

Tugas Besar 1 IF3170 Inteligensi Artifisial

Pencarian Solusi Diagonal Magic Cube dengan Local

Search



Disusun oleh:

Denise Felicia Tiowanni	(13522013)
Amalia Putri	(13522042)
Angelica Kierra Ninta Gurning	(13522048)
Immanuel Sebastian Girsang	(13522058)
Muhammad Naufal Aulia	(13522074)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2024

DAFTAR ISI

DAFTAR ISI.....	0
BAB I.....	2
DESKRIPSI PERSOALAN.....	2
1.1 Deskripsi Tugas.....	2
BAB II.....	4
PEMBAHASAN.....	4
2.1 Pemilihan Objective Function.....	4
2.2 Penjelasan Implementasi Algoritma Local Search.....	6
2.2.1 Fungsi/Kelas Umum.....	6
2.2.2 Steepest Ascent Hill-climbing.....	11
2.2.3 Hill-climbing with Sideways Move.....	13
2.2.4 Random Restart Hill-climbing.....	16
2.2.5 Stochastic Hill-climbing.....	18
2.2.6 Simulated Annealing.....	20
2.2.7 Genetic Algorithm.....	22
2.3 Hasil Eksperimen dan Analisis Setiap Algoritma.....	27
2.3.1 Steepest Ascent Hill-climbing.....	27
2.3.2 Hill-climbing with Sideways Move.....	31
2.3.3 Random Restart Hill-climbing.....	36
2.3.4 Stochastic Hill-climbing.....	41
2.3.5 Simulated Annealing.....	45
2.3.6 Genetic Algorithm.....	52
2.4 Analisis.....	77
BAB III.....	84
KESIMPULAN DAN SARAN.....	84
PEMBAGIAN TUGAS.....	86
REFERENSI.....	88

BAB I

DESKRIPSI PERSOALAN

1.1 Deskripsi Tugas

25	16	80	104	90				
115	98	4	1	97				90
42	111	85	2	75			91	
66	72	27	102	48		75		70
67	18	119	106	5	5	48	13	
116	17	14	73	95	95	114	23	86
40	50	81	65	79	79	19	37	10
56	120	55	49	35	35	74	96	100
36	110	46	22	101	101	60	84	11

Diagonal magic cube merupakan kubus yang tersusun dari angka 1 hingga n^3 tanpa pengulangan dengan n adalah panjang sisi pada kubus tersebut. Angka-angka pada tersusun sedemikian rupa sehingga properti-properti berikut terpenuhi:

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

Pada persoalan ini, akan diselesaikan permasalahan Diagonal Magic Cube berukuran $5 \times 5 \times 5$. Initial state dari suatu kubus adalah susunan angka 1 hingga 5^3 secara acak. Kemudian, tiap iterasi pada algoritma local search, langkah yang boleh dilakukan adalah

menukar posisi dari 2 angka pada kubus tersebut (2 angka yang ditukar tidak harus bersebelahan).

Magic number sendiri ditentukan berdasarkan rumus berikut,

$$\text{magic_number} = \frac{n \times (n^3 + 1)}{2}$$

dengan n adalah jumlah sisi pada kubus.

BAB II

PEMBAHASAN

2.1 Pemilihan Objective Function

Fungsi Objektif yang digunakan pada permasalahan ini, memanfaatkan salah satu properti yang dimiliki oleh Diagonal Magic Cube. Untuk suatu Magic Cube dengan ukuran $N \times N \times N$, maka Magic Number atau angka konstan yang pasti menjadi jumlah penjumlahan dari setiap baris, kolom, tiang, diagonal level, maupun diagonal ruang didefinisikan sebagai berikut.

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

Sehingga untuk ukuran magic cube dengan $n=5$ ($5 \times 5 \times 5$), Magic Number yang didapatkan adalah

$$M = \frac{5(125 + 1)}{2} = 315$$

Fungsi objektif yang dipilih akan mengukur seberapa jauh konfigurasi kubus saat ini dari kondisi ideal Diagonal Magic Cube. Salah satu pendekatan yang dapat digunakan adalah menghitung jumlah deviasi terkecil dari penjumlahan pada setiap baris, kolom, tiang, diagonal ruang dan diagonal level. Dengan menggunakan pendekatan fungsi objektif ini, semakin kecil jumlah deviasi, semakin mendekati konfigurasi penyusunan kubus tersebut dengan Diagonal Magic Cube. Jika sebuah konfigurasi tepat, nilai fungsi objektif akan bernilai 0, karena setiap baris, kolom, tiang, dan diagonal ruang telah memenuhi syarat penjumlahan dan menghasilkan magic number yang sesuai.

Sebuah magic cube $5 \times 5 \times 5$ dapat dipecah menjadi 5 tingkatan dan secara spesifik setiap konstrain yang harus dipenuhi adalah:

- 1) Jumlah tiap baris pada lima tingkat kubus harus 315 (terdapat 25 baris yang harus dihitung).
- 2) Jumlah tiap kolom pada lima tingkat kubus tingkatan harus 315 (terdapat 25 kolom yang harus dihitung).
- 3) Jumlah tiap tiang yang merambat pada sumbu Z harus 315 (terdapat 25 tiang yang harus dihitung).

- 4) Jumlah tiap diagonal ruang pada kubus harus 315 (pada kasus ini terdapat 4 diagonal ruang).
- 5) Jumlah tiap diagonal bidang pada tiap sisi kubus harus 315 (terdapat 6 sisi pada kubus ditambah dengan diagonal bidang tiap level, sehingga 30 diagonal bidang yang harus dihitung).

Terdapat 109 total penjumlahan yang harus dilakukan untuk mencari deviasi suatu konfigurasi dengan magic cube. Secara matematis rumus pencarian suatu deviasi dari tiap syarat dapat dituliskan sebagai berikut.

$$f(x) = \sum |sum - M|$$

Untuk setiap baris, kolom, diagonal, dan tiang, dihitung selisih antara hasil penjumlahannya dengan magic number/constant (315 untuk kubus berukuran $5 \times 5 \times 5$). Selisih-selisih ini kemudian dijumlahkan untuk membentuk nilai fungsi objektif. Semakin kecil nilai fungsi objektif tersebut (mendekati atau sama dengan 0), semakin baik pula konfigurasi penyusunan angka pada kubus. Nilai 0 menandakan bahwa setiap baris, kolom, tiang, dan diagonal telah memenuhi syarat penjumlahan yang sesuai dengan magic number, menunjukkan konfigurasi yang optimal.

Rumus fungsi objektif secara keseluruhan adalah sebagai berikut,

$$\begin{aligned} f(x) = & - \left(\sum_{i=1}^5 \sum_{j=1}^5 |jumlah\ baris(i,j) - M| + \sum_{i=1}^5 \sum_{k=1}^5 |jumlah\ kolom(i,k) - M| + \right. \\ & \left. \sum_{j=1}^5 \sum_{k=1}^5 |jumlah\ tiang(j,k) - M| + \sum_{d=1}^4 |jumlah\ diagonal\ ruang(d) - M| + \right. \\ & \left. \sum_{p=1}^6 \sum_{d=1}^4 |jumlah\ diagonal\ bidang(p,d) - M| + \sum_{l=2}^4 \sum_{d=1}^2 |jumlah\ diagonal\ level(l,d) - M| \right) \end{aligned}$$

Penjelasan:

- 1) Jumlah baris(i,j) merupakan jumlah jumlah baris ke-*i* pada tingkat ke-*j*
- 2) Jumlah kolom(i,k) merupakan jumlah kolom ke-*i* pada tingkat ke-*k*
- 3) Jumlah tiang(j,k) merupakan jumlah baris ke-*j* pada tiang ke-*k*
- 4) Jumlah diagonal ruang(d) merupakan jumlah keempat diagonal ruang
- 5) Jumlah diagonal bidang(p,d) merupakan jumlah diagonal ke-*d* pada sisi ke-*p*

- 6) Jumlah diagonal level(l,d) merupakan jumlah diagonal ke- d pada tiap level l kubus (level yang dihitung hanyalah level ke 2-4 karena level pertama dan ke-5 telah dihitung diagonal bidangnya)

Hasil fungsi objektif akan < 0 karena pendekatan yang digunakan merupakan deviasi tiap baris,kolom,diagonal, dan tiang dari *magic number* sehingga jika jumlah semakin besar, maka konfigurasi semakin jauh dari keadaan optimal, yaitu saat fungsi objektif bernilai 0 atau penjumlahan tiap baris, kolom, tiang, dan diagonal sudah sama dengan *magic number*.

2.2 Penjelasan Implementasi Algoritma Local Search

2.2.1 Fungsi/Kelas Umum

Berisi fungsi dan kelas yang digunakan secara umum pada setiap algoritma local search.

1. Kelas MagicCube

Kelas untuk membentuk dan mengevaluasi kubus *magic cube* berukuran $n \times n \times n$. Kelas ini menyediakan metode untuk menginisialisasi kubus, menghitung *magic number*, menghitung jumlah elemen di sepanjang baris, kolom, pilar, dan diagonal, serta mengevaluasi skor kubus sesuai dengan kriteria kubus ajaib.

2. Fungsi `__init__(self, n)`

Konstruktor untuk menginisialisasi objek MagicCube dengan ukuran n .

```
def __init__(self, n):
    self.n = n
    self.magic_number = self.calculate_magic_number(n)
    self.cube = self.initialize_cube(n)
    self.position_map = self.create_position_map()
    # Track rows, columns, pillars, and diagonals
    self.row_sums = np.zeros((n, n))      # row sums for each layer
    self.col_sums = np.zeros((n, n))      # column sums for each layer
    self.pillar_sums = np.zeros((n, n))   # pillar sums for each layer
    self.diagonal_sums = {
        "xy_main": np.zeros(n),           # main xy diagonal for each
        "xy_anti": np.zeros(n),          # anti xy diagonal for each
        "yz_main": np.zeros(n),           # main yz diagonal for each
        "yz_anti": np.zeros(n),          # anti yz diagonal for each
        "zx_main": np.zeros(n),           # main zx diagonal for each
        "zx_anti": np.zeros(n)           # anti zx diagonal for each
    }
    for layer in range(n):
        for row in range(n):
            for col in range(n):
                self.cube[layer][row][col] = random.randint(1, 9)
```

```

        }
        self.main_diagonals = [0, 0, 0, 0]      # 3D diagonals
        self.initialize_sums()
        self.score = self.calculate_objective_function()
    
```

3. Fungsi calculate_magic_number(self, n)

Menghitung *magic number* untuk kubus berukuran n, yaitu jumlah nilai yang harus dicapai oleh baris, kolom, pilar, atau diagonal.

```

def calculate_magic_number(self, n):
    return n * (n**3 + 1) // 2
    
```

4. Fungsi initialize_cube(self, n)

Menginisialisasi kubus dengan susunan angka acak dari 1 hingga n^3 untuk *initial state* kubus.

```

def initialize_cube(self, n):
    return np.random.permutation(n**3).reshape(n, n, n) + 1
    
```

5. Fungsi create_position_map(self)

Membuat *map* posisi dari setiap nilai dalam kubus untuk mempermudah pencarian posisi elemen tertentu.

```

def create_position_map(self):
    return {self.cube[i, j, k]: (i, j, k) for i in range(self.n) for j in
range(self.n) for k in range(self.n)}
    
```

6. Fungsi get_position(self, value)

Mengembalikan posisi (koordinat) dari suatu nilai tertentu dalam kubus.

```

def get_position(self, value):
    return self.position_map[value]
    
```

7. Fungsi initialize_sums(self)

Menghitung jumlah nilai pada setiap baris, kolom, pilar, serta diagonal dalam tiap lapisan, termasuk diagonal 3D utama.

```

def initialize_sums(self):
    for i in range(self.n):
        for j in range(self.n):
            # Row, column, and pillar sums
            self.row_sums[i, j] = np.sum(self.cube[i, j, :])
            self.col_sums[i, j] = np.sum(self.cube[i, :, j])
            self.pillar_sums[i, j] = np.sum(self.cube[:, i, j])
    
```

```

        # Diagonal sums in each layer
        self.diagonal_sums["xy_main"][i] = np.sum([self.cube[i, j, j] for j
in range(self.n)])
        self.diagonal_sums["xy_anti"][i] = np.sum([self.cube[i, j, self.n -
j - 1] for j in range(self.n)])
        self.diagonal_sums["yz_main"][i] = np.sum([self.cube[j, i, j] for j
in range(self.n)])
        self.diagonal_sums["yz_anti"][i] = np.sum([self.cube[j, i, self.n -
j - 1] for j in range(self.n)])
        self.diagonal_sums["zx_main"][i] = np.sum([self.cube[j, j, i] for j
in range(self.n)])
        self.diagonal_sums["zx_anti"][i] = np.sum([self.cube[j, self.n - j
- 1, i] for j in range(self.n)])

        # 3D main diagonals
        self.main_diagonals[0] = np.sum([self.cube[i, i, i] for i in
range(self.n)])
        self.main_diagonals[1] = np.sum([self.cube[i, i, self.n - i - 1] for i
in range(self.n)])
        self.main_diagonals[2] = np.sum([self.cube[i, self.n - i - 1, i] for i
in range(self.n)])
        self.main_diagonals[3] = np.sum([self.cube[self.n - i - 1, i, i] for i
in range(self.n)])

```

8. Fungsi swap_elements(self, pos1, pos2)

Menukar dua elemen dalam kubus lalu memperbarui jumlah baris, kolom, pilar, dan diagonal yang terkait, serta memperbarui skor keseluruhan.

```

def swap_elements(self, pos1, pos2):
    val1 = self.cube[pos1]
    val2 = self.cube[pos2]
    self.cube[pos1], self.cube[pos2] = val2, val1

    # Update position map
    self.position_map[val1], self.position_map[val2] = pos2, pos1

    # Update the row, column, and pillar sums
    x1, y1, z1 = pos1
    x2, y2, z2 = pos2

    # Update row, column, and pillar sums
    self.row_sums[x1, y1] += val2 - val1
    self.row_sums[x2, y2] += val1 - val2
    self.col_sums[x1, z1] += val2 - val1

```

```

        self.col_sums[x2, z2] += val1 - val2
        self.pillar_sums[y1, z1] += val2 - val1
        self.pillar_sums[y2, z2] += val1 - val2

        # Update diagonal sums if affected
        if y1 == z1:
            self.diagonal_sums["xy_main"][x1] += val2 - val1
        if y2 == z2:
            self.diagonal_sums["xy_main"][x2] += val1 - val2
        if y1 == self.n - z1 - 1:
            self.diagonal_sums["xy_anti"][x1] += val2 - val1
        if y2 == self.n - z2 - 1:
            self.diagonal_sums["xy_anti"][x2] += val1 - val2
        if x1 == z1:
            self.diagonal_sums["yz_main"][y1] += val2 - val1
        if x2 == z2:
            self.diagonal_sums["yz_main"][y2] += val1 - val2
        if x1 == self.n - z1 - 1:
            self.diagonal_sums["yz_anti"][y1] += val2 - val1
        if x2 == self.n - z2 - 1:
            self.diagonal_sums["yz_anti"][y2] += val1 - val2
        if x1 == y1:
            self.diagonal_sums["zx_main"][z1] += val2 - val1
        if x2 == y2:
            self.diagonal_sums["zx_main"][z2] += val1 - val2
        if x1 == self.n - y1 - 1:
            self.diagonal_sums["zx_anti"][z1] += val2 - val1
        if x2 == self.n - y2 - 1:
            self.diagonal_sums["zx_anti"][z2] += val1 - val2

        # Similar updates for yz and zx diagonals, as well as 3D main
        diagonals
        if x1 == y1 == z1:
            self.main_diagonals[0] += val2 - val1
        if x2 == y2 == z2:
            self.main_diagonals[0] += val1 - val2
        if x1 == y1 == self.n - z1 - 1:
            self.main_diagonals[1] += val2 - val1
        if x2 == y2 == self.n - z2 - 1:
            self.main_diagonals[1] += val1 - val2
        if x1 == self.n - y1 - 1 == z1:
            self.main_diagonals[2] += val2 - val1
    
```

```

        if x2 == self.n - y2 - 1 == z2:
            self.main_diagonals[2] += val1 - val2
        if x1 == self.n - y1 - 1 == self.n - z1 - 1:
            self.main_diagonals[3] += val2 - val1
        if x2 == self.n - y2 - 1 == self.n - z2 - 1:
            self.main_diagonals[3] += val1 - val2

    self.score = self.calculate_objective_function()

```

9. Fungsi calculate_objective_function(self)

Menghitung skor kubus berdasarkan perbedaan antara jumlah elemen pada setiap baris, kolom, pilar, dan diagonal dengan nilai sihir. Semakin rendah skornya, semakin mendekati sifat kubus ajaib (seperti yang telah dipaparkan pada bagian sebelumnya).

```

def calculate_objective_function(self):
    target = self.magic_number
    score = 0

    for i in range(self.n):
        for j in range(self.n):
            score += abs(self.row_sums[i, j] - target)
            score += abs(self.col_sums[i, j] - target)
            score += abs(self.pillar_sums[i, j] - target)

    for key in self.diagonal_sums:
        for i in range(self.n):
            score += abs(self.diagonal_sums[key][i] - target)

    for main_diag in self.main_diagonals:
        score += abs(main_diag - target)

    return -score

```

10. Fungsi getCurrentScore(self)

Mengembalikan skor saat ini dari kubus, yang menunjukkan seberapa dekat susunan kubus dengan kondisi kubus ajaib.

```

def getCurrentScore(self):
    return self.score

```

11. Fungsi evaluate_swap(self, pos1, pos2)

Mengevaluasi skor dari menukar dua elemen pada posisi pos1 dan pos2 tanpa benar-benar mengubah kubus aslinya.

```

def evaluate_swap(self, pos1, pos2):

```

```

# Swap elements temporarily
self.swap_elements(pos1, pos2)

# Calculate score
score = self.getCurrentScore()

# Revert swap
self.swap_elements(pos1, pos2)

return score, pos1, pos2

```

12. Kelas AlgorithmManager

Kelas untuk mengelola strategi algoritma yang digunakan untuk menyelesaikan masalah Magic Cube. Memungkinkan pengguna untuk mengganti strategi sesuai dengan kebutuhan dan mengimplementasikan eksekusi solusi menggunakan strategi yang dipilih.

13. Fungsi setStrategy(self, strategy: AlgorithmStrategy)

Mengatur strategi algoritma yang akan digunakan dalam penyelesaian Magic Cube.

```

def setStrategy(self, strategy: AlgorithmStrategy):
    self.strategy = strategy

```

14. Fungsi solve(self, magic_cube, **kwargs)

Mengeksekusi strategi yang telah diatur pada Magic Cube yang diberikan, menggunakan parameter tambahan jika diperlukan.

```

def solve(self, magic_cube, **kwargs):
    return self.strategy.execute(magic_cube, **kwargs)

```

2.2.2 Steepest Ascent Hill-climbing

Steepest Ascent Hill-climbing adalah algoritma local search yang bekerja dengan memilih tetangga terbaik dari state saat ini pada setiap iterasi, yaitu tetangga yang memiliki nilai objective function paling tinggi (dalam kasus maksimisasi). Algoritma ini dimulai dengan solusi awal acak, kemudian evaluasi semua tetangga dan **pindah ke tetangga hanya jika nilai objective function tetangga lebih besar dari nilai objective function saat ini** (ada lebih banyak elemen dari kubus yang sesuai dengan magic number). Proses ini diulangi sampai tidak ada lagi tetangga yang lebih baik dari solusi saat ini.

- **Kelebihan:** Steepest Ascent selalu memilih langkah yang membawa peningkatan terbesar pada objective function.

- **Kekurangan:** Algoritma ini dapat terjebak pada local optimum, di mana tidak ada tetangga yang lebih baik, tetapi solusi tersebut bukanlah solusi global terbaik (belum mencapai puncak).

Berikut adalah proses penyelesaian Diagonal Magic Cube berukuran $5 \times 5 \times 5$ dengan Steepest Ascent Hill-climbing:

- 1) Tentukan *initial state* dengan menginisialisasi secara acak magic cube dengan susunan angka 1 hingga 5^3 . Setelah itu, hitung magic number kubus.
 - 2) Setelah mendapatkan kubus awal dan magic number, evaluasi skor *state* saat ini dengan menggunakan objective function yang telah ditentukan untuk disimpan sebagai skor saat ini.
 - 3) Selanjutnya, bangkitkan semua kemungkinan suksesor dengan menukar setiap pasangan elemen di dalam kubus. Setiap kondisi baru yang dihasilkan kemudian dievaluasi skornya menggunakan objective function yang sama.
 - 4) Bandingkan skor saat ini dengan skor suksesor. Jika salah satu suksesor memiliki skor yang lebih tinggi daripada skor saat ini, maka pindahkan state ke suksesor tersebut dan perbarui skor saat ini. Iterasi proses ini hingga semua suksesor di cek untuk mendapatkan *highest-valued successor*.
 - 5) Ulangi proses evaluasi dan pemilihan suksesor ini hingga tidak ada lagi skor yang lebih baik ditemukan dari suksesor state saat ini. Hal ini dijadikan kondisi terminasi sehingga algoritma mengembalikan hasil terbaiknya.

Fungsi/Kelas Umum untuk Steepest Ascent Hill-climbing

```
from itertools import combinations
from concurrent.futures import ThreadPoolExecutor, as_completed
import time
from Algorithms.strategy import AlgorithmStrategy
from plot import PlotManager

class SteepestAscentStrategy(AlgorithmStrategy):
    def execute(self, cube, plot=True):
        start_time = time.time()
        current_score = cube.getCurrentScore()

        improved = True
        iterations = []
        scores = []

        while improved:
            improved = False
```

```

        best_score = current_score
        best_positions = None

        # Generate all unique pairs of positions
        positions = [cube.get_position(i) for i in range(1, cube.n**3)]

        with ThreadPoolExecutor(max_workers=8) as executor:
            futures = {
                executor.submit(cube.evaluate_swap_score, pos1, pos2): (pos1,
pos2)
                for pos1, pos2 in combinations(positions, 2)
            }

            # Retrieve swap scores as they complete
            for future in as_completed(futures):
                swap_score = future.result()
                pos1, pos2 = futures[future]

                if swap_score > best_score:
                    best_score = swap_score
                    best_positions = (pos1, pos2)

            if best_positions:
                print(f"Best Score: {best_score}")
                pos1, pos2 = best_positions
                cube.swap_elements(pos1, pos2)
                current_score = best_score
                improved = True
                scores.append(current_score)

                iterations.append((cube.cube.copy(), current_score))

        # Calculate the time taken for execution
        end_time = time.time()
        total_time = end_time - start_time
        total_iterations = len(scores)
        final_score = current_score

        if plot:
            plot_manager = PlotManager(scores)
            plot_manager.plot_objective_function(
                total_iterations=total_iterations,
                total_time=total_time,
                final_score=final_score
            )

    return cube.cube, current_score, iterations

```

2.2.3 Hill-climbing with Sideways Move

Hill-climbing with Sideways Move adalah variasi dari hill-climbing yang memungkinkan pergerakan lateral ketika objective function tidak membaik. Misalnya, jika setelah melakukan pertukaran dua angka dalam kubus tidak ada tetangga yang memberikan nilai objective

function lebih baik, algoritma ini tidak langsung berhenti, melainkan **tetap dapat bergerak ke tetangga yang memiliki nilai objective function sama dengan solusi saat ini** (sideways move). Hal ini bertujuan untuk menghindari stuck pada kondisi dimana solusi tidak memburuk, tetapi juga tidak membaik. Algoritma ini baru akan berhenti ketika tidak ada lagi tetangga yang lebih baik atau sama dengan solusi saat ini (semua tetangga lebih buruk).

- **Kelebihan:** Sideways move meningkatkan peluang sukses suatu masalah dengan tidak berhenti di kondisi *flat* (plato).
- **Kekurangan:** Walaupun lebih baik dari algoritma Steepest Ascent, algoritma ini masih dapat *stuck* di sekitