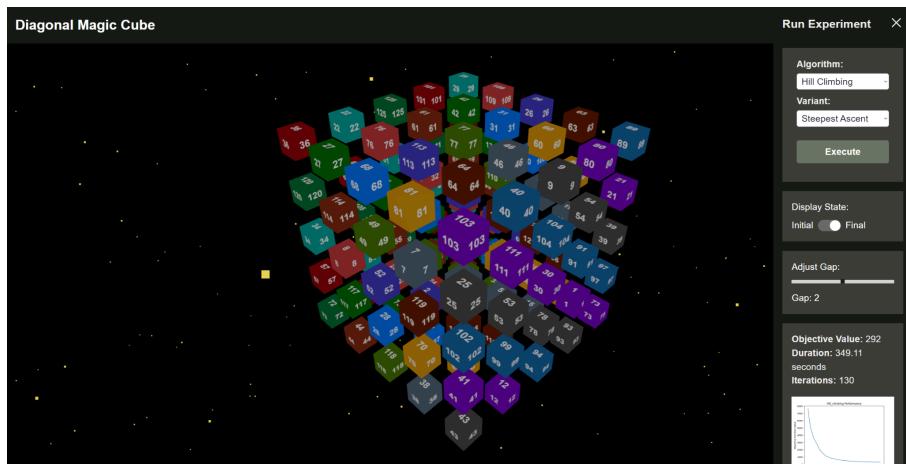
## 2.3 Hasil Eksperimen dan Analisis

### 2.3.1 Hill Climbing

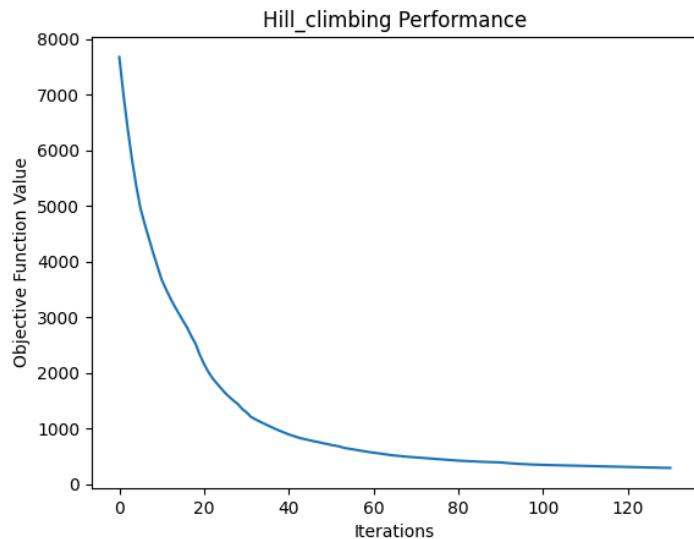
Hasil Eksperimen dengan masing-masing varian Hill-Climbing

A. Steepest Ascent

- Tangkapan Layar Steepest Ascent 1:



Plot:

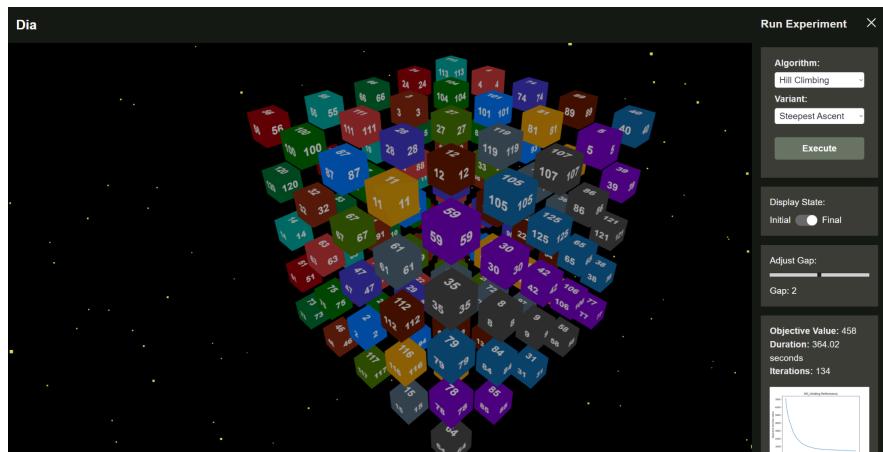


### Results

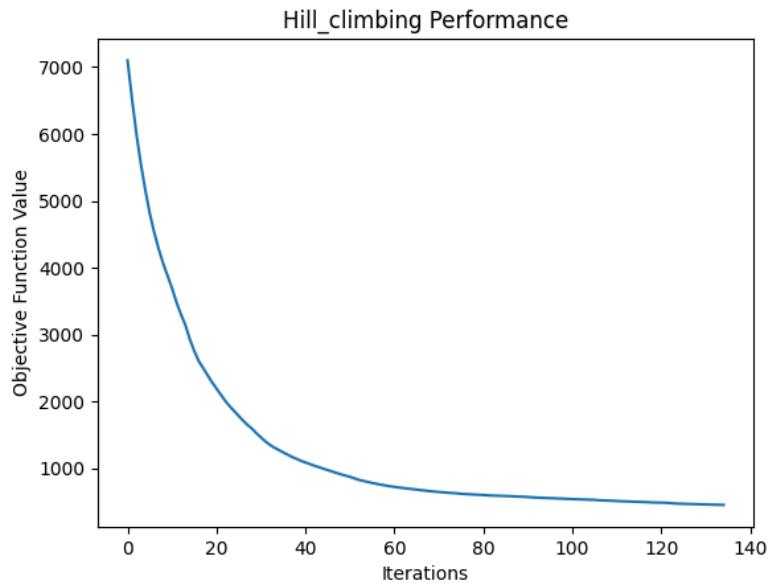
Objective value = 292 (or -292 when inverted)

Iterations = 130

- Tangkapan Layar Steepest Ascent 2:



Plot:

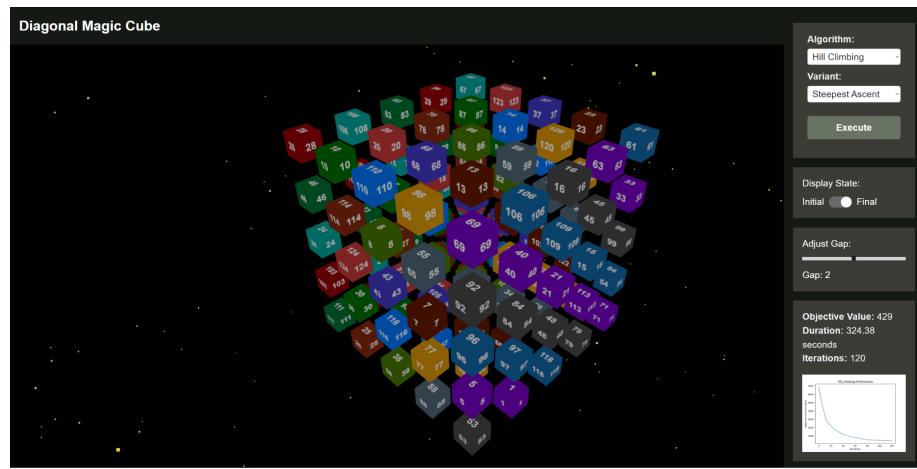


### Results

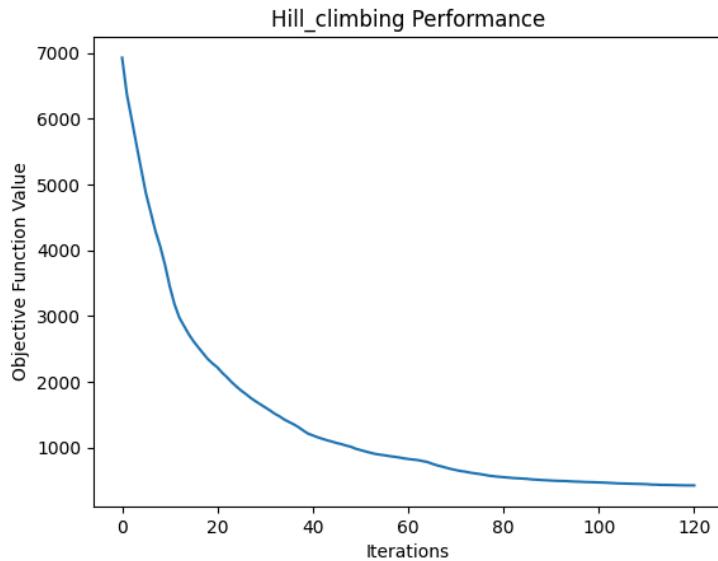
Objective value = 458 (or -458 when inverted)

Iterations = 134

- Tangkapan Layar Steepest Ascent 3:



Plot:



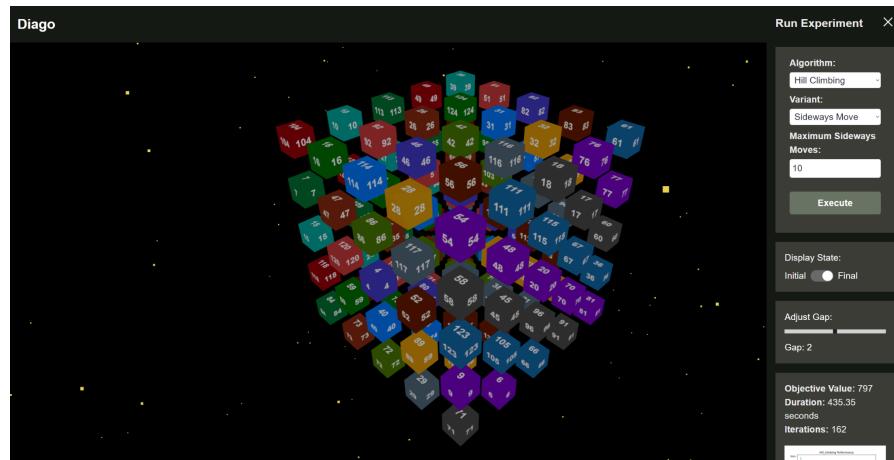
### Results

Objective value = 429 (or -429 when inverted)

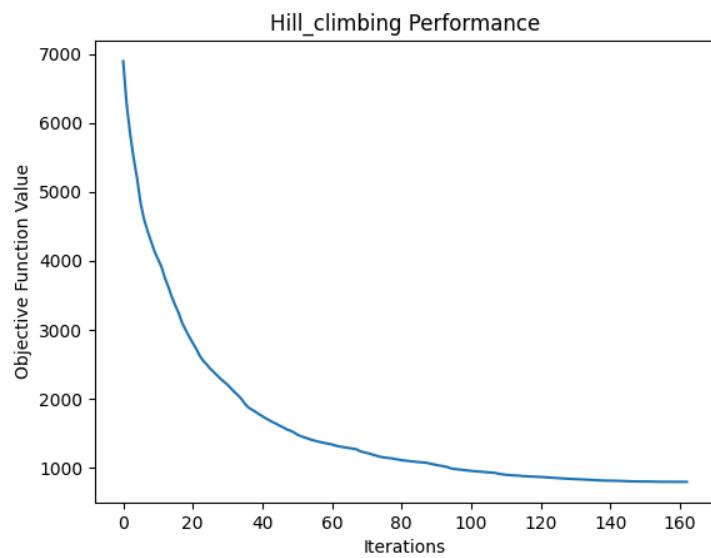
Iterations = 120

## B. Steepest Ascent with Sideways Move Hill-Climbing

- Tangkapan Layar Steepest Ascent with Sideways Move 1:



Plot:



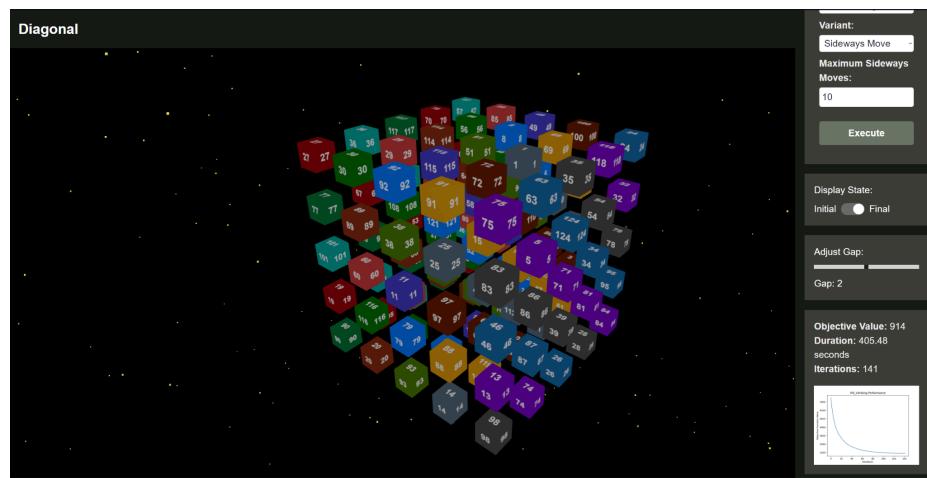
## Results

Objective value = 797 (or -797 when inverted)

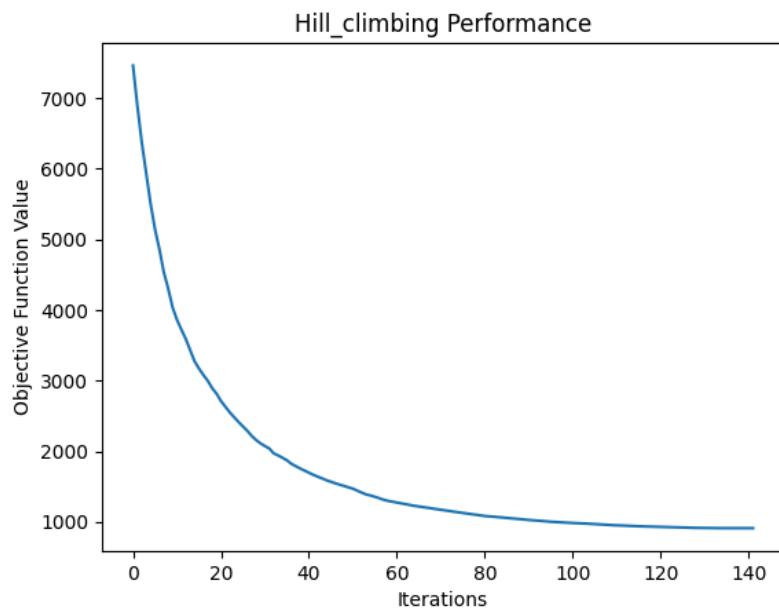
Iterations = 162

Maximum Sideways Move = 10

- Tangkapan Layar Steepest Ascent with Sideways Move 2:



Plot:



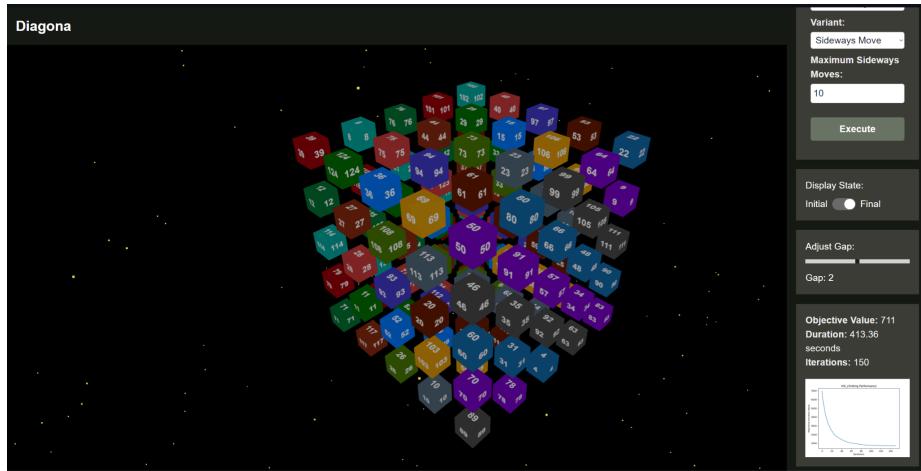
### Results

Objective value = 914 (or -914 when inverted)

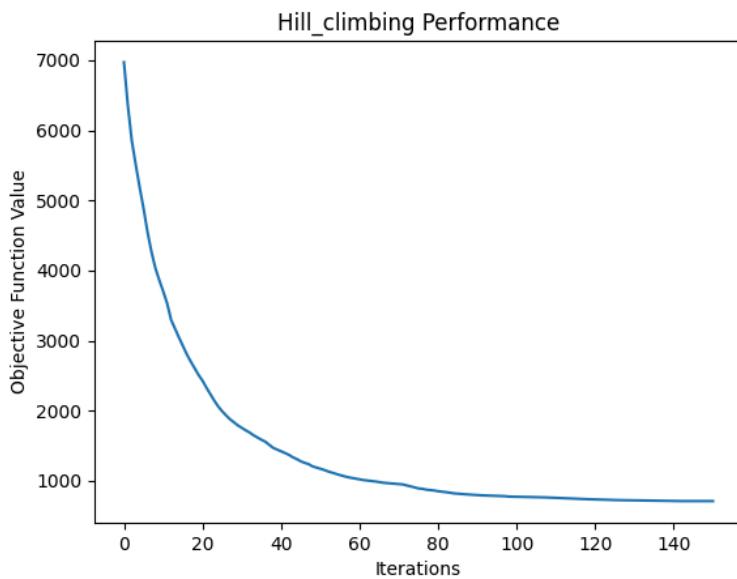
Iterations = 141

Maximum Sideways Move = 10

- Tangkapan Layar Steepest Ascent with Sideways Move 3:



Plot:



### Results

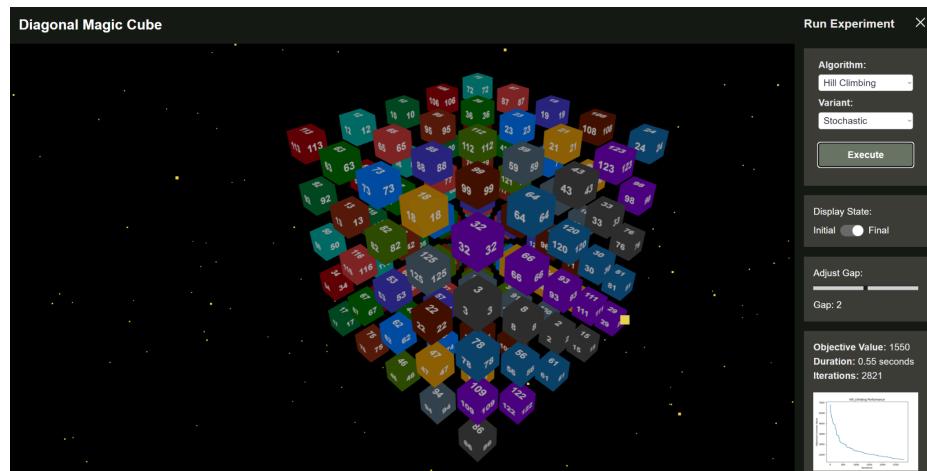
Objective value = 711 (or -711 when inverted)

Iterations = 150

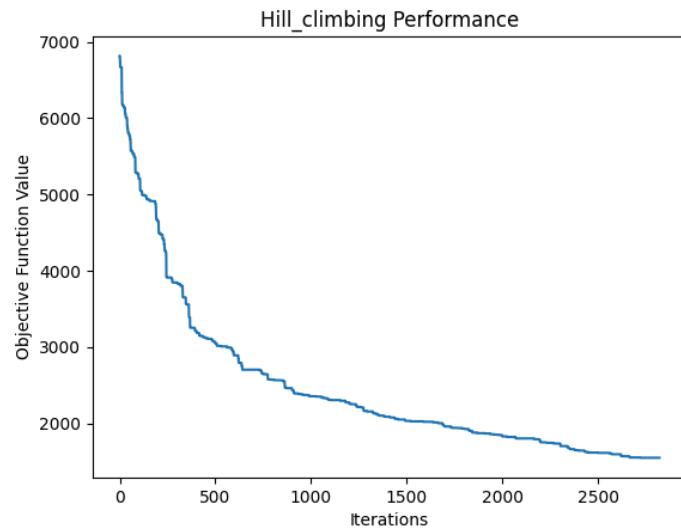
Maximum Sideways Move = 10

## C. Stochastic Hill-Climbing

- Tangkapan Layar Stochastic 1:



Plot:



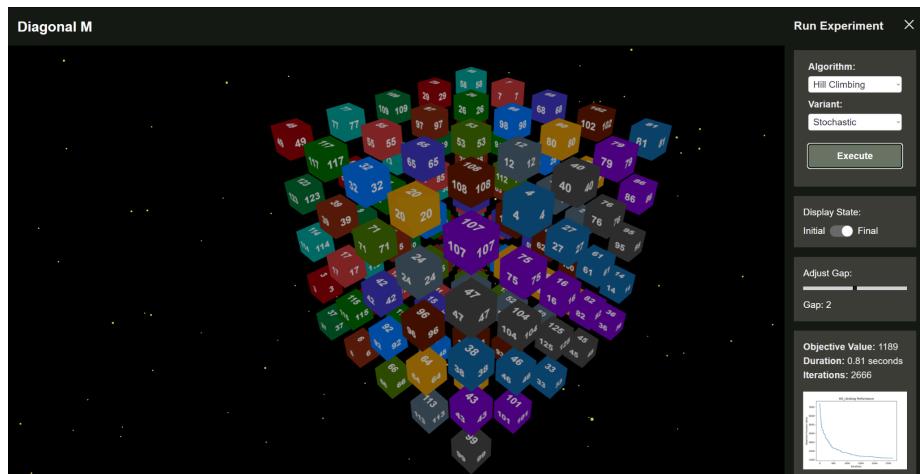
## Results

Objective value = 1550 (or -1550 when inverted)

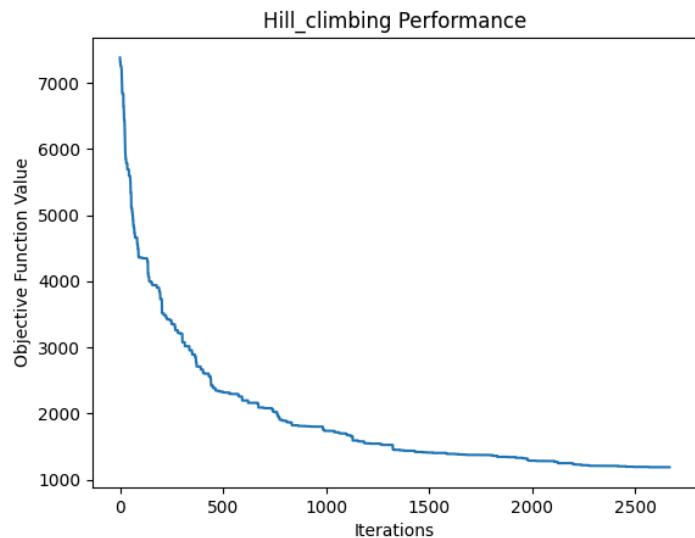
Iterations = 2821

Tolerance = 100

- Tangkapan Layar Stochastic 2:



Plot:



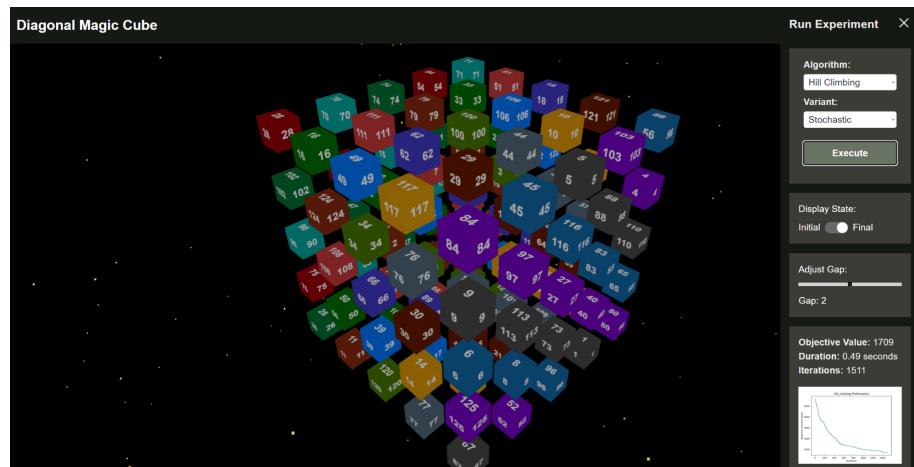
Results

Objective value = 1189 (or -1189 when inverted)

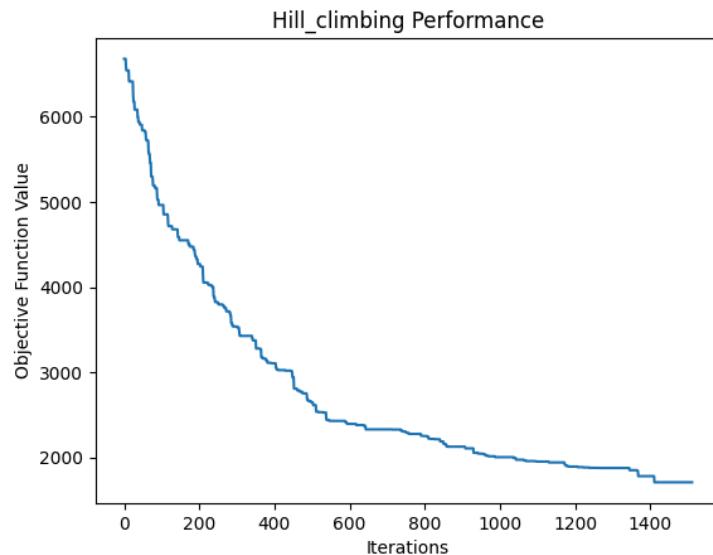
Iterations = 2666

Tolerance = 100

- Tangkapan Layar Stochastic 3:



Plot:



## Results

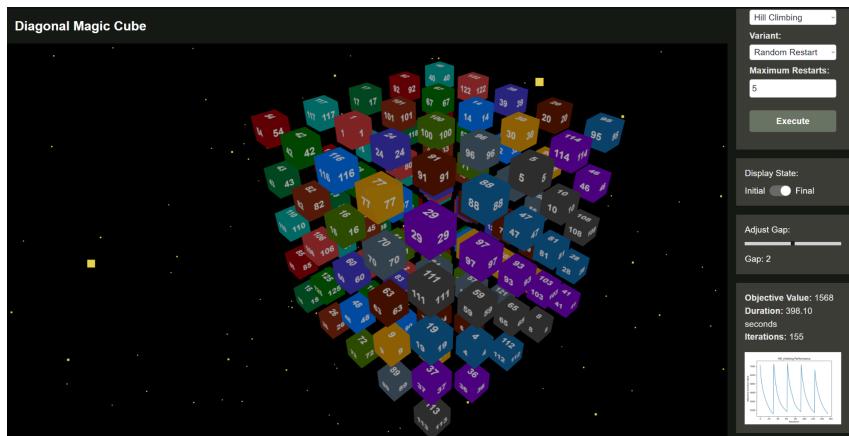
Objective value = 1709 (or -1709 when inverted)

Iterations = 1511

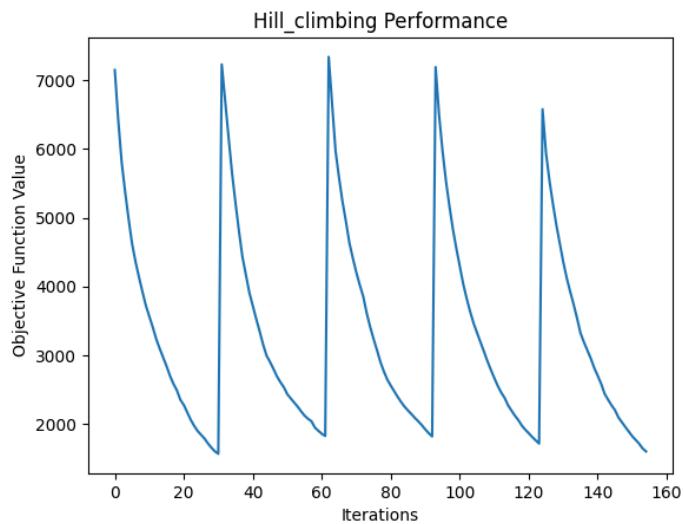
Tolerance = 100

## D. Random Restart

- Tangkapan Layar Random Restart 1:



Plot:



## Results

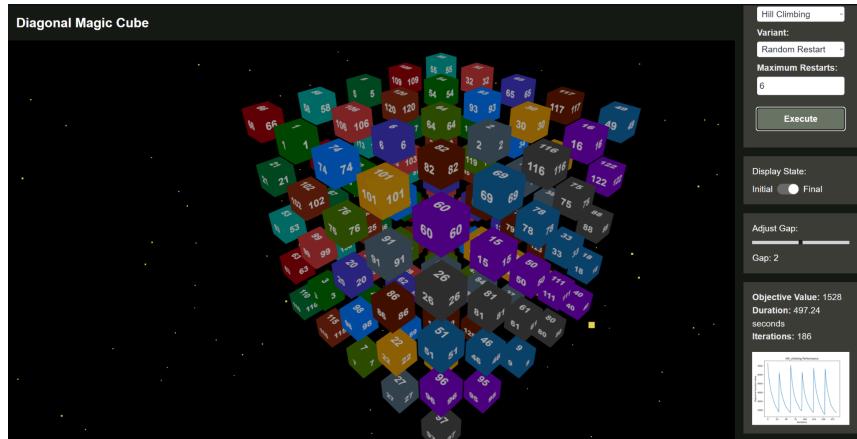
Objective value = 1568 (or -1568 when inverted)

Iterations = 155 (including 0th and new initial state)

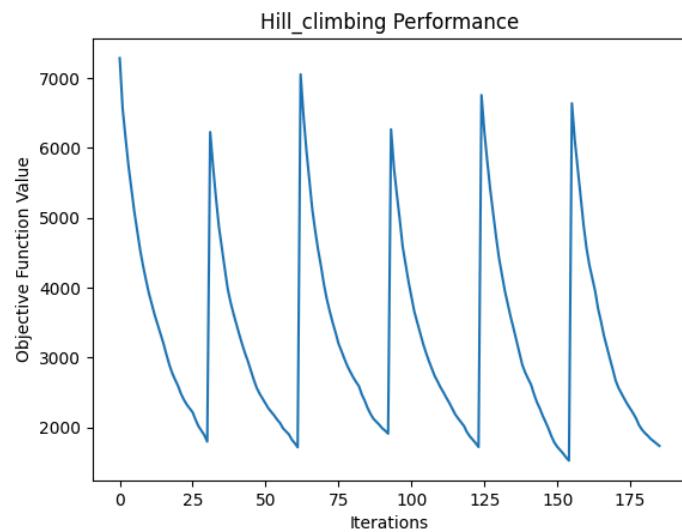
maxRestart = 5

Iterations per restart = 30

- Tangkapan Layar Random Restart 2:



Plot:



## Results

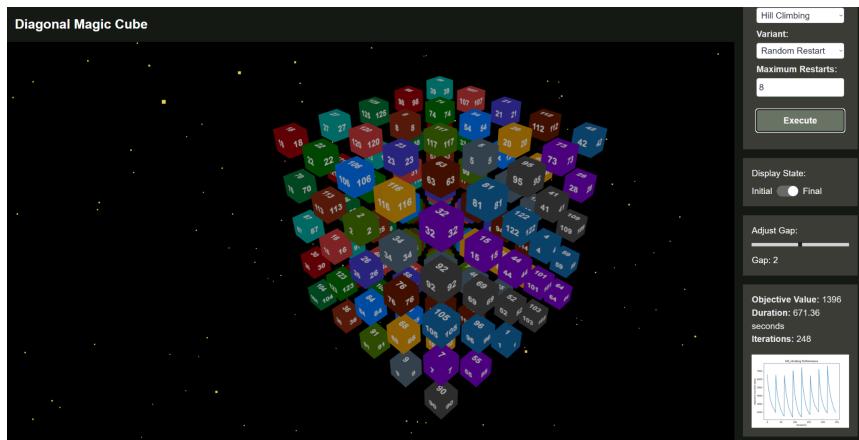
Objective value = 1528 (or -1568 when inverted)

Iterations = 186 (including 0th and new initial state)

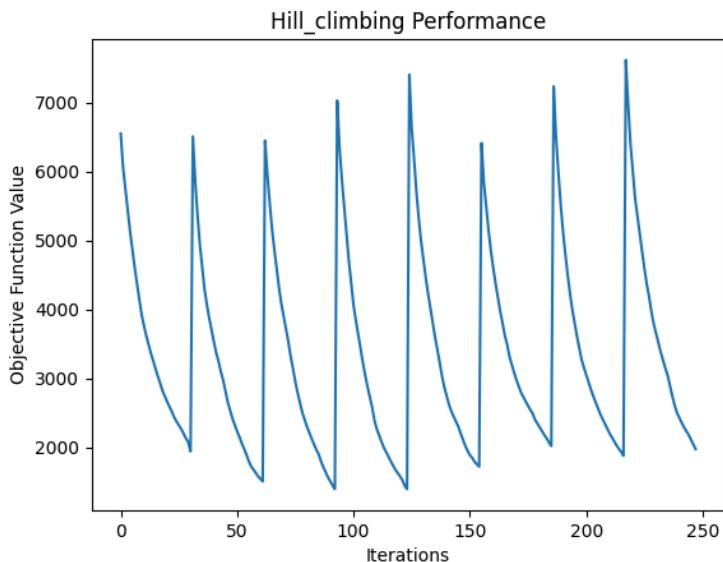
maxRestart = 6

Iterations per restart = 30

- Tangkapan Layar Random Restart 3:



Plot:



### Results

Objective value = 1396 (or -1396 when inverted)

Iterations = 248 (including 0th and new initial state)

maxRestart = 8

Iterations per restart = 30

### Analisis Percobaan Hill-Climbing

Waktu yang dibutuhkan untuk melakukan *Steepest Ascent Hill-Climbing* dan variannya membutuhkan waktu yang sangat lama dibanding algoritma yang memilih *neighbor* secara acak.

Ini karena setiap iterasi dari varian ini, akan dilakukan  $\sum_{i=1}^{124} i = 7750$  pencarian untuk *successors*

yang mungkin, menyebabkan fungsi evaluasi dipanggil sebanyak itu pula. Sehingga  $1\text{iterasi} \textit{Steepest} \approx 7750\text{iterasi} \textit{Stochastic}$ .

Setiap algoritma hill-climbing membentuk kurva yang logaritmik. Pada iterasi-iterasi awal, perubahan yang besar sering terjadi kemudian melambat hingga tidak menemukan neighbor dengan nilai yang lebih baik. Pada *Stochastic*, dapat terjadi pengambilan neighbor dengan value yang dekat dengan state sebelumnya.

Hasil dari algoritma *Steepest Ascent with Sideways Move* tidak jauh berbeda dengan hasil dari *Steepest Ascent* biasa. Ini terjadi karena fungsi objektif yang memiliki jangkauan yang luas sehingga jarang ditemukan neighbor dengan nilai objektif yang sama. Mungkin hanya akan menemukan beberapa neighbor yang sama, tetapi sideways move yang terjadi hanya akan terjadi sedikit sehingga menambah parameter maksimum sideways tidak terlalu berpengaruh kecuali untuk menghindari *loop* antara dua state akhir.

State yang dihasilkan dari algoritma *Random Restart* dengan parameter maxRestart berbeda, memiliki nilai yang tidak jauh berbeda jika dilihat dari grafik yang dibentuk. Nilai objektif state yang dihasilkan akan jauh lebih baik jika iterasi yang dilakukan setiap restart lebih banyak. Terutama dari analisis iterasi *Steepest Ascent*, nilai objektif memiliki lompatan dalam suatu jangkauan yang serupa pada nilai objektif tertentu.

### 2.3.2 Simulated Annealing

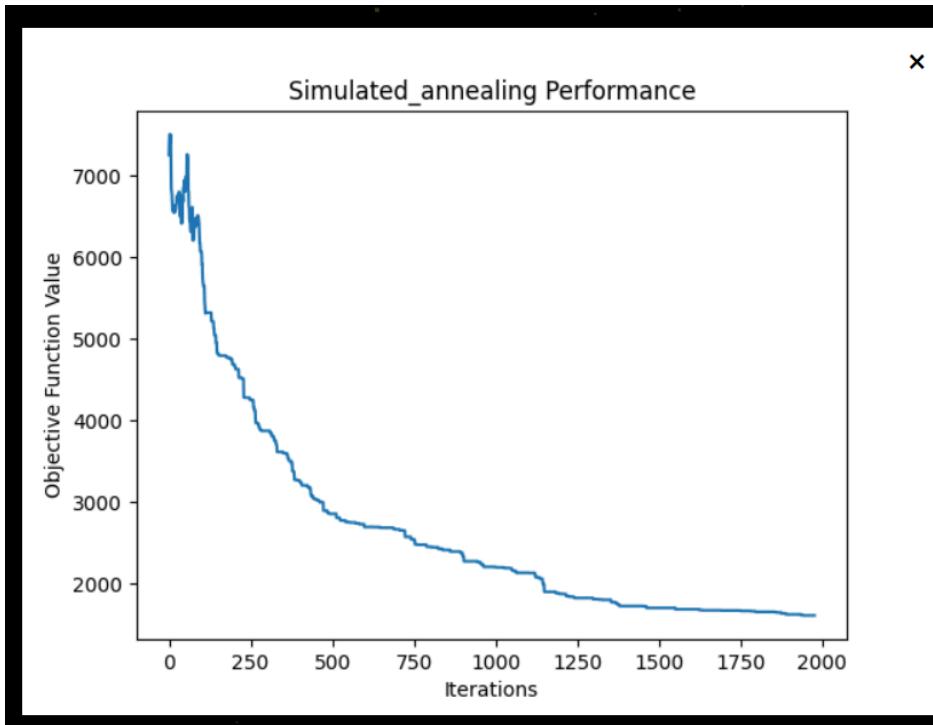
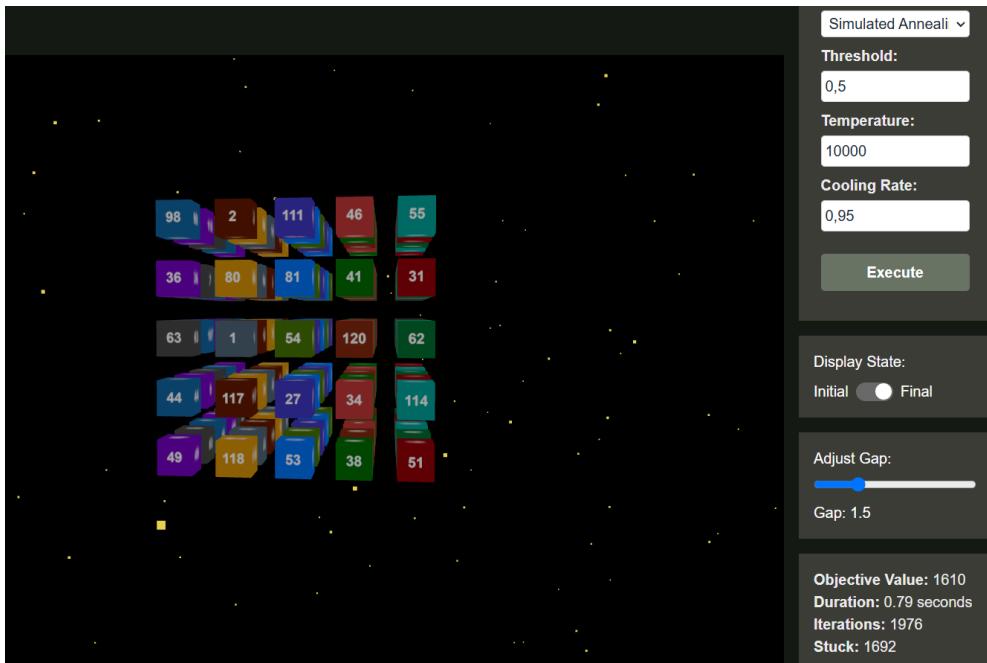
Hasil Eksperimen dengan berbagai konfigurasi adalah sebagai berikut.

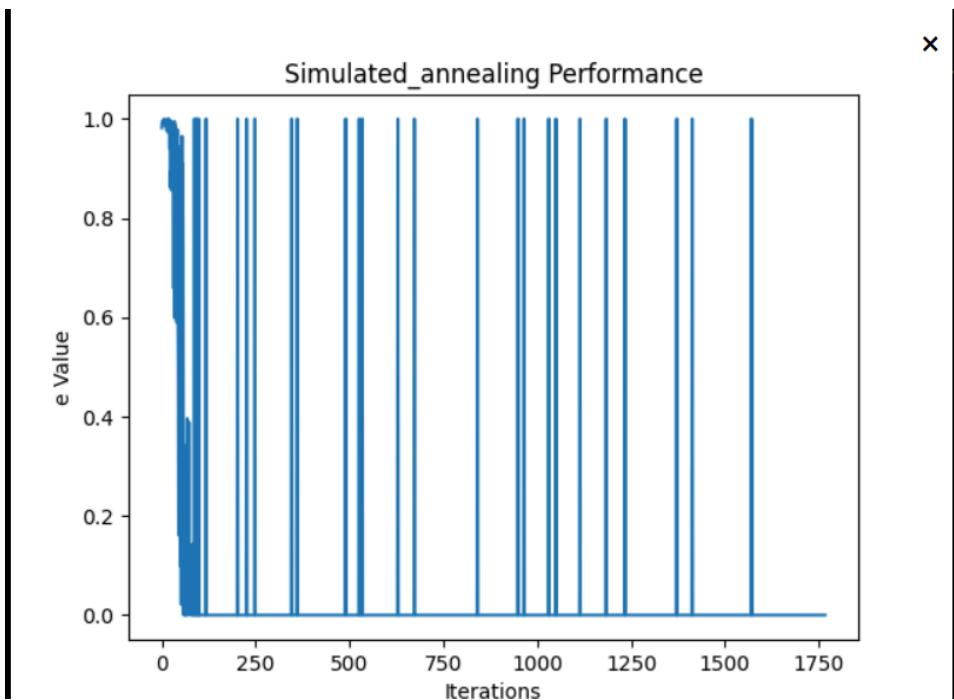
- Percobaan 1

Temperature : 10000

Threshold : 0,5

Cooling rate : 0.95





Berdasarkan hasil eksperimen didapatkan:

Objective value : 1610

Duration : 0.79 seconds

Iterations : 1976

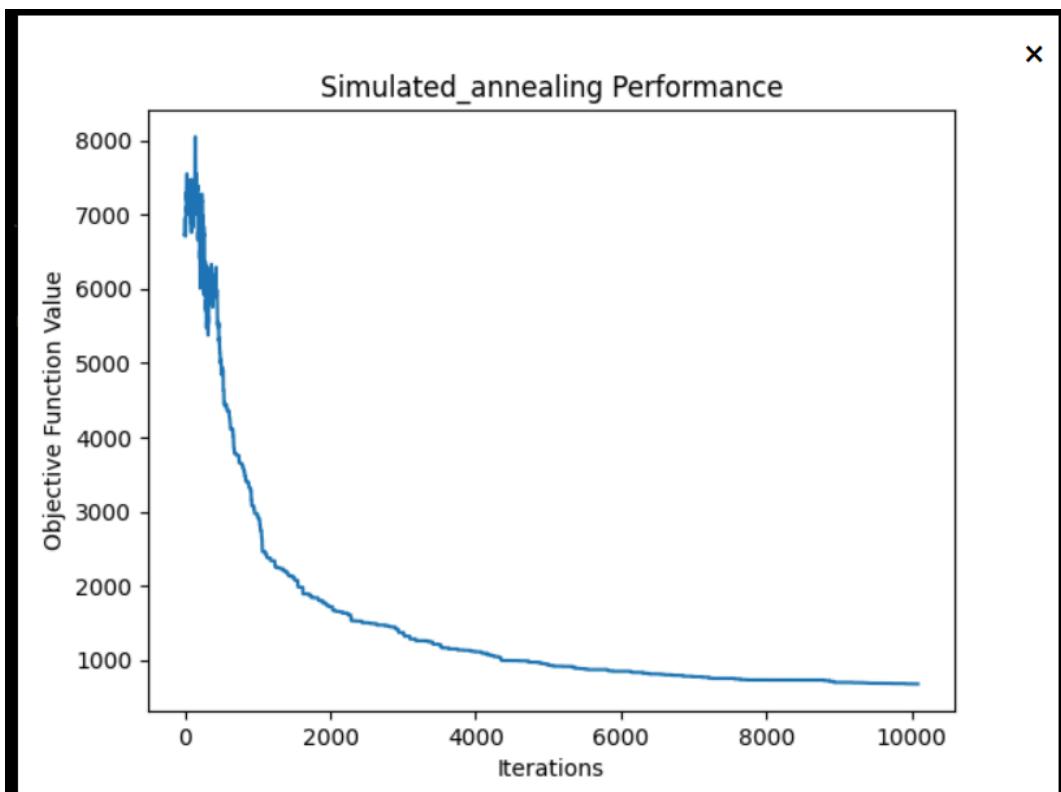
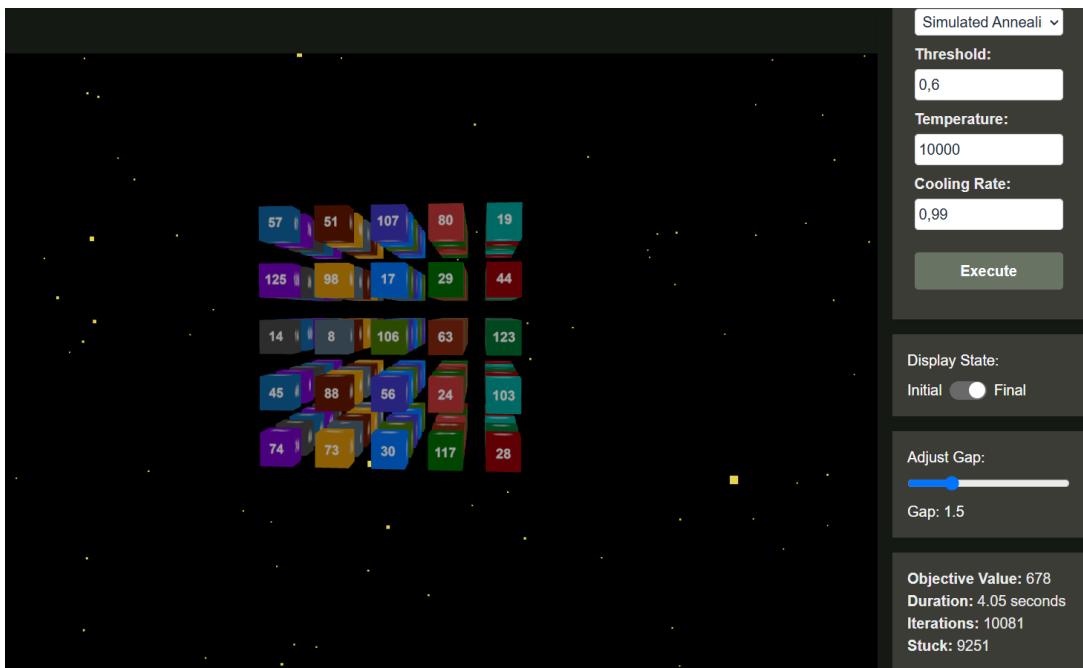
Stuck : 1692

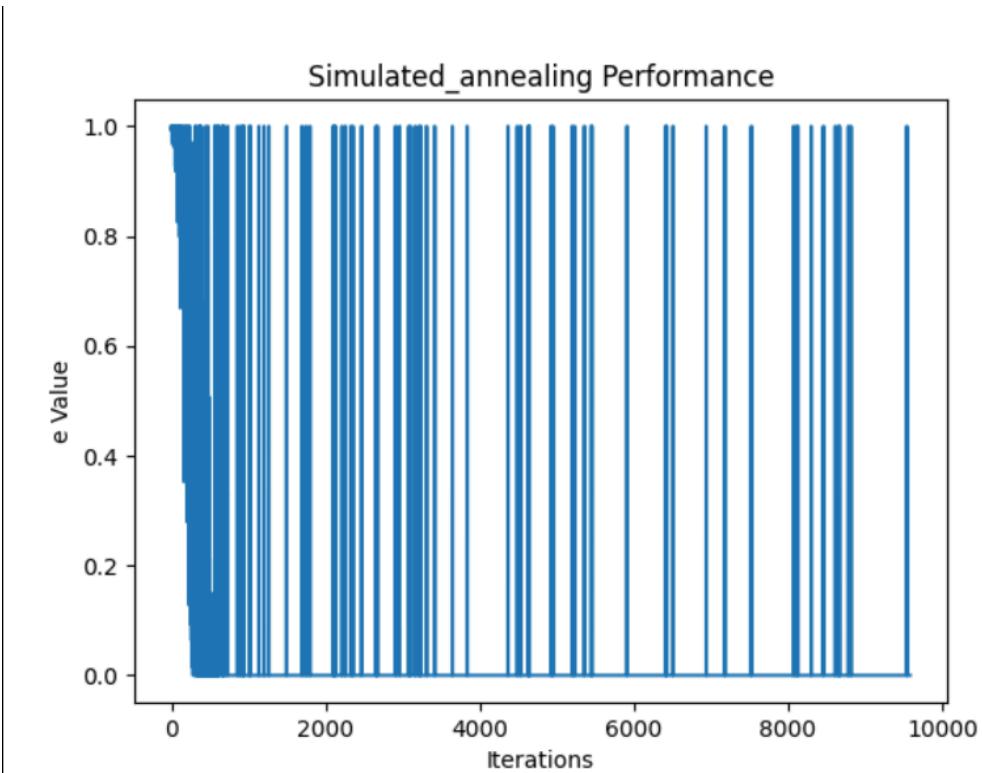
- Percobaan 2

Temperature : 10000

Threshold : 0,6

Cooling rate : 0.99





Berdasarkan hasil eksperimen didapatkan:

Objective value : 678

Duration : 4.06 seconds

Iterations : 10081

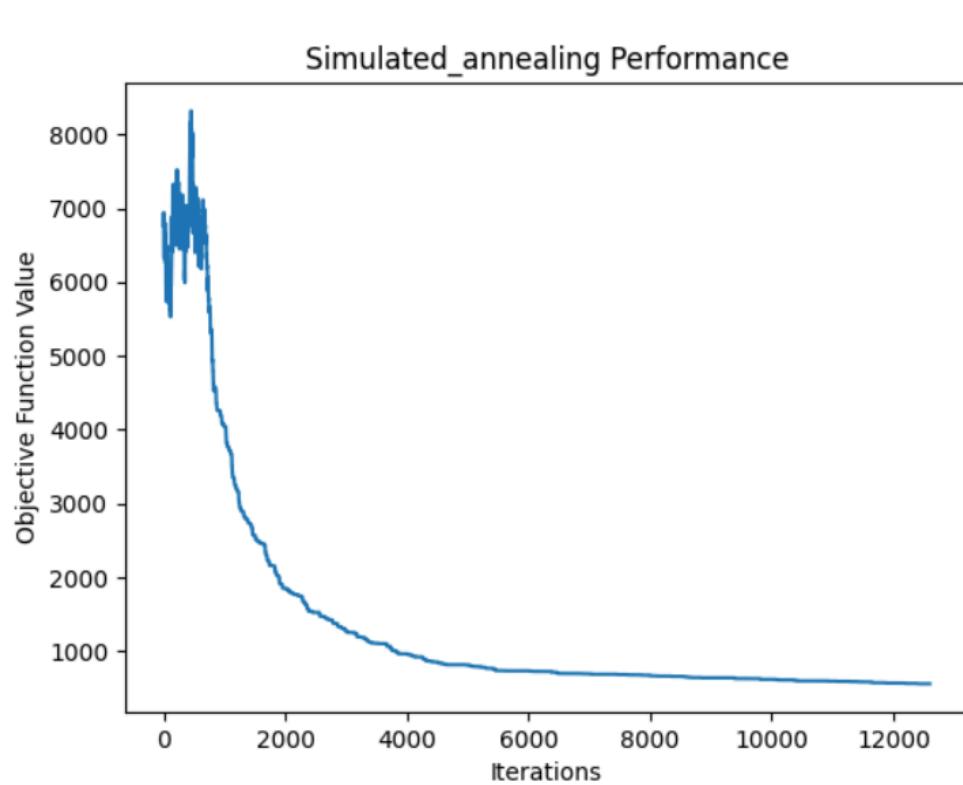
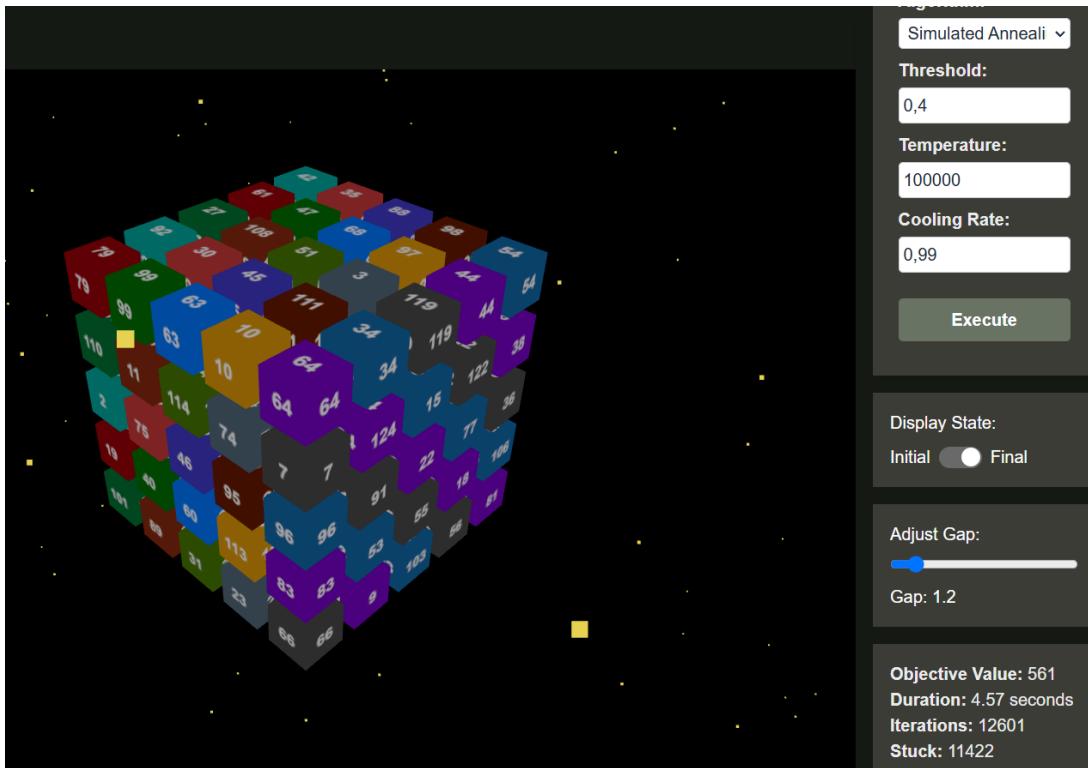
Stuck : 9251

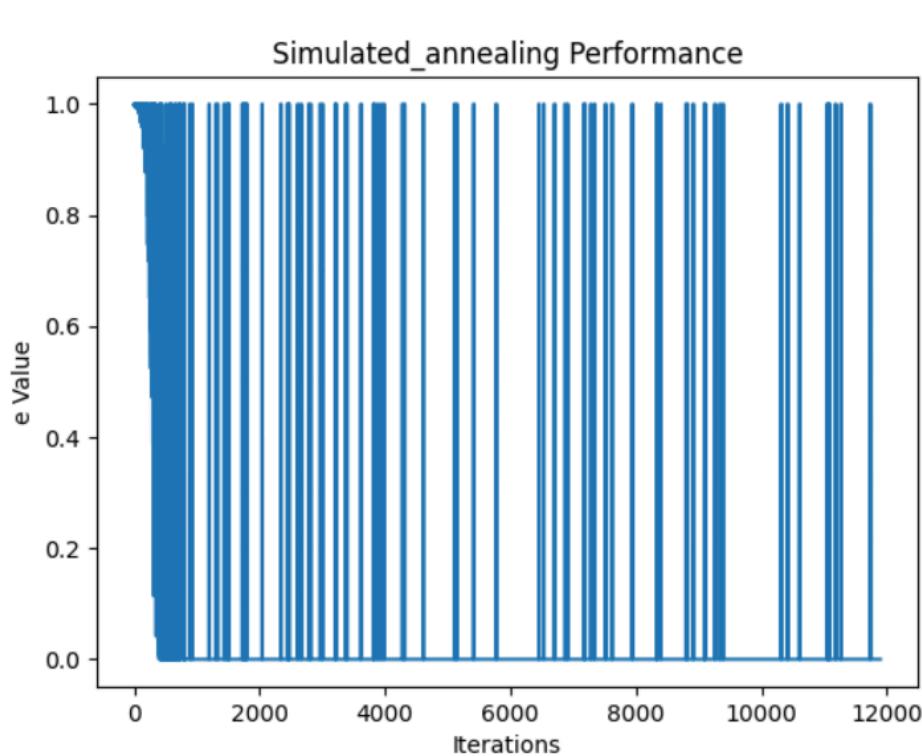
- Percobaan 3

Temperature : 100000

Threshold : 0,4

Cooling rate : 0,99





Berdasarkan hasil eksperimen didapatkan:

Objective value : 561

Duration : 4.57 seconds

Iterations : 12601

Stuck : 11422

Dari beberapa eksperimen yang telah dilakukan pada algoritma Simulated Annealing didapatkan hasil analisis sebagai berikut.

Dari nilai objective value yang didapatkan dapat dilihat hasilnya sudah mendekati global optimum. Hal ini bisa didapatkan karena Pada algoritma Simulated annealing terdapat parameter

yang dapat menerima neighbor value lebih kecil jika nilai  $e^{\frac{\Delta E}{T}} >$  threshold sehingga dapat terhindar dari local optima. Dengan menggunakan Temperatur dan cooling rate yang sesuai proses iterasi bisa berlangsung lebih banyak. Semakin banyak iterasi semakin mendekati nilai global optimum.

Semakin banyak iterasi semakin lama durasi pencarian. Banyaknya iterasi menandakan semakin banyak langkah yang diaktifkan, dengan parameter threshold yang tepat setiap pilihan iterasi akan mengarahkan hasil ke solusi optimum. Jadi semakin lama durasi pencarian semakin baik hasil yang didapatkan.

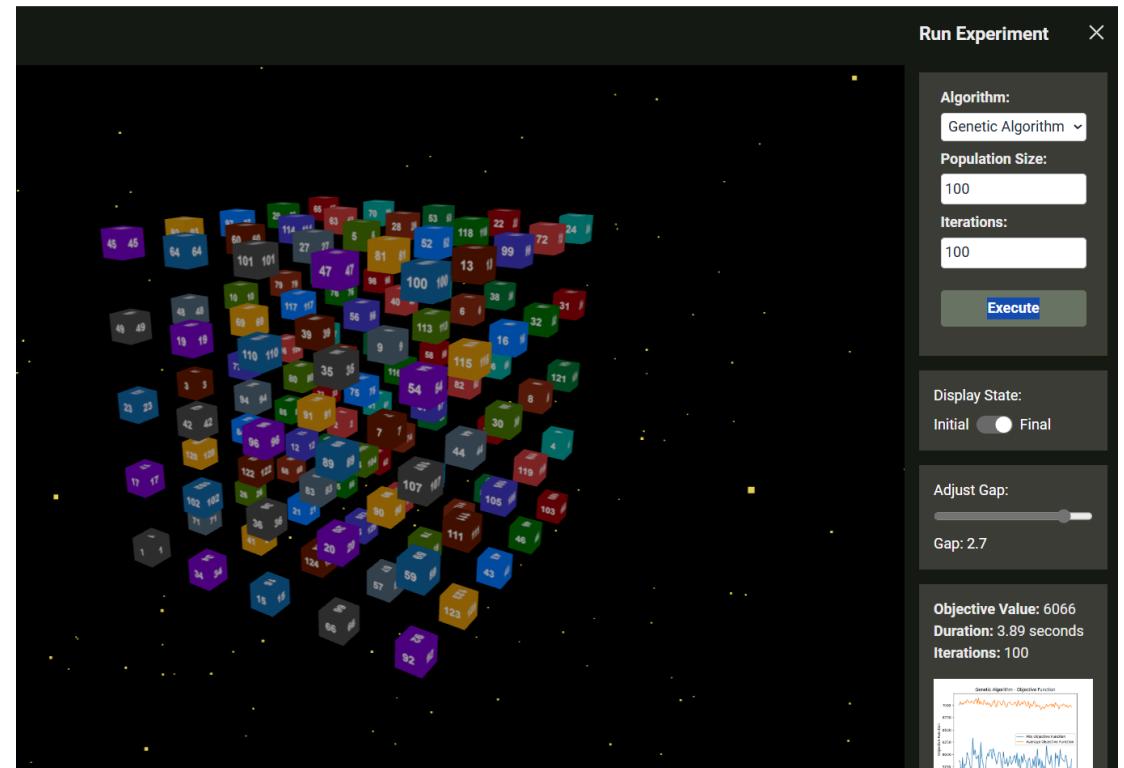
Dibandingkan dengan algoritma local search yang lain, simulated annealing mendapatkan hasil yang paling optimal. Dengan waktu yang lebih singkat algoritma simulated annealing dapat menghasilkan objective value yang mendekati global optimum bahkan sampai global optimum. Dengan penilaian waktu dan solusi optimum algoritma ini bisa menjadi pilihan yang terbaik.

Hasil yang didapatkan oleh algoritma simulated annealing tidak konsisten. Hal ini karena algoritma ini menggunakan mekanisme random successor algoritma ini menghasilkan solusi yang tidak konsisten. Contohnya pada percobaan 3 dilakukan beberapa kali pengulangan didapatkan hasilnya dalam rentang 500-900.

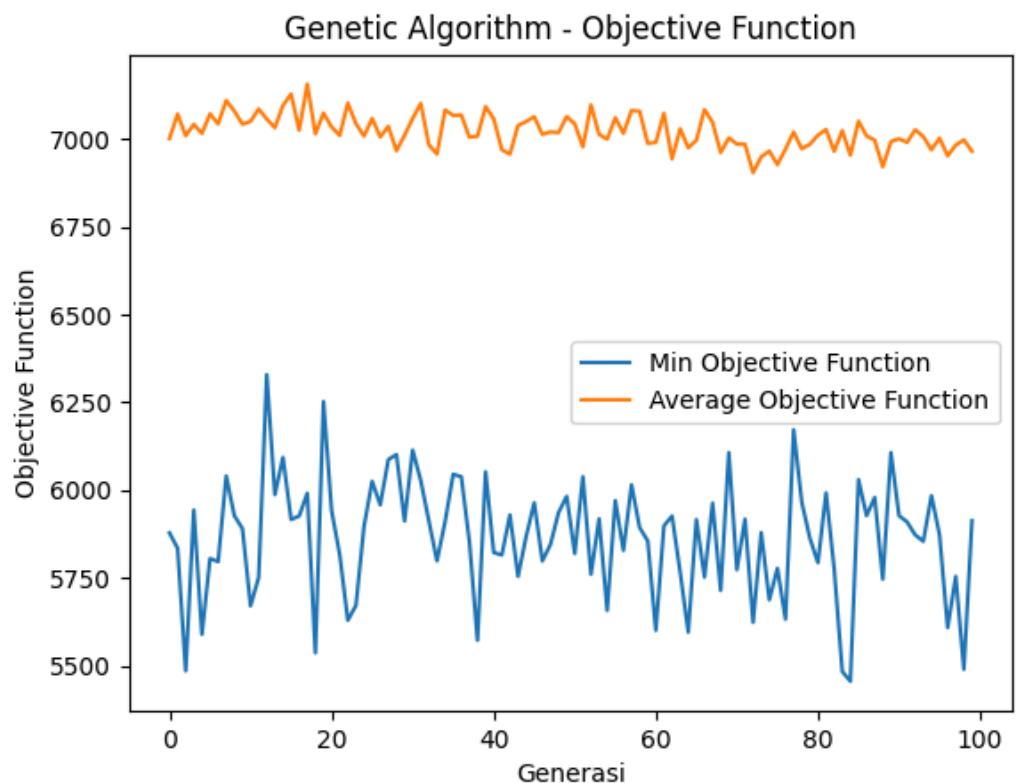
### **2.3.3 Genetic Algorithm**

Hasil Eksperimen dengan berbagai konfigurasi adalah sebagai berikut.

- Populasi Sebagai kontrol (n\_populasi : 100)
  - Iterasi : 100

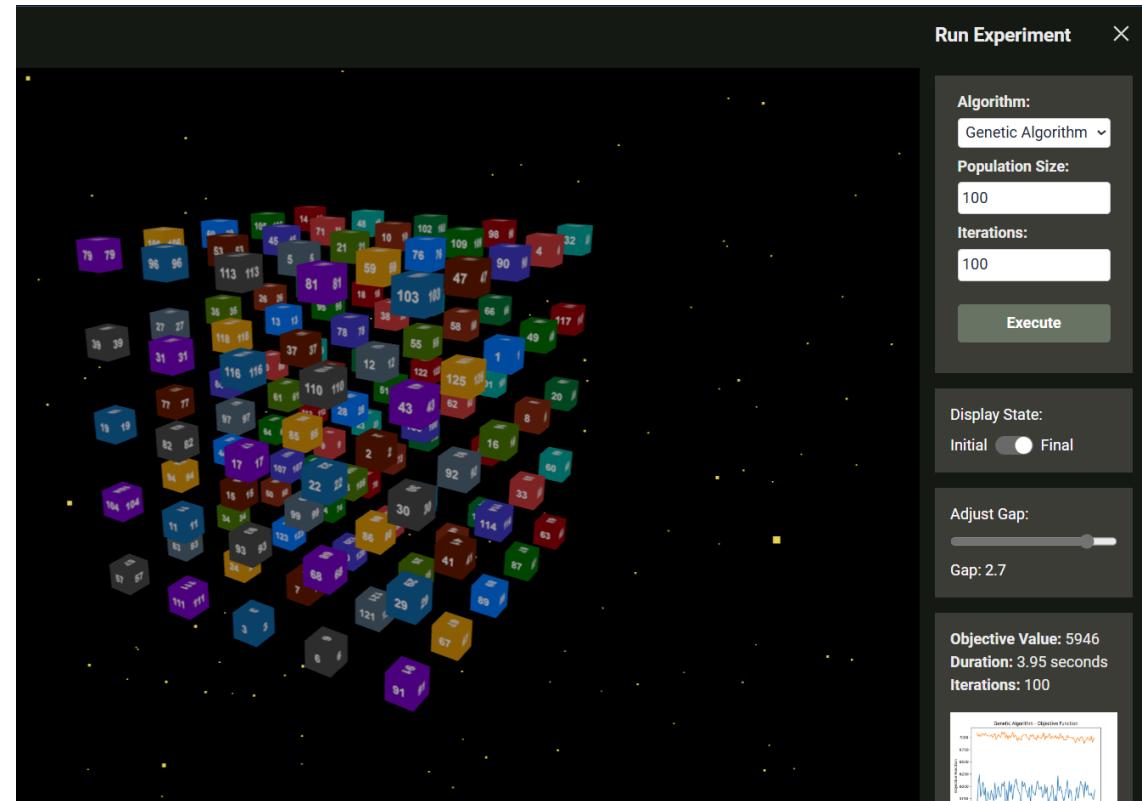


1.

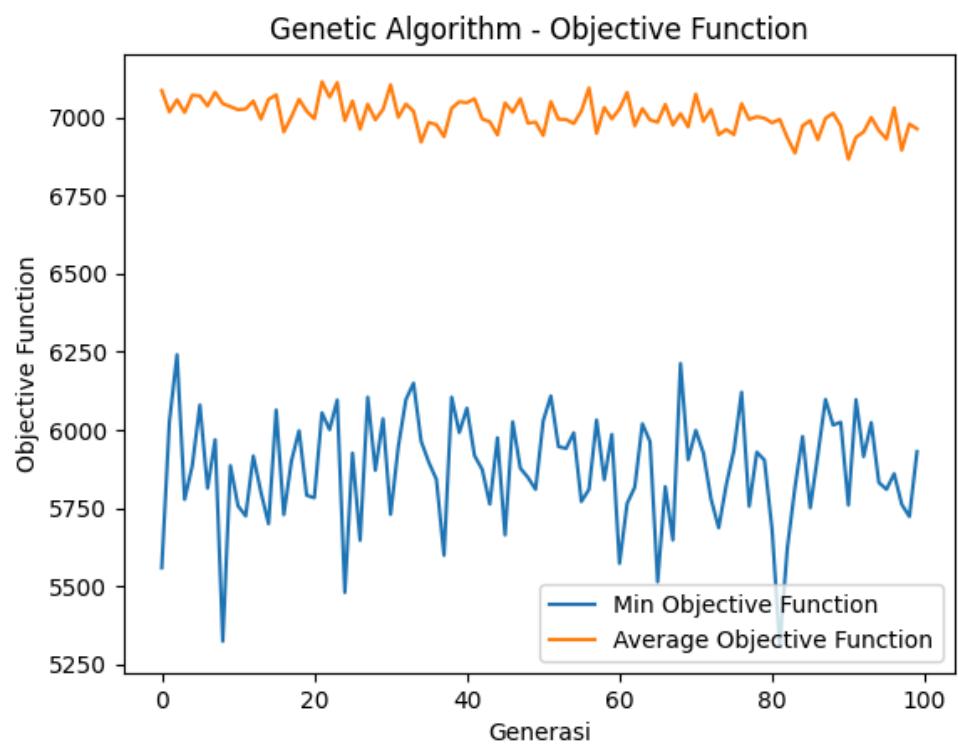


Berdasarkan dari hasil eksperimen didapat :

- Objektif Function akhir → 6066
- Durasi → 3.89s

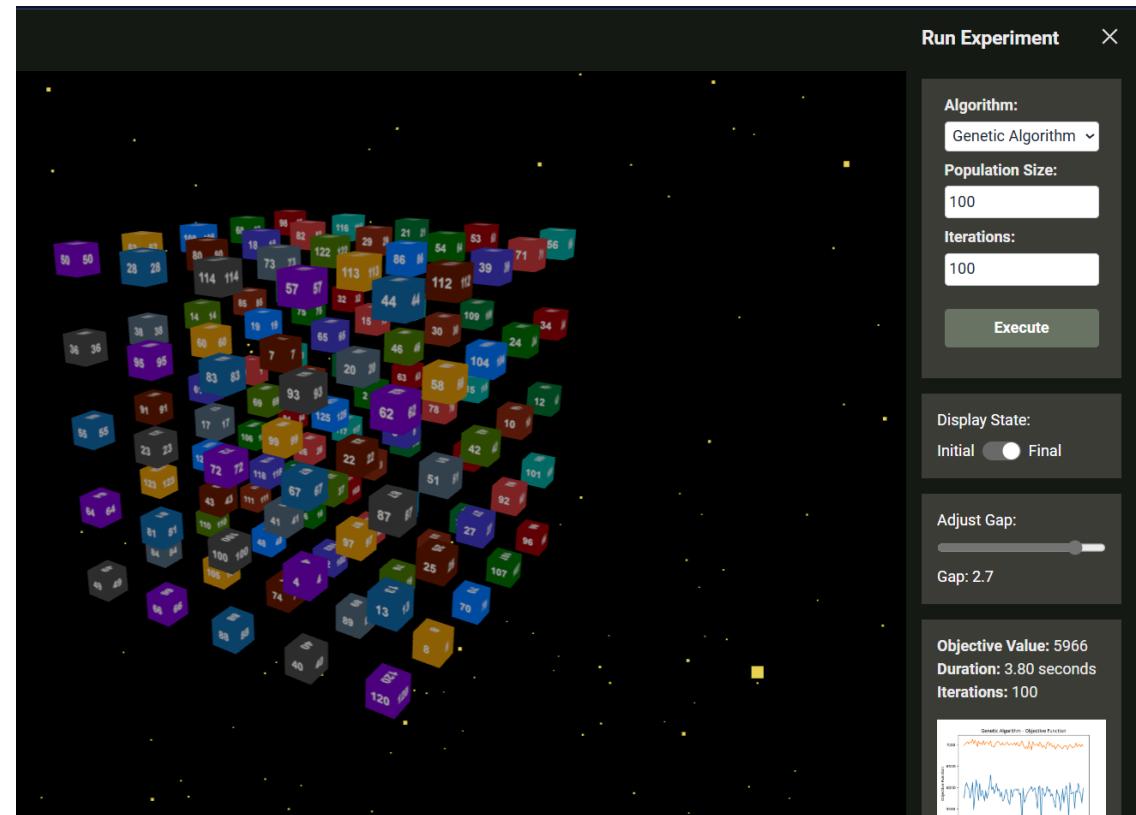


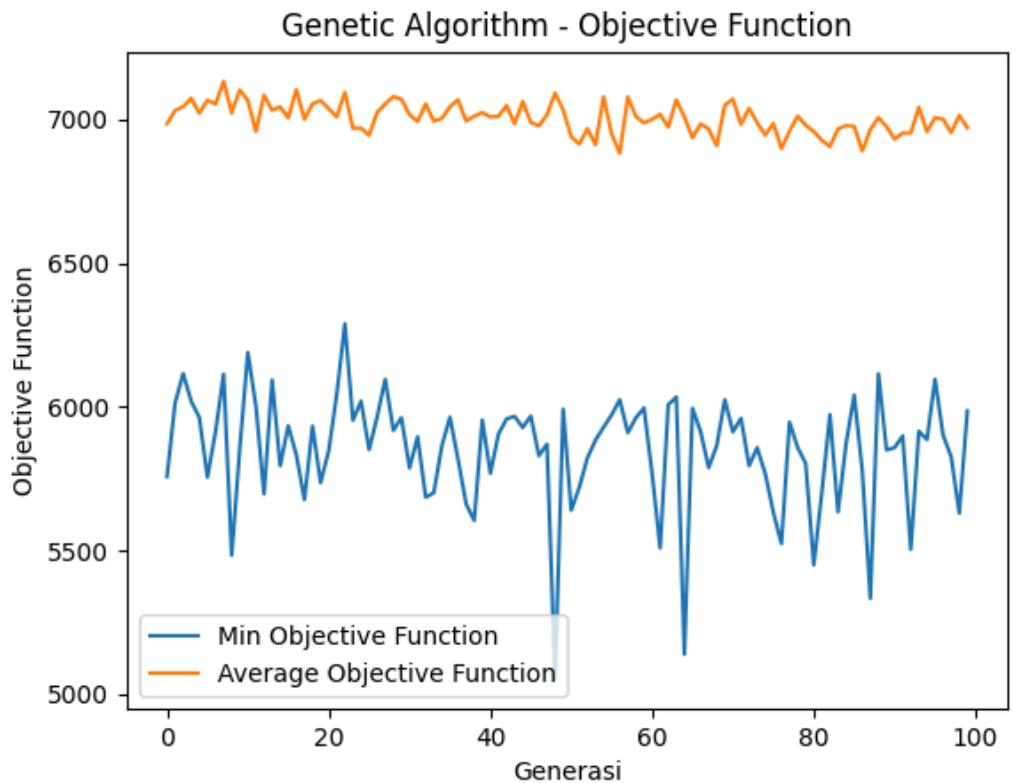
2.



Berdasarkan dari hasil eksperimen didapat :

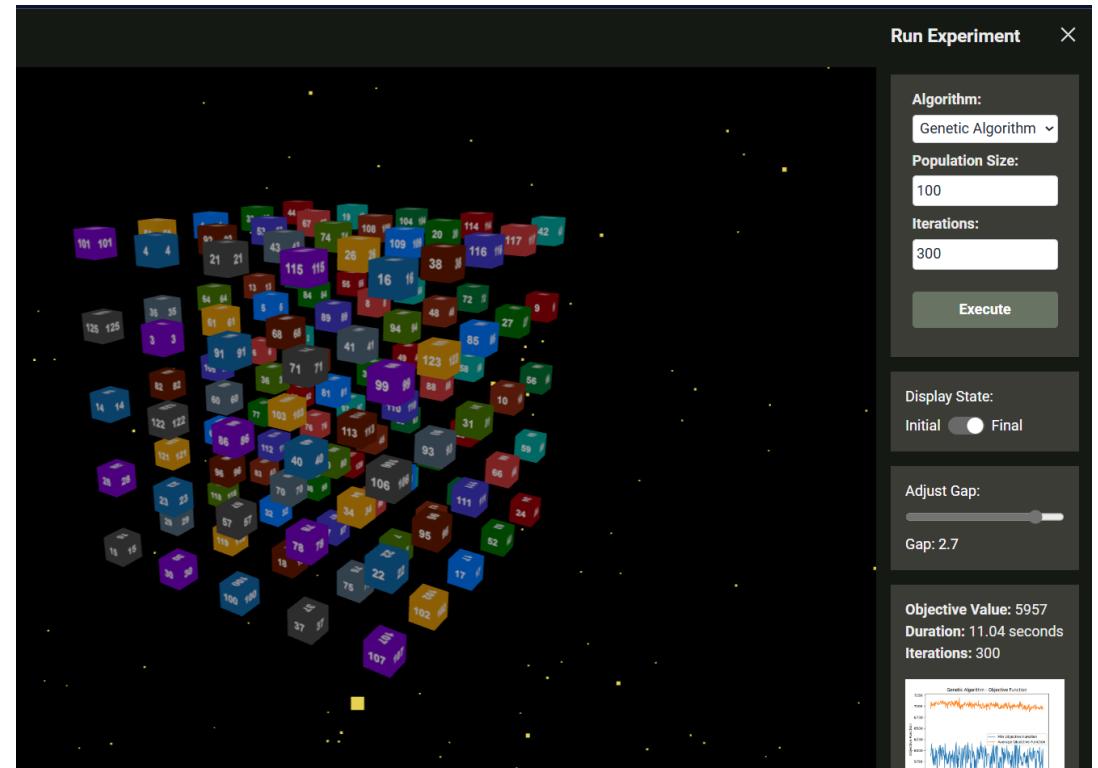
- Objektif Function akhir → 5946
- Durasi → 3.95s



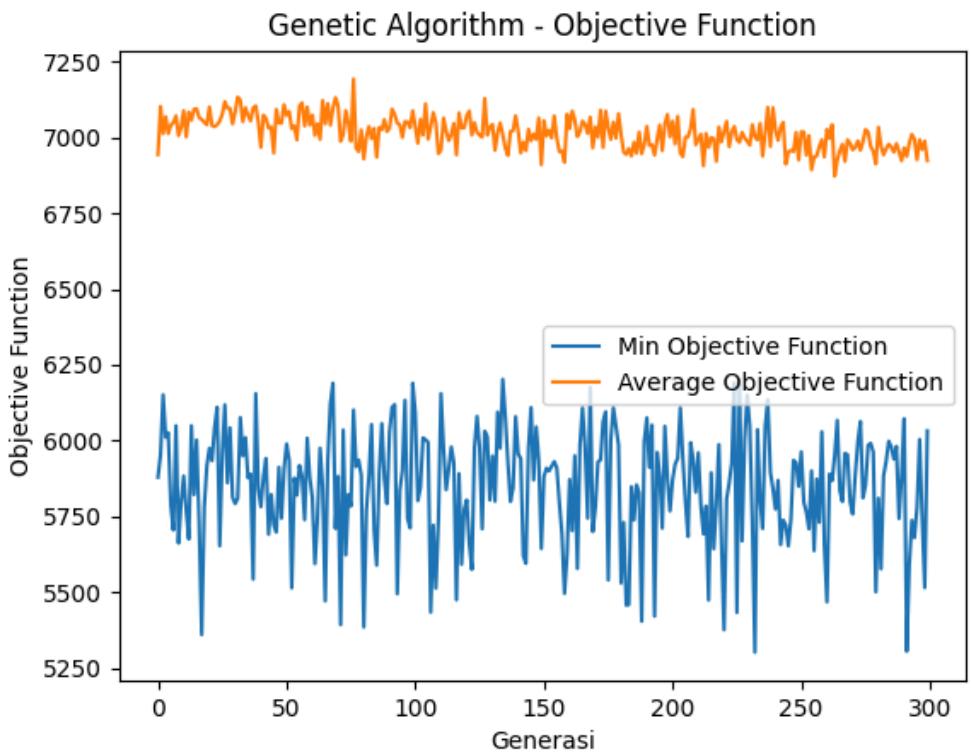


Berdasarkan dari hasil eksperimen didapat :

- Objektif Function akhir  $\rightarrow 5966$
- Durasi  $\rightarrow 3.80s$
- Iterasi : 300

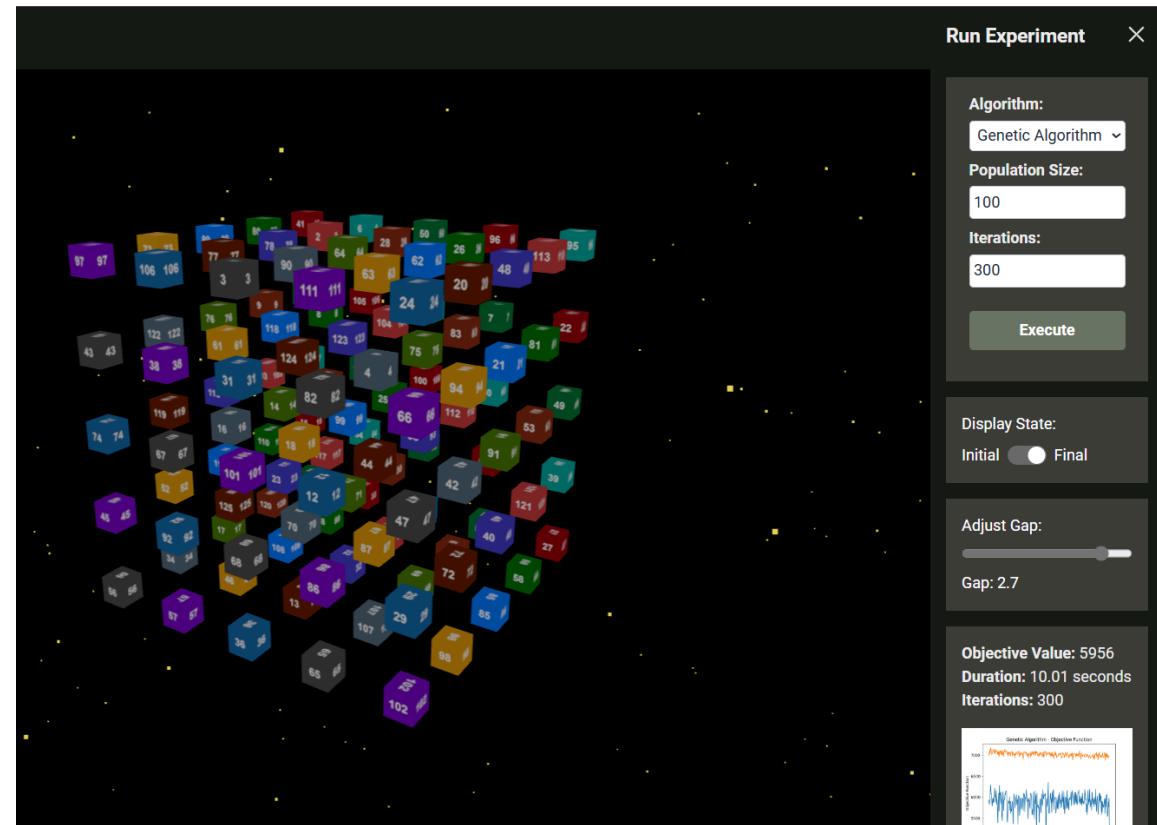


1.

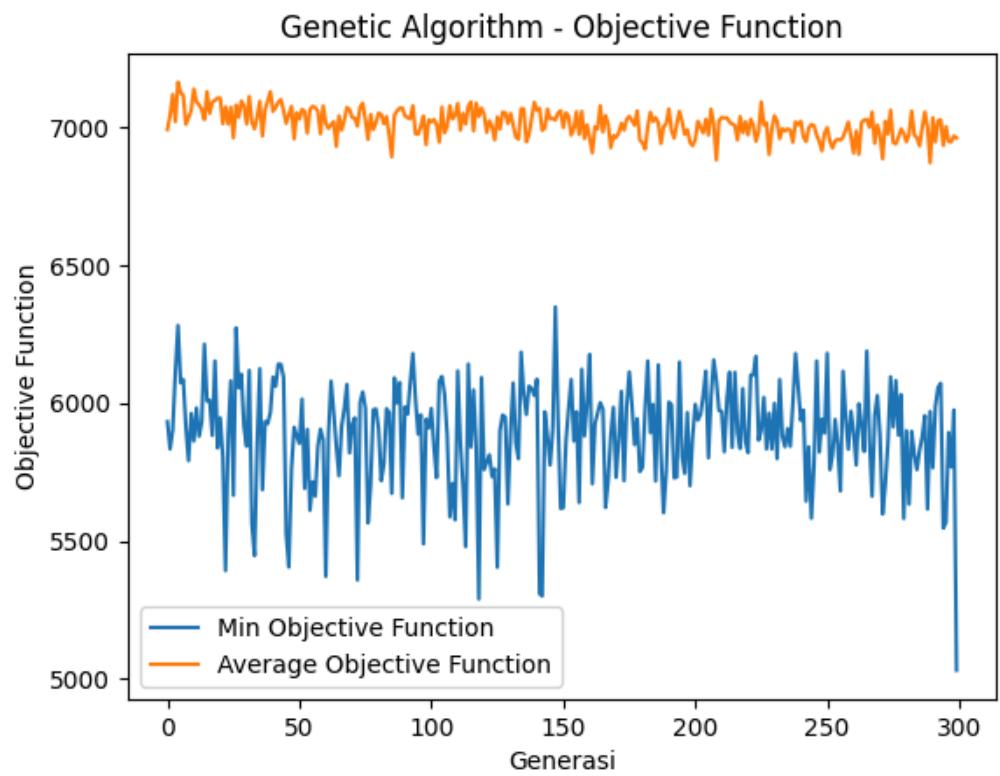


Berdasarkan dari hasil eksperimen didapat :

- a. Objektif Function akhir → 5957
- b. Durasi → 11.04s

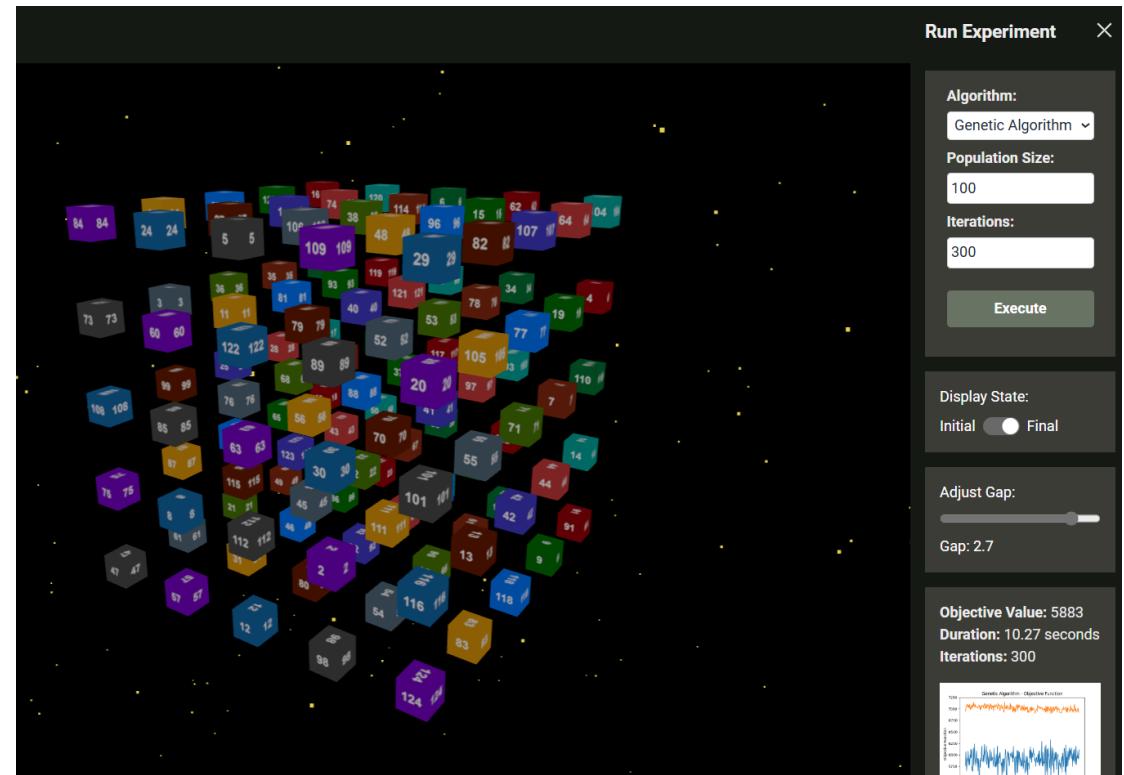


2.

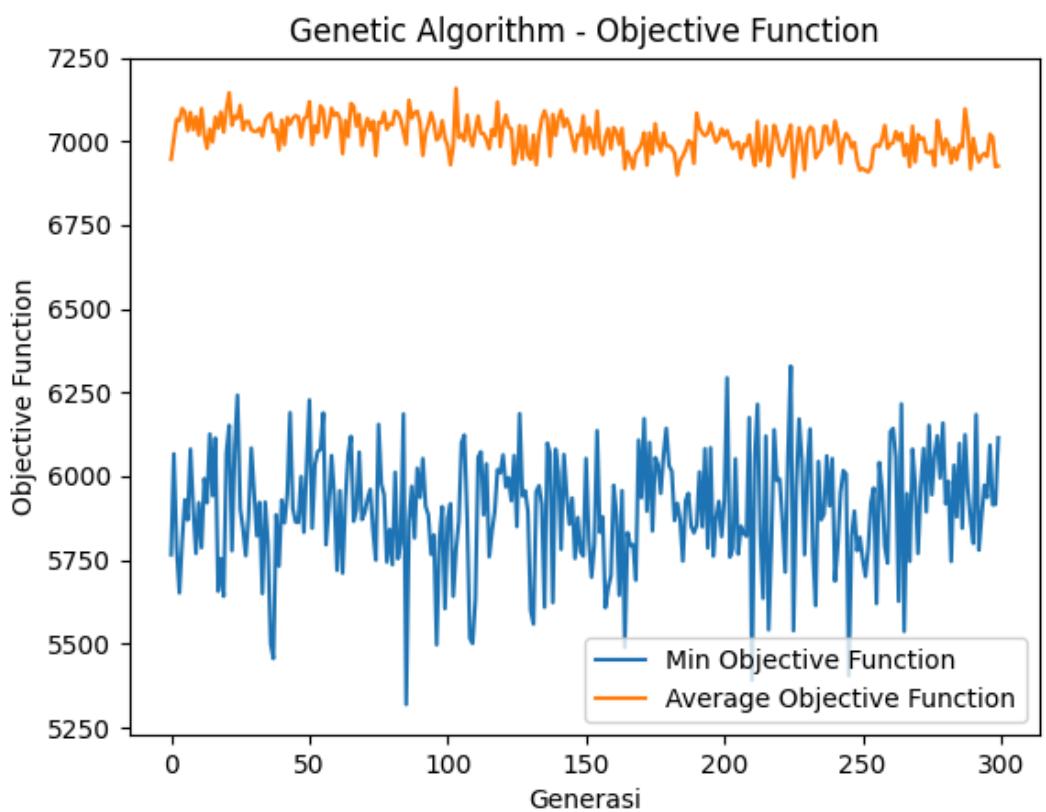


Berdasarkan dari hasil eksperimen didapat :

- Objektif Function akhir  $\rightarrow 5956$
- Durasi  $\rightarrow 10.01\text{s}$

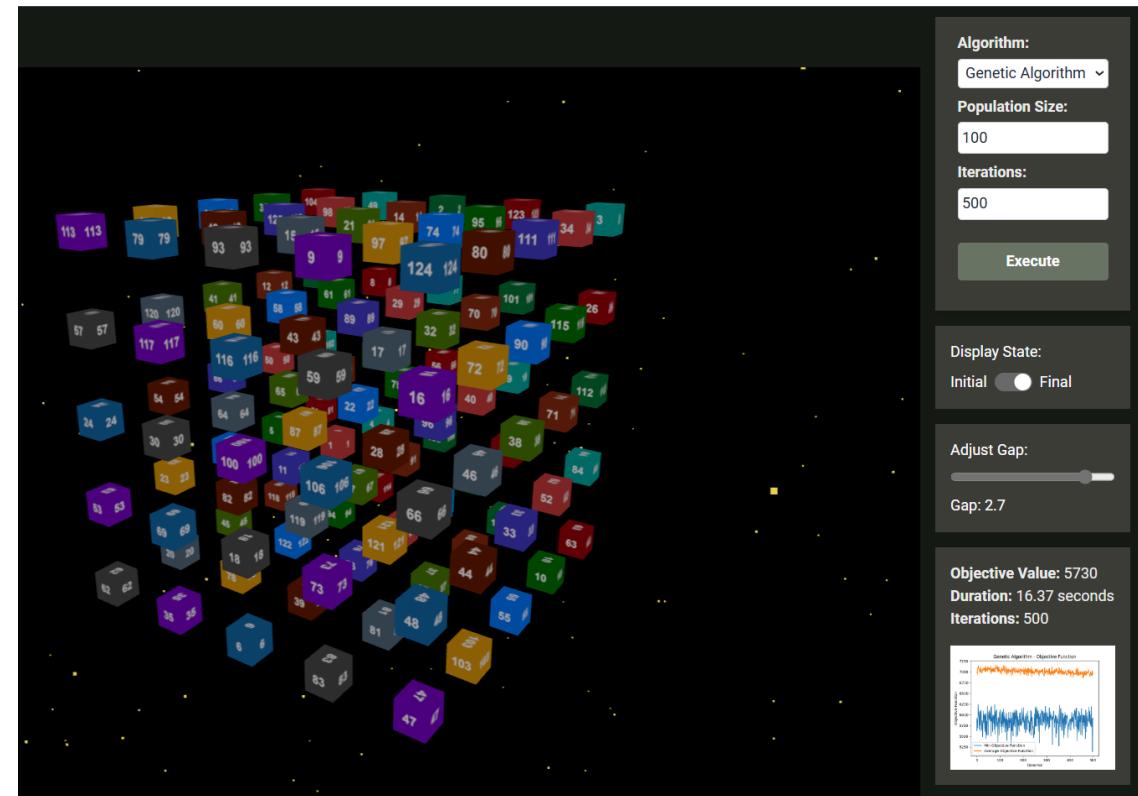


3.

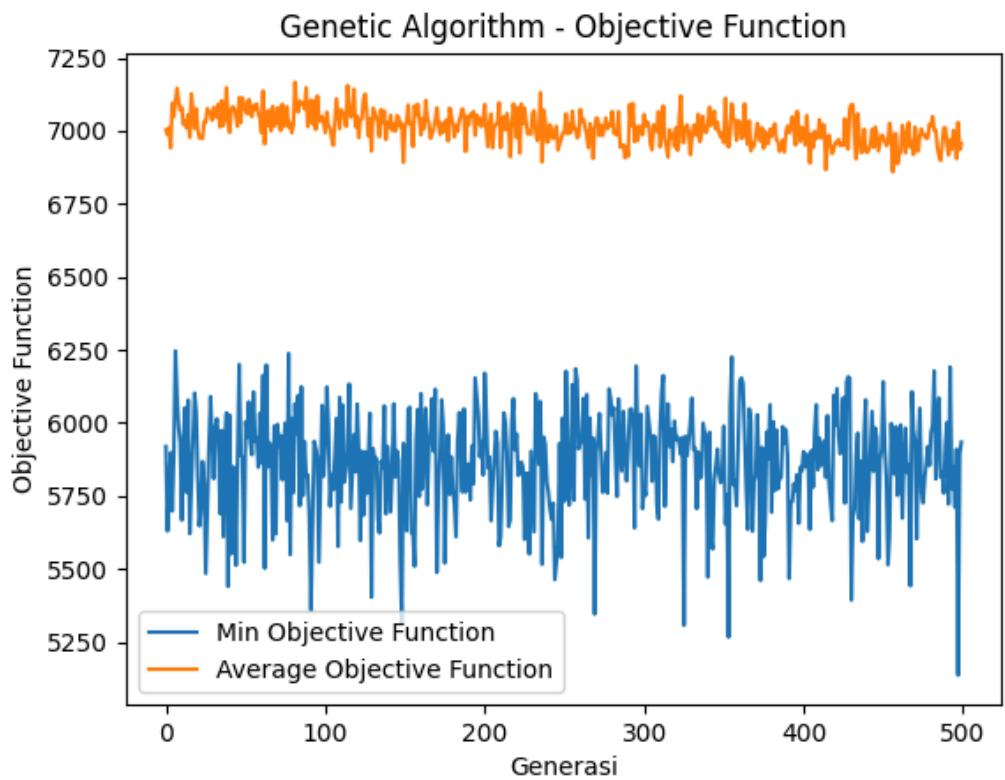


Berdasarkan dari hasil eksperimen didapat :

- a. Objektif Function akhir  $\rightarrow 5883$
  - b. Durasi  $\rightarrow 10.27s$
- o Iterasi : 500

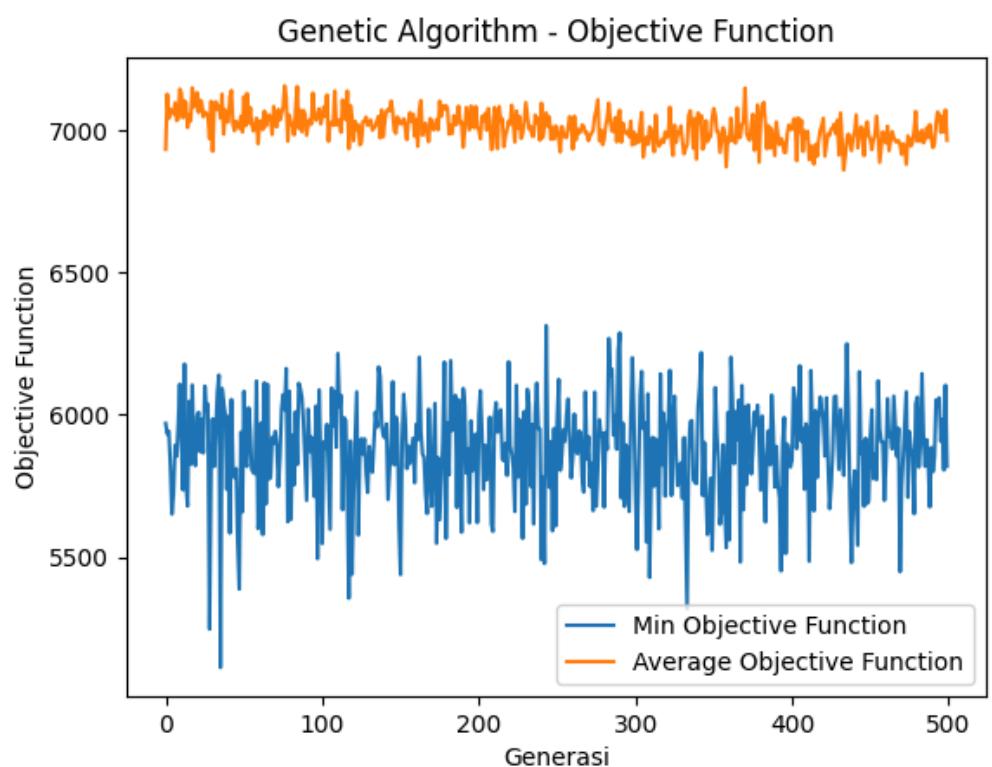
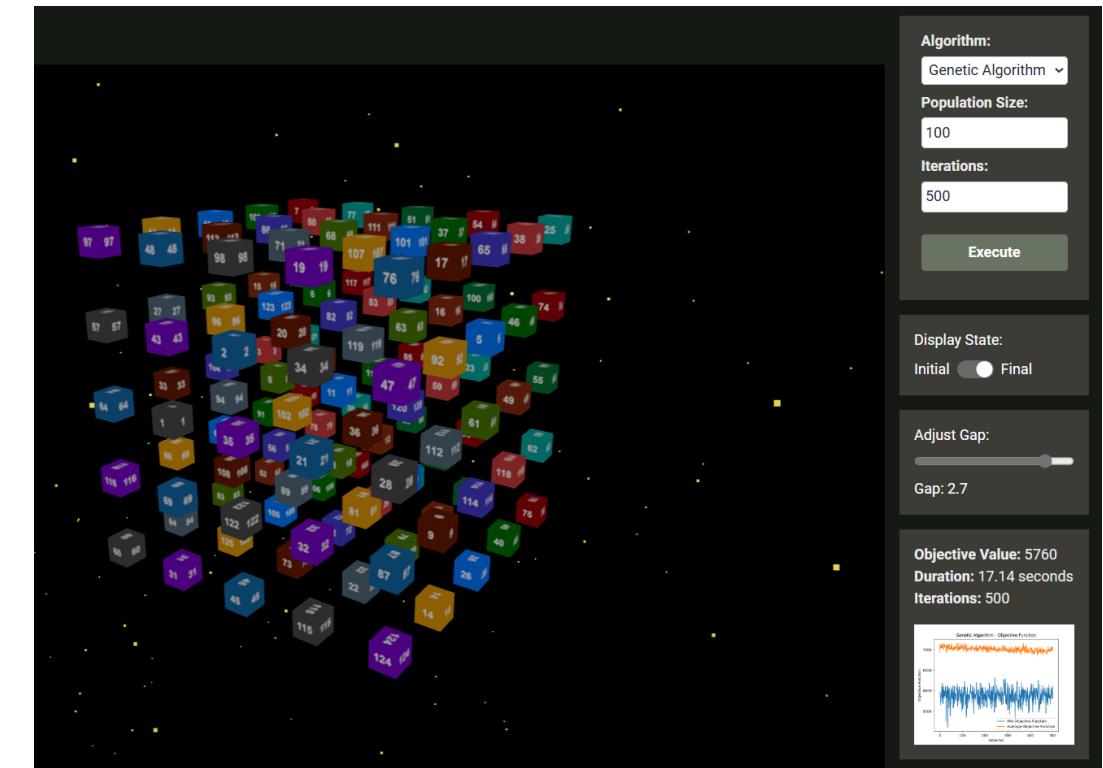


1.



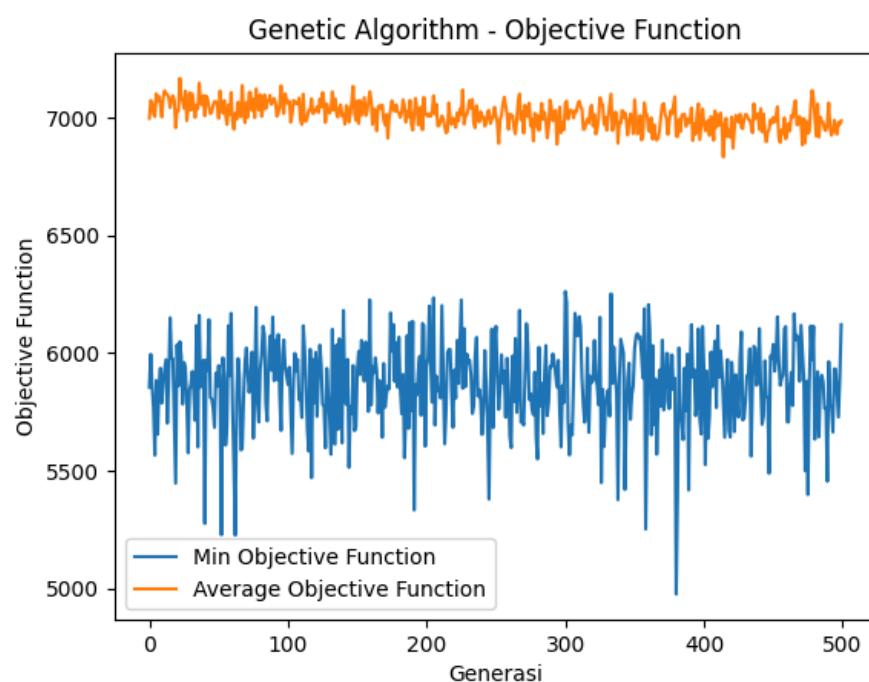
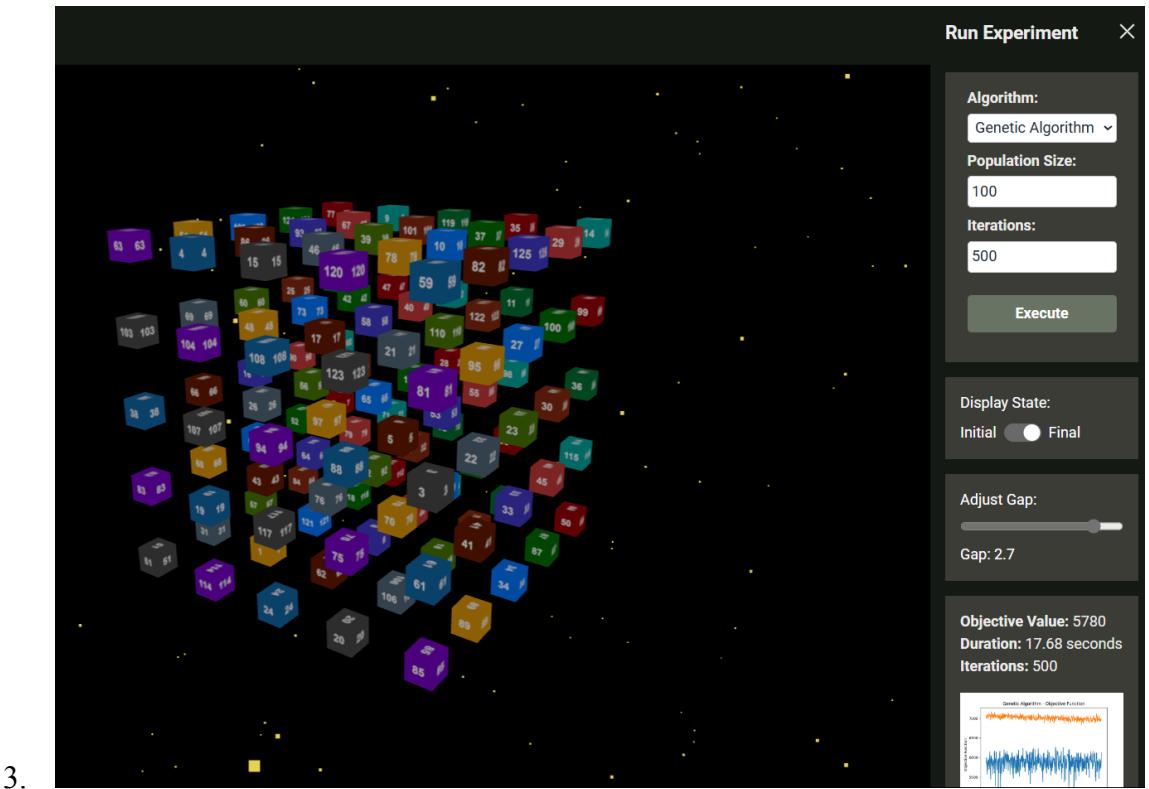
Berdasarkan dari hasil eksperimen didapat :

- a. Objektif Function akhir → 5730
- b. Durasi → 16.37s



Berdasarkan dari hasil eksperimen didapat :

- a. Objektif Function akhir → 5760
- b. Durasi → 17.14s

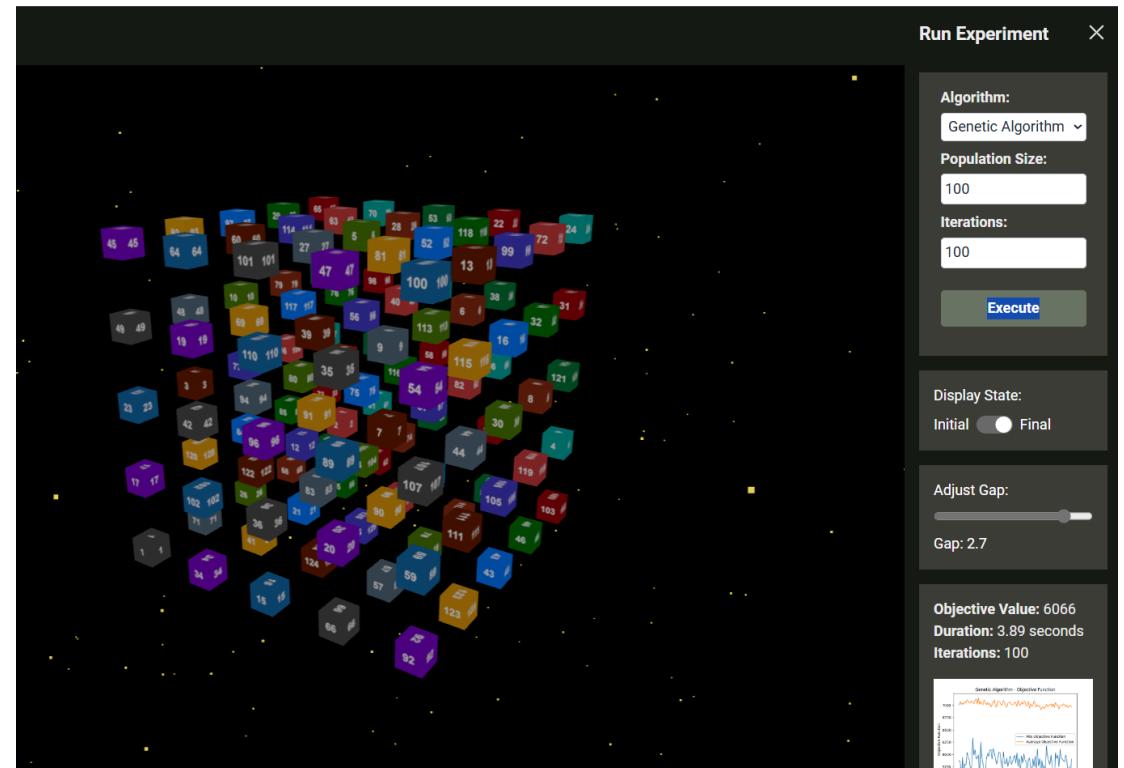


Berdasarkan dari hasil eksperimen didapat :

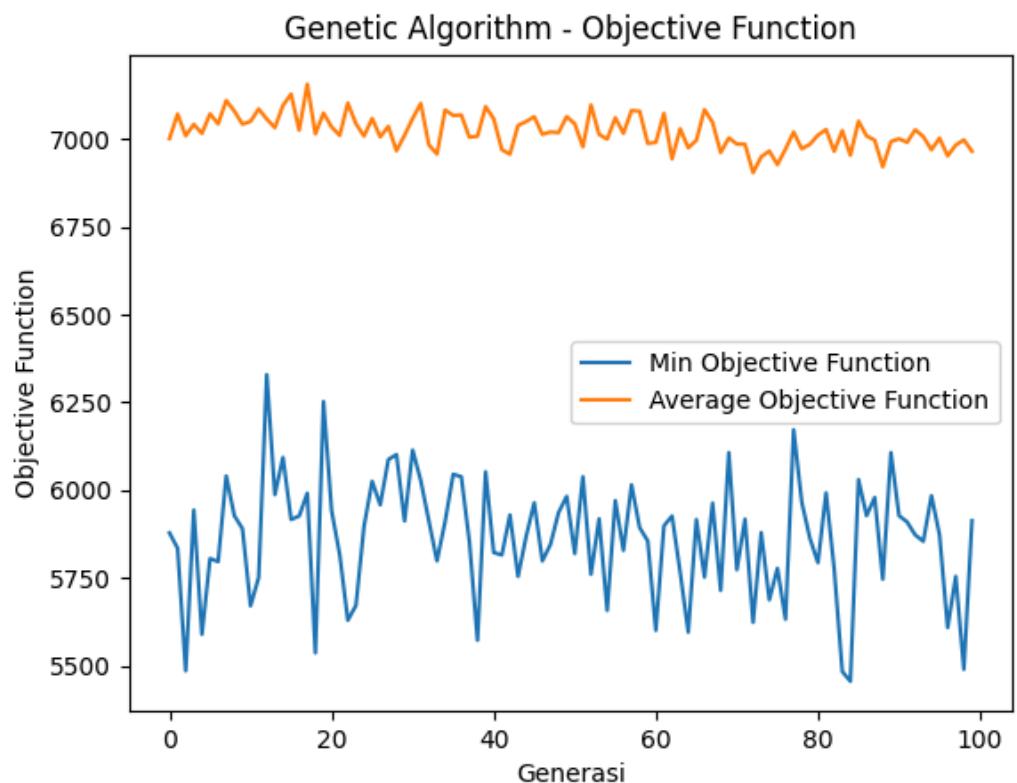
- a. Objektif Function akhir → 5780

b. Durasi → 17.68s

- Iterasi Sebagai kontrol (iterasi : 100)
  - n\_populasi : 100

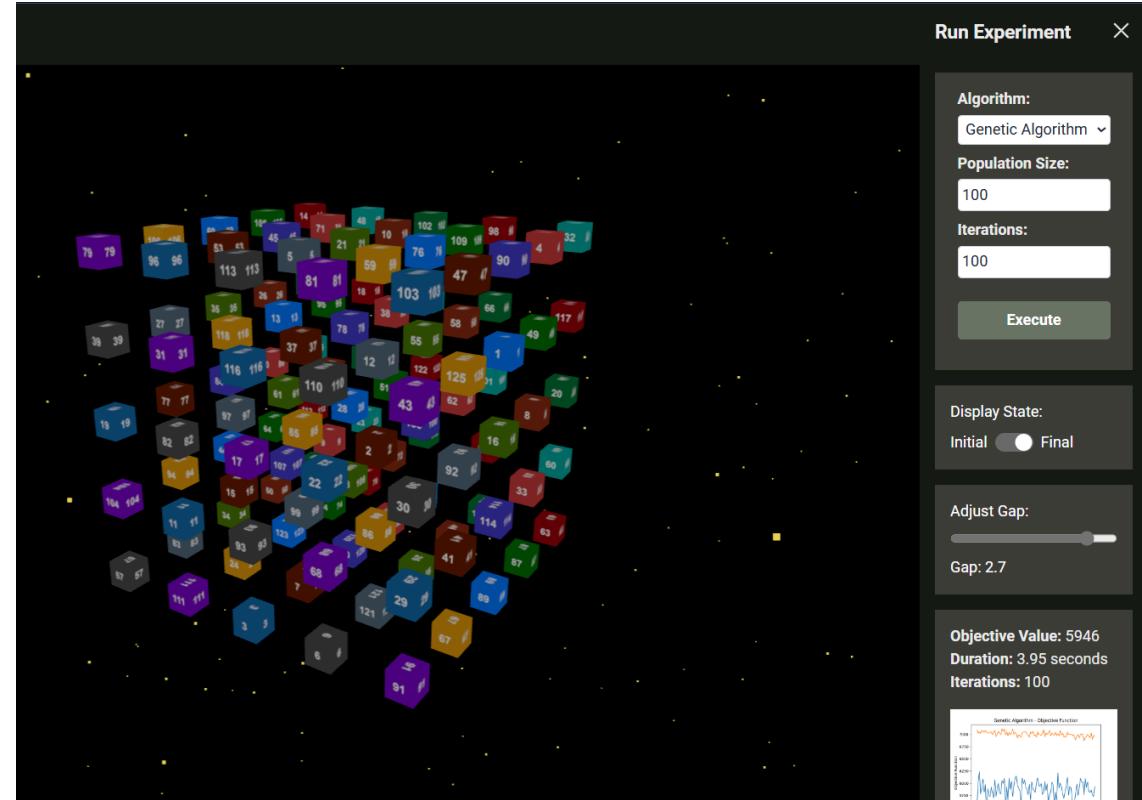


1.

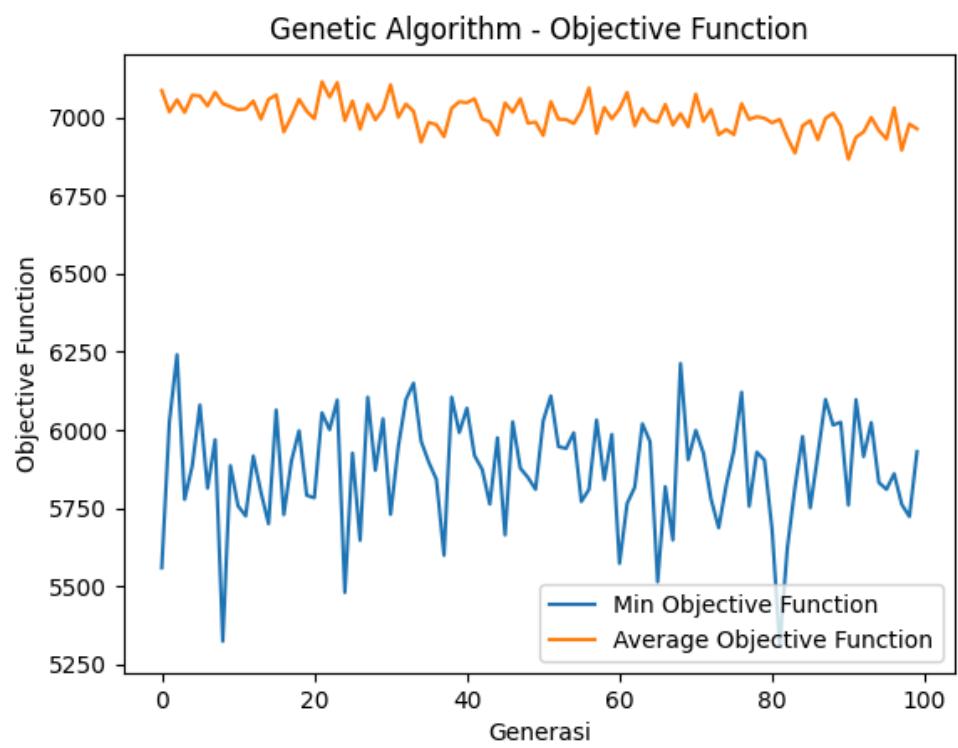


Berdasarkan dari hasil eksperimen didapat :

- Objektif Function akhir → 6066
- Durasi → 3.89s

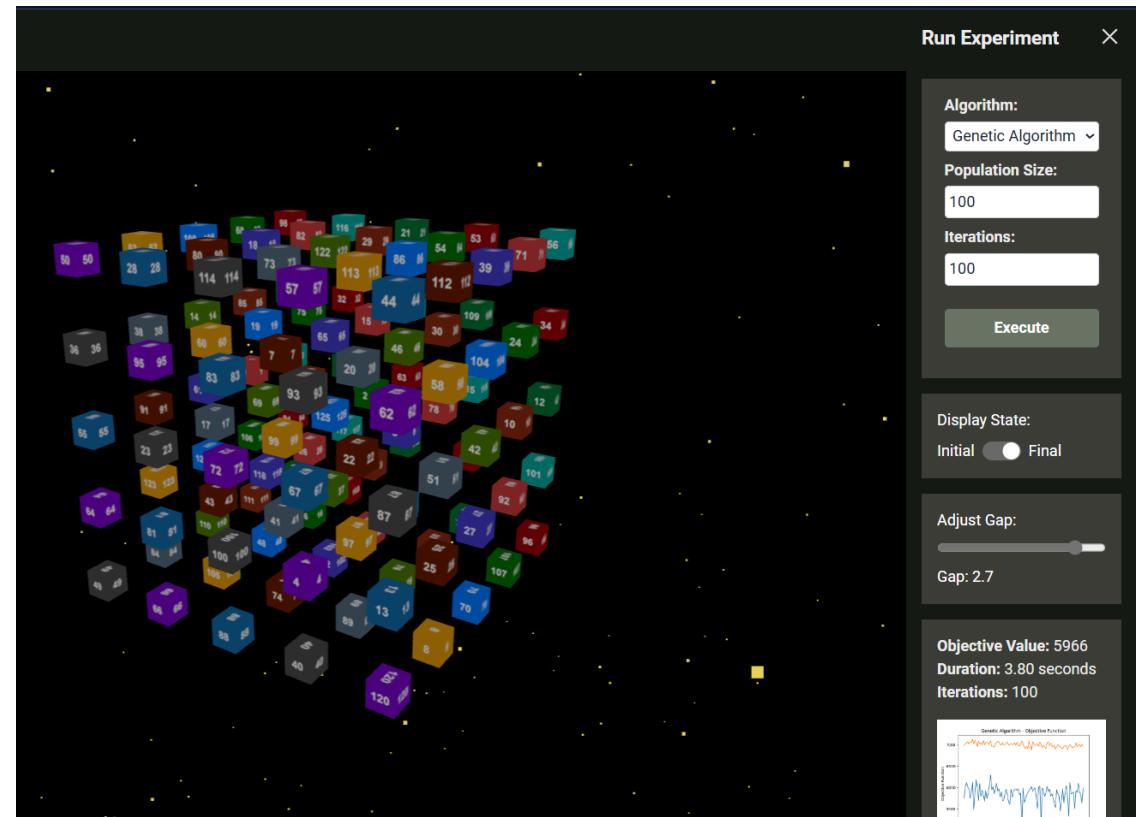


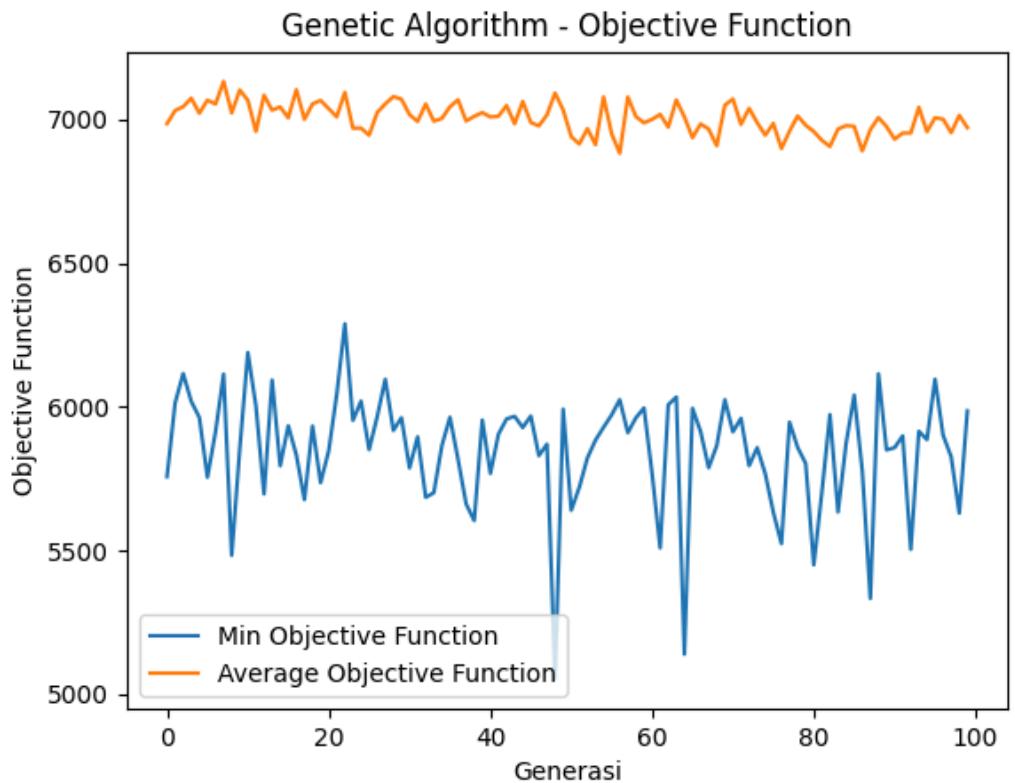
2.



Berdasarkan dari hasil eksperimen didapat :

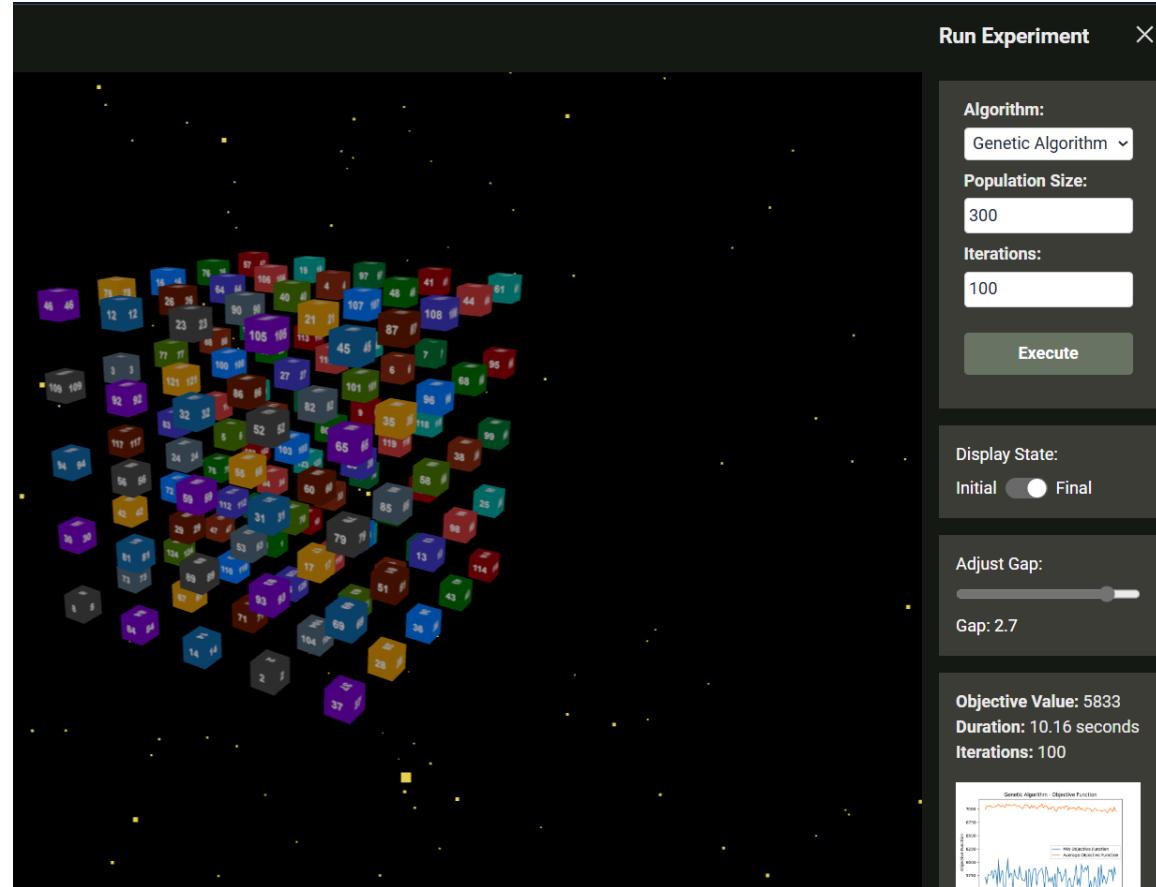
- Objektif Function akhir → 5946
- Durasi → 3.95s



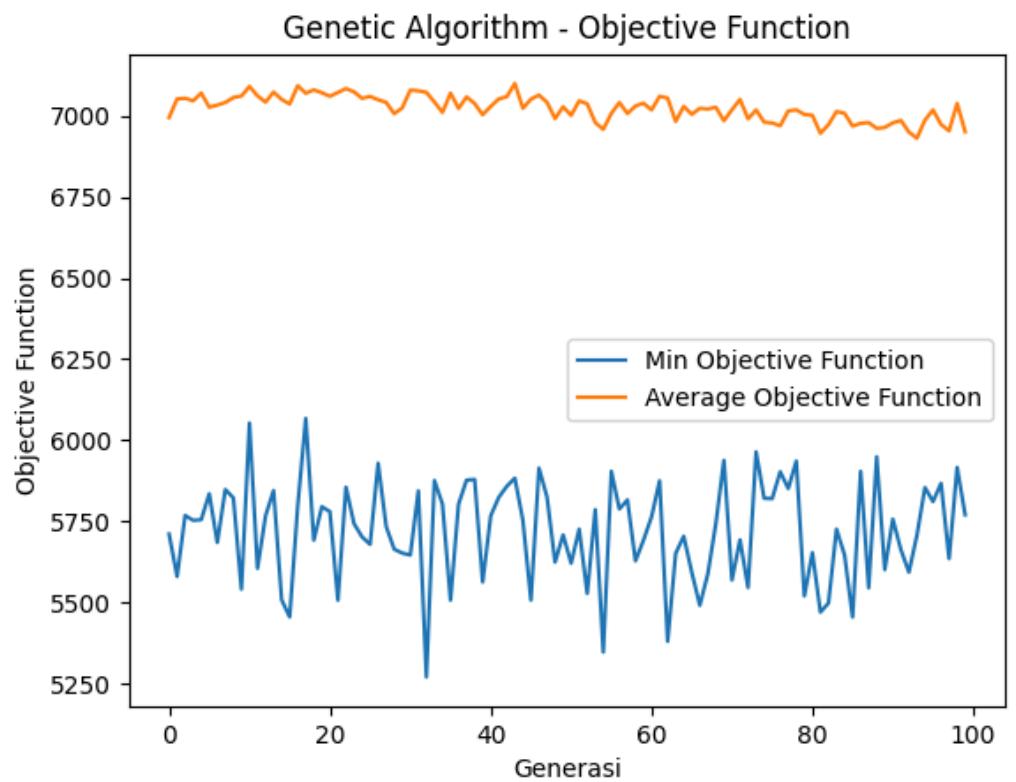


Berdasarkan dari hasil eksperimen didapat :

- a. Objektif Function akhir  $\rightarrow 5966$
- b. Durasi  $\rightarrow 3.80\text{s}$ 
  - o n\_populasi : 300

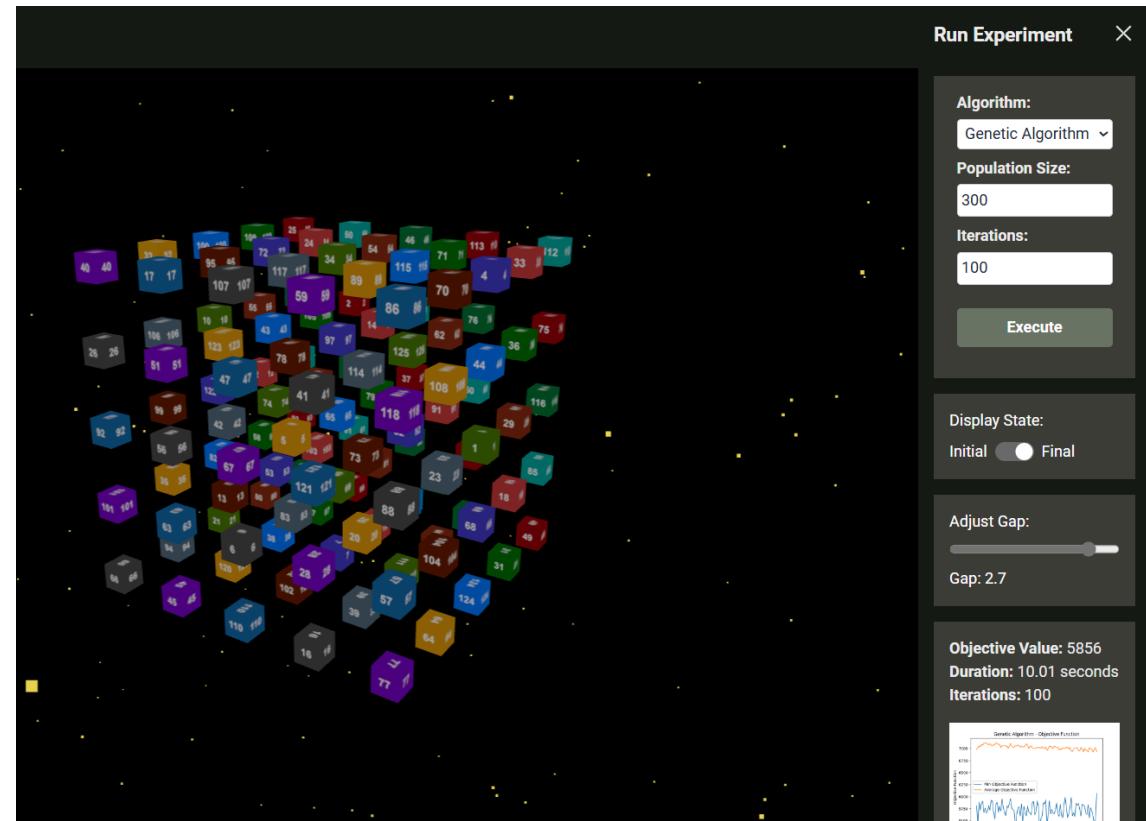


1.

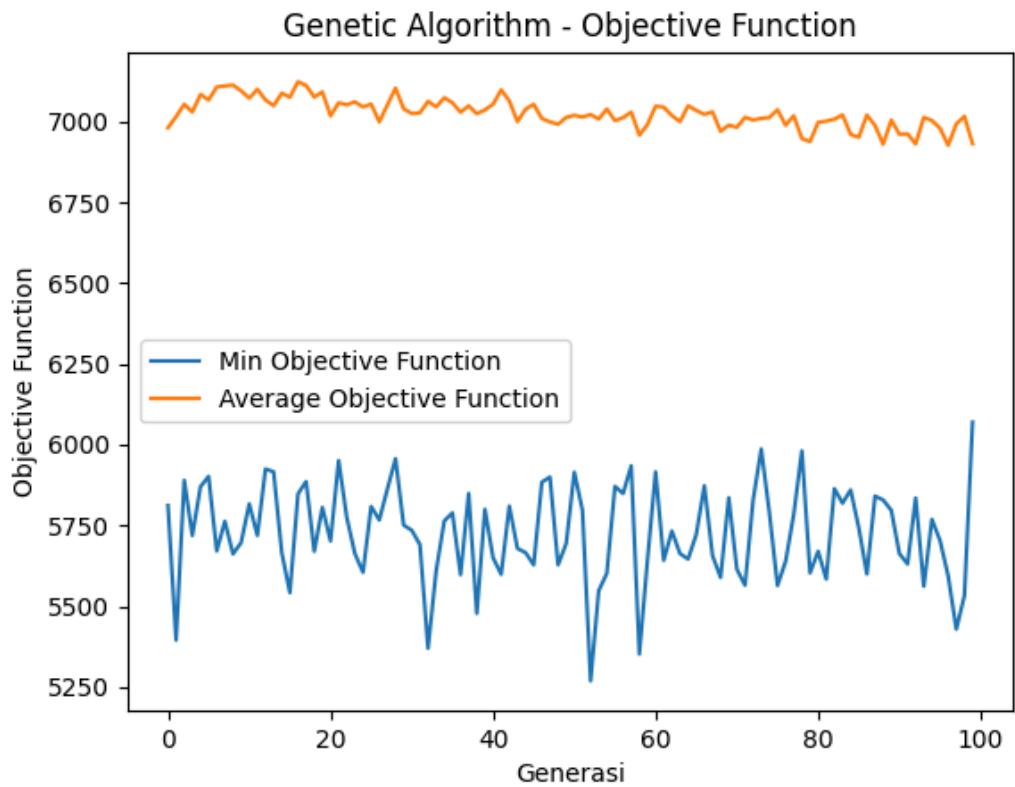


Berdasarkan dari hasil eksperimen didapat :

- Objektif Function akhir  $\rightarrow 5833$
- Durasi  $\rightarrow 10.16s$

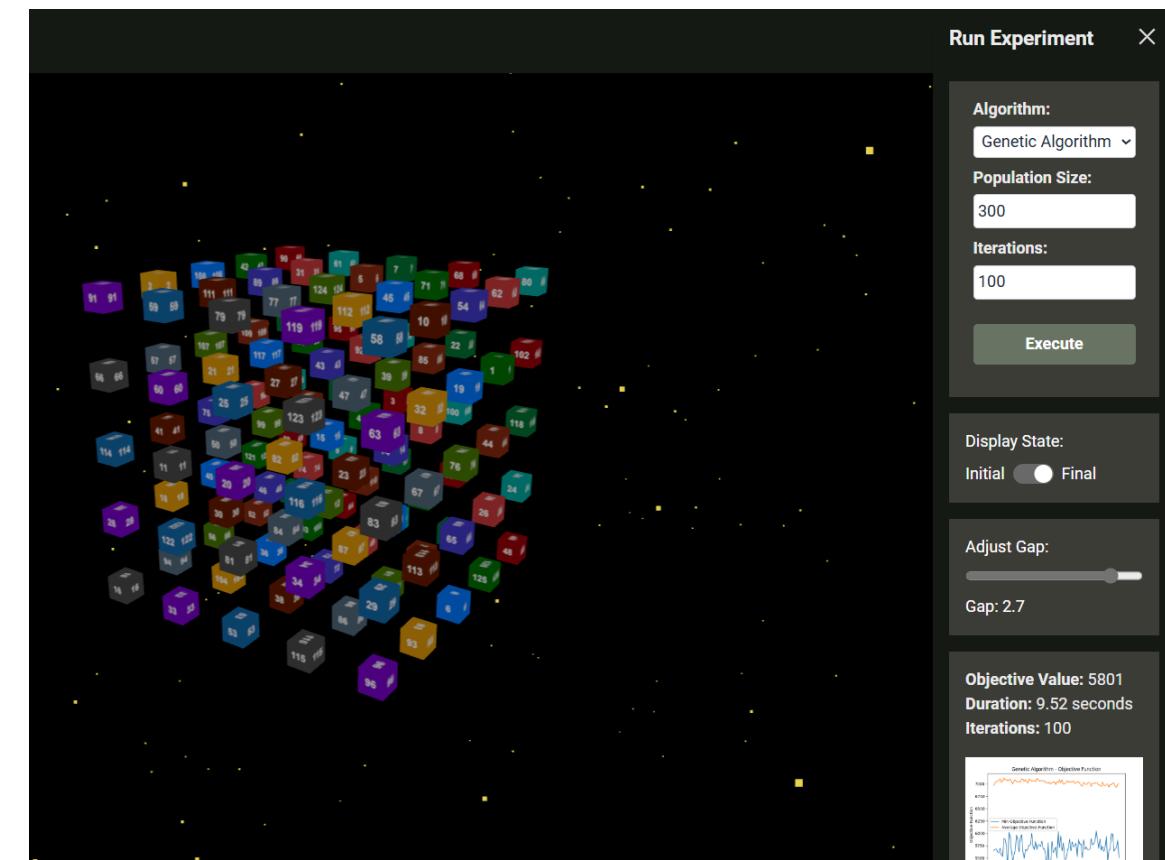


2.

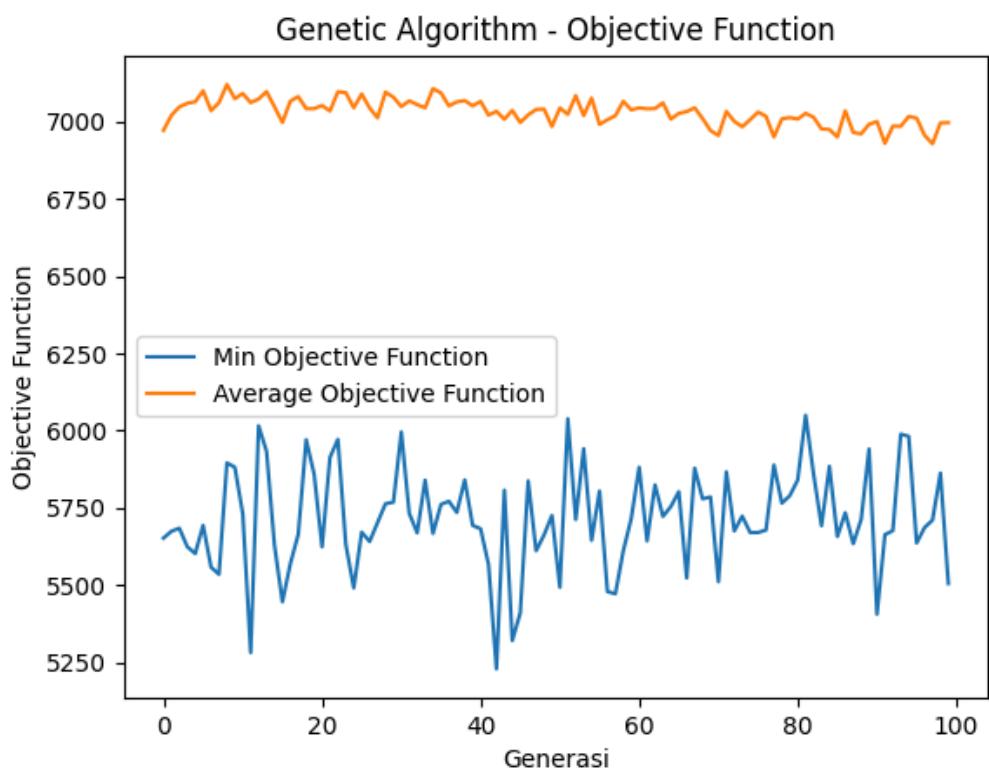


Berdasarkan dari hasil eksperimen didapat :

- a. Objektif Function akhir  $\rightarrow 5856$
- b. Durasi  $\rightarrow 10.01\text{s}$

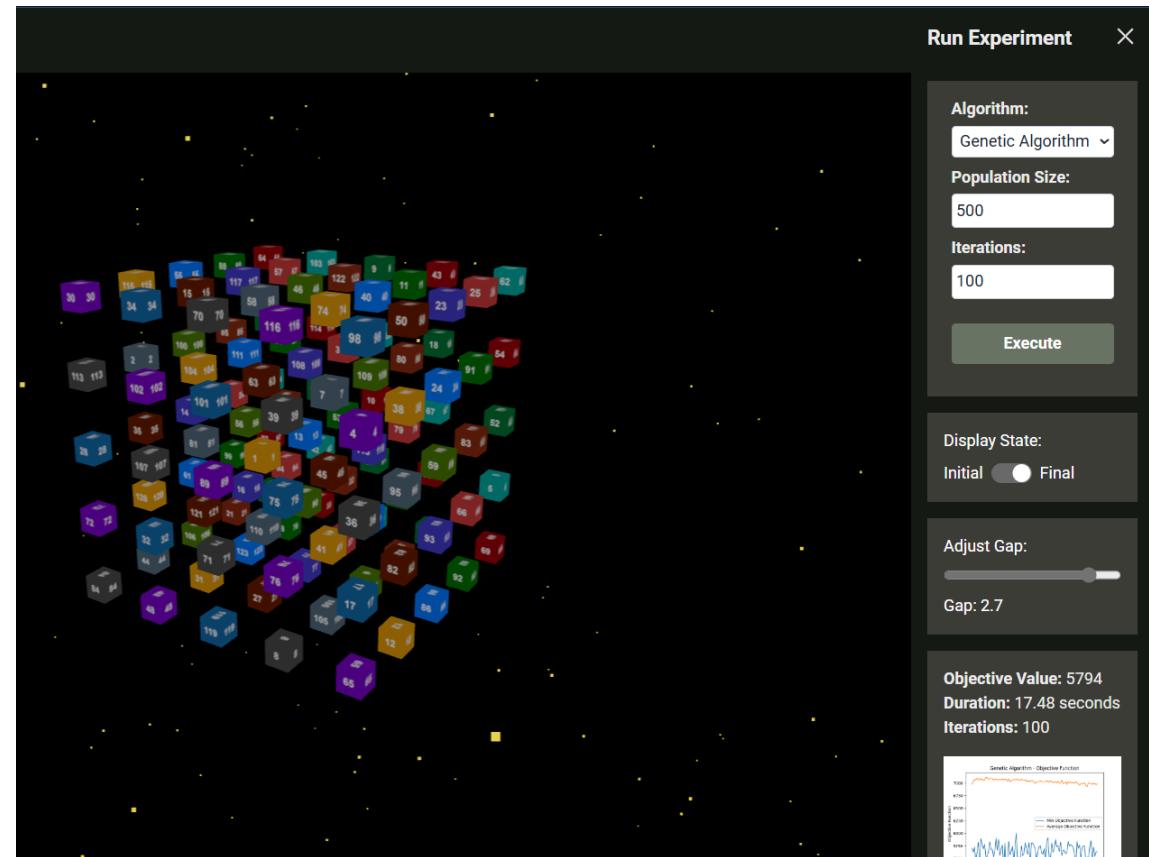


3.

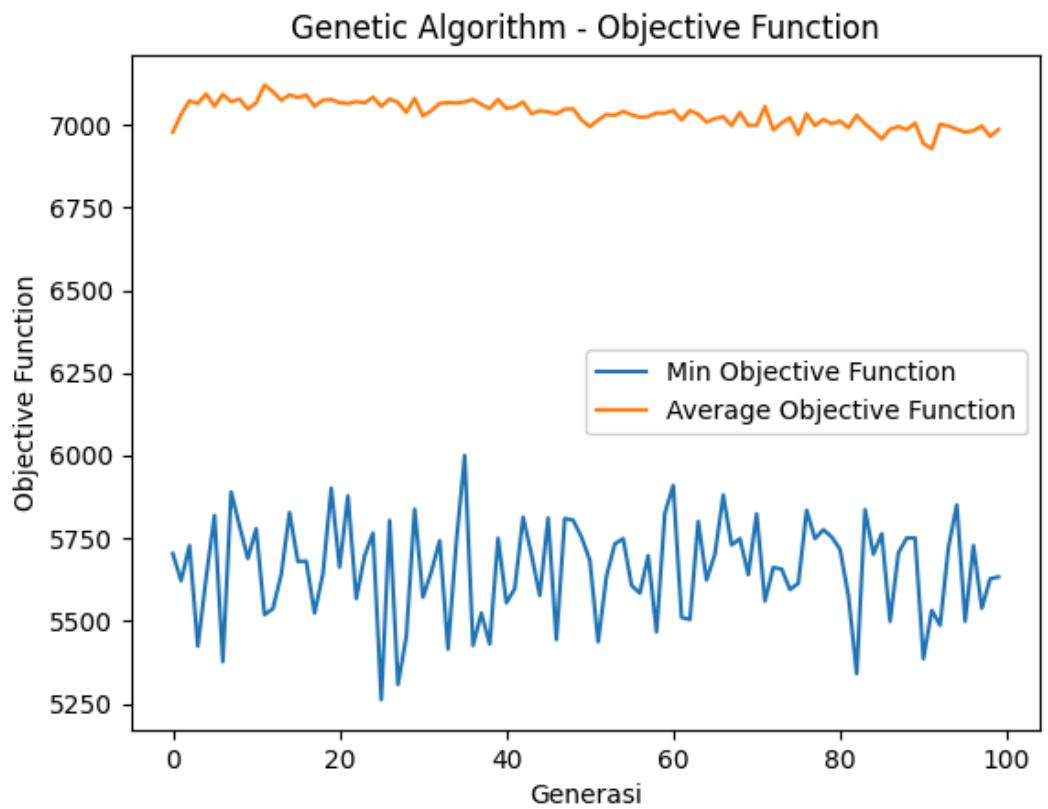


Berdasarkan dari hasil eksperimen didapat :

- c. Objektif Function akhir → 5801
- d. Durasi → 9.52s
- o n\_populasi : 500

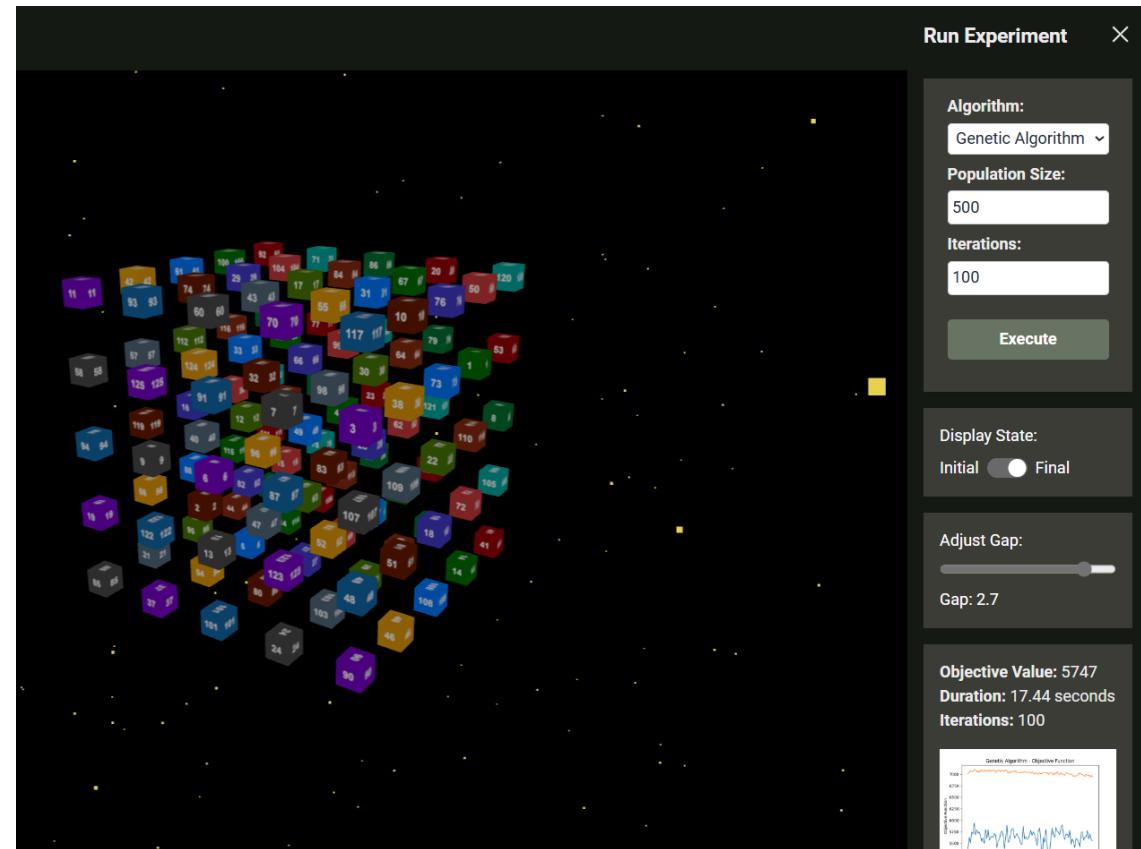


1.

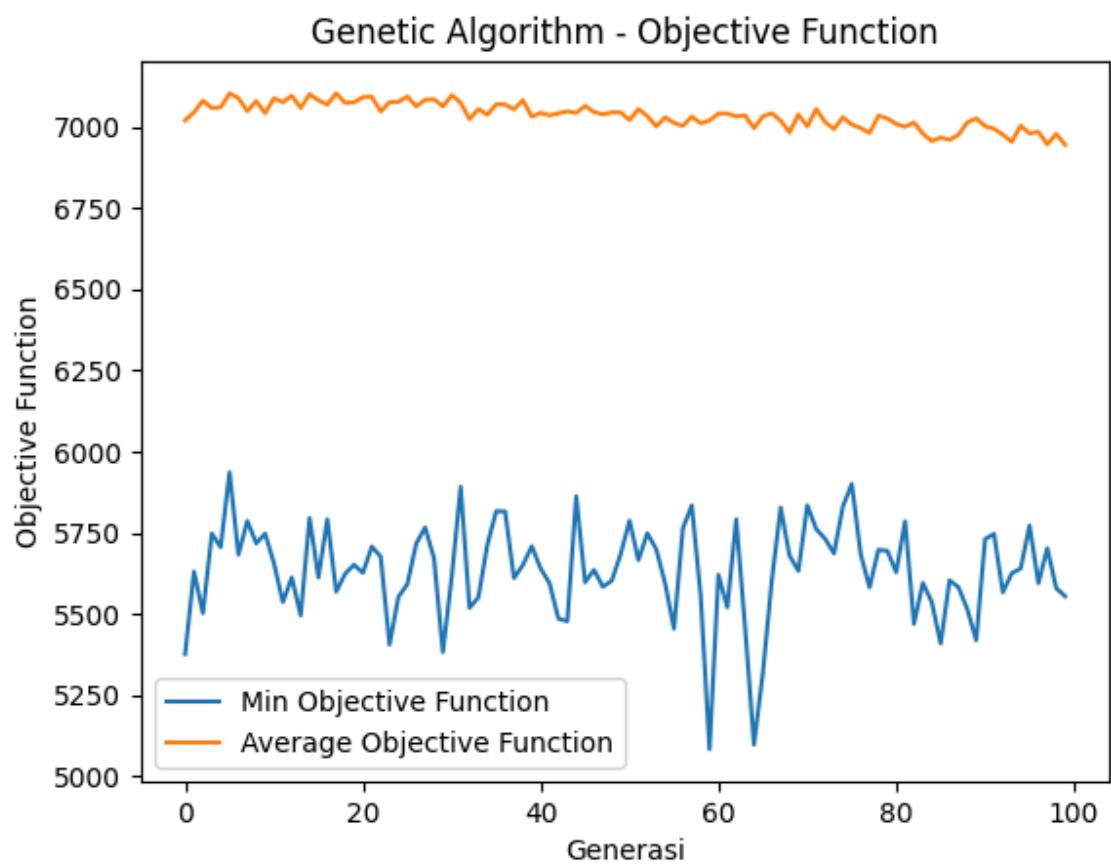


Berdasarkan dari hasil eksperimen didapat :

- Objektif Function akhir → 5794
- Durasi → 17.48

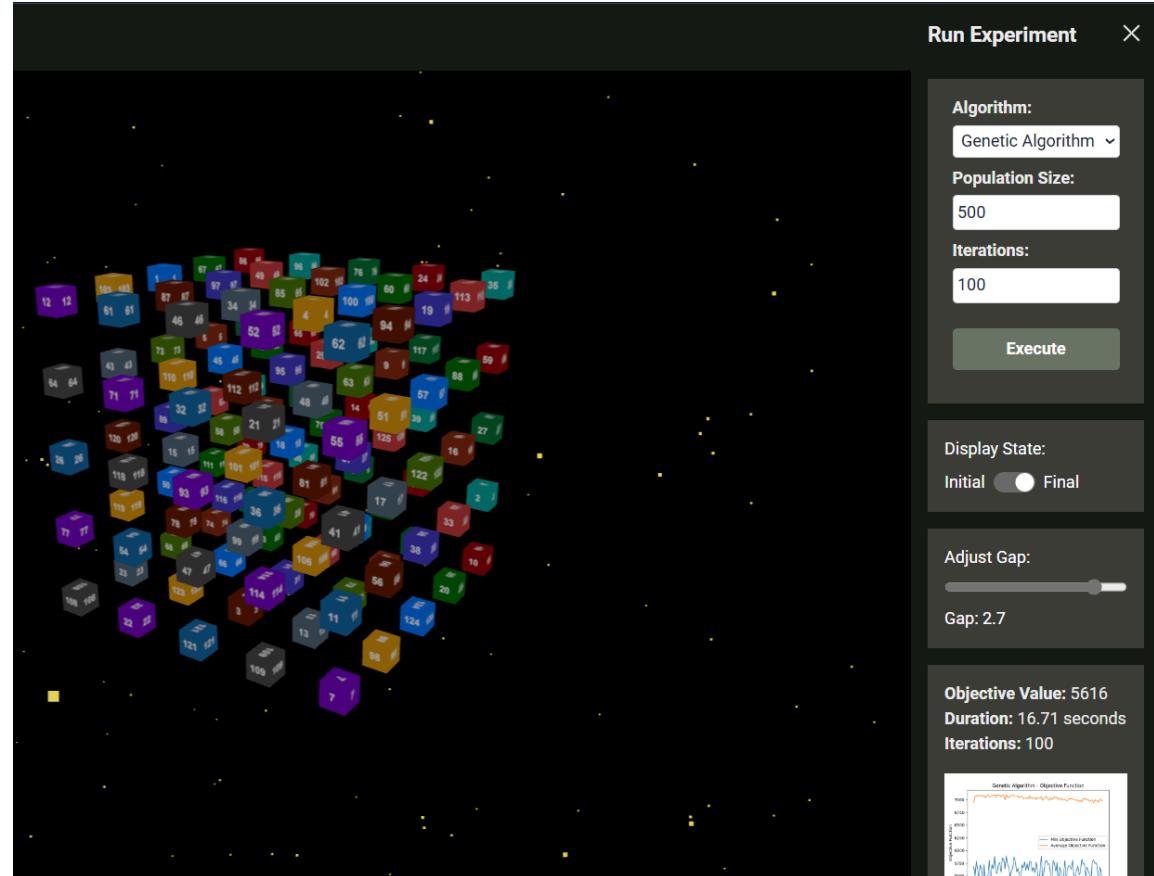


2.

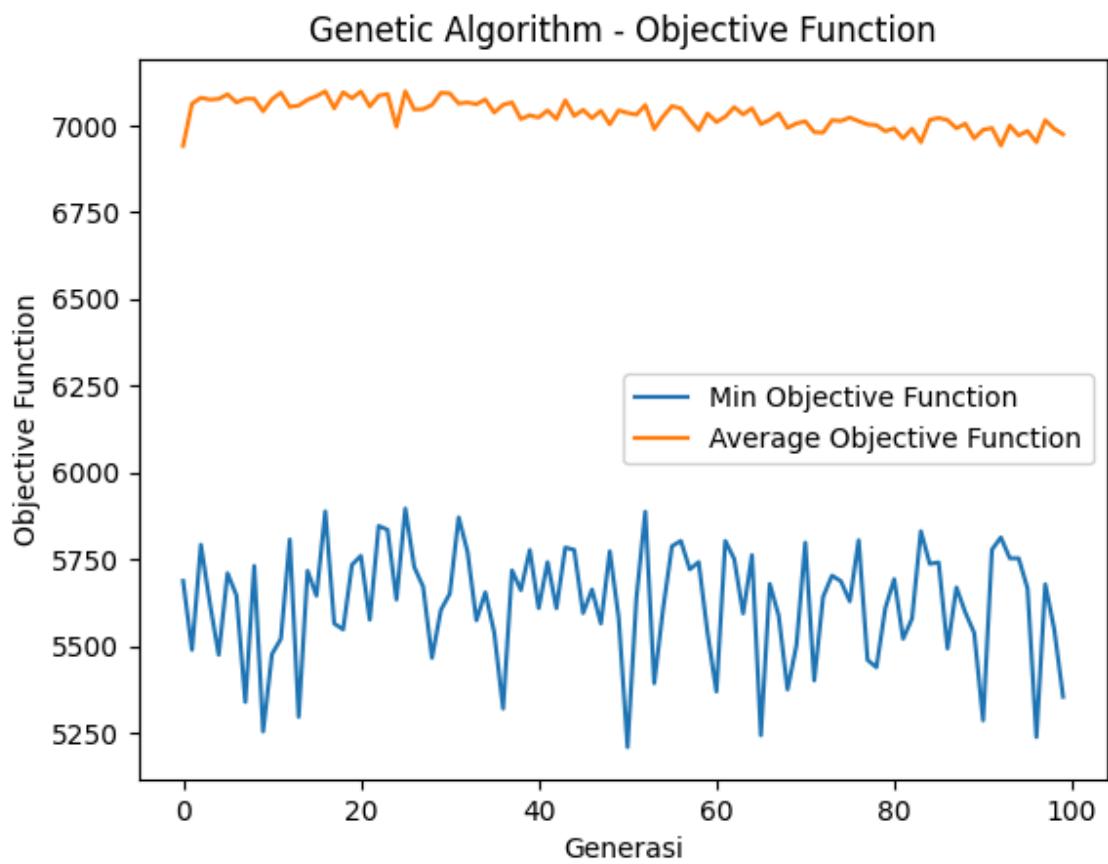


Berdasarkan dari hasil eksperimen didapat :

- Objektif Function akhir → 5747
- Durasi → 17.44



3.



Berdasarkan dari hasil eksperimen didapat :

- Objektif Function akhir → 5616
- Durasi → 16.71

Hasil eksperimen menunjukkan bahwa seiring meningkatnya jumlah iterasi dan ukuran populasi, nilai objective function pada genetic algorithm (GA) cenderung mendekati optimal. Namun, GA ini belum sepenuhnya mencapai solusi global optimal, melainkan hanya mendekati, yang dapat disebabkan oleh beberapa faktor.

- Keseimbangan antara eksplorasi dan eksloitasi sangat penting dalam GA; kurangnya keragaman dalam populasi atau jumlah iterasi yang terlalu sedikit dapat membuat algoritma ini terjebak pada local optima. Selain itu, laju mutasi yang terlalu rendah membatasi variasi solusi baru, sementara laju mutasi yang terlalu tinggi dapat mengganggu solusi yang sudah baik. Elitisme, yang mempertahankan individu terbaik

dalam setiap generasi, memang menjaga kualitas solusi tetapi bisa menurunkan keragaman populasi dan membatasi eksplorasi jika terlalu banyak diterapkan.

- Dibandingkan algoritma pencarian lokal seperti Hill Climbing dan Simulated Annealing, GA lebih fleksibel dalam menjelajahi ruang pencarian. Akan tetapi, berdasarkan dari hasil eksperimen ini Hill Climbing justru memberikan solusi terbaik berdasarkan objective function, begitu juga dengan Simulated Annealing yang memiliki mekanisme probabilistik "pendinginan" untuk menghindari local optima yang juga memberikan solusi yang lebih baik dibandingkan GA.
- Durasi proses pada GA cenderung lebih cepat dibandingkan algoritma hill climb yang mengevaluasi tiap tetangganya per iterasi sehingga GA akan lebih cepat karena proses per iterasi dari GA ini juga hanya mengevaluasi tiap populasinya. Akan tetapi, jika dibandingkan dengan simulated annealing GA cenderung lebih lambat karena komputasi dari simulated annealing pasti lebih cepat karena pengecekannya hanya 1 cube dan mengecek tetangga sesuai dengan probabilitasnya.
- Pengaruh jumlah iterasi dan populasi terhadap performa GA memberikan hasil dimana semakin banyak iterasi, semakin banyak waktu bagi GA untuk menemukan solusi yang lebih optimal karena proses evolusi berlangsung lebih lama. Demikian pula, populasi yang lebih besar meningkatkan kemungkinan menemukan solusi yang lebih baik karena lebih banyak individu yang berevolusi, dan kombinasi silang yang lebih bervariasi dapat terjadi. Namun, populasi besar juga meningkatkan durasi komputasi. Secara keseluruhan, GA menunjukkan hasil yang baik dalam mendekati optimal global, dengan iterasi yang lebih banyak membantu proses konvergensi, dan populasi yang lebih besar meningkatkan kualitas solusi meskipun dengan biaya komputasi yang lebih tinggi.

## **BAB 3**

### **KESIMPULAN DAN SARAN**

#### **3.1 Kesimpulan**

Dari hasil analisis eksperimen yang telah dilakukan, didapatkan kesimpulan bahwa algoritma Simulated annealing merupakan algoritma yang terbaik dalam mencari solusi yang optimal. algoritma ini terbukti efektif mendekati solusi global dan dengan waktu yang lebih singkat dibandingkan Steepest Ascent Hill-Climbing yang membutuhkan evaluasi neighbor yang lebih banyak per iterasi. Selain itu, Algoritma Hill-Climbing tidak memiliki mekanisme yang fleksibel dalam menghindari jebakan local optima. SA juga menunjukkan fleksibilitas dalam menemukan solusi optimal dibandingkan Genetic Algorithm (GA) dan Hill Climbing, meskipun hasilnya dapat bervariasi akibat mekanisme random successor. GA menunjukkan potensi dalam menjelajahi ruang pencarian secara fleksibel, terutama dengan dukungan variasi populasi dan iterasi yang lebih banyak. Namun, GA lebih lambat dari SA karena komputasi yang lebih intensif, khususnya dengan populasi yang besar, yang berdampak pada durasi proses yang panjang.

#### **3.1 Saran**

Saran pengembangan untuk tugas besar ini adalah :

1. Eksperimen Tambahan dengan Variasi Parameter

Meskipun setiap algoritma memiliki parameter yang khas (seperti laju mutasi pada Genetic Algorithm atau temperatur pada Simulated Annealing), penting untuk mengeksplorasi bagaimana variasi parameter ini mempengaruhi kinerja algoritma. Eksperimen lebih lanjut dengan variasi parameter yang lebih luas dapat membantu menemukan kombinasi optimal yang memberikan solusi terbaik dan meningkatkan adaptabilitas algoritma terhadap berbagai jenis permasalahan.

2. Pendekatan Hybrid untuk Meningkatkan Kinerja

Mengingat kekuatan dan kelemahan masing-masing algoritma local search, pendekatan hybrid dapat menjadi solusi yang efektif. Sebagai contoh, menggabungkan kemampuan

eksplorasi Genetic Algorithm dengan fokus eksploratif dari Hill Climbing atau pendekatan probabilistik Simulated Annealing dapat menciptakan algoritma yang lebih fleksibel dalam menangani berbagai jenis permasalahan dan menghindari jebakan local optima.

### 3. Eksperimen pada Skala Permasalahan yang Berbeda

Untuk lebih memahami skalabilitas algoritma local search, eksperimen pada skala permasalahan yang berbeda (misalnya, ukuran kubus atau ruang pencarian yang lebih besar atau lebih kecil) sangat dianjurkan. Ini akan membantu mengidentifikasi batas-batas efektivitas algoritma serta memberikan wawasan tentang kompleksitas komputasi dan adaptabilitas terhadap perubahan skala masalah.

### 4. Penggunaan Algoritma Metaheuristik Lain sebagai Alternatif

Selain algoritma yang telah diuji, ada banyak algoritma metaheuristik lain yang juga populer, seperti Particle Swarm Optimization, Ant Colony Optimization, dan Tabu Search. Menerapkan dan menguji algoritma-algoritma ini dapat memberikan perspektif baru tentang metode pencarian solusi yang lebih efisien dan mungkin menghasilkan hasil yang lebih baik dibandingkan algoritma yang telah digunakan dalam eksperimen.

### 5. Optimalisasi Komputasi melalui Teknik Pemrograman Paralel

Beberapa algoritma local search, terutama yang memerlukan evaluasi banyak solusi (seperti Genetic Algorithm dengan populasi besar), bisa sangat intensif dalam hal komputasi. Optimalisasi kinerja komputasi melalui pemrograman paralel atau distribusi tugas pada beberapa prosesor dapat mengurangi waktu pemrosesan dan memungkinkan eksperimen dengan jumlah iterasi dan ukuran populasi yang lebih besar tanpa membebani sistem.

### 6. Analisis Konsistensi dan Stabilitas Solusi

Mengingat variabilitas hasil pada algoritma seperti Simulated Annealing atau pendekatan probabilistik lainnya, eksperimen lanjutan untuk memahami faktor-faktor yang mempengaruhi konsistensi hasil sangat disarankan. Dengan lebih memahami faktor-faktor ini, parameter atau metode seleksi dalam algoritma dapat disesuaikan untuk meningkatkan stabilitas hasil, sehingga menghasilkan solusi yang lebih dapat diprediksi.

## **LAMPIRAN**

Tautan repository GitHub: <https://github.com/sibobbbbb/tubes-1-inteligensi-artifisial>

## DAFTAR PUSTAKA

- [Features of the magic cube - Magisch vierkant](#)
- [Perfect Magic Cubes \(trump.de\)](#)
- [Magic cube - Wikipedia](#)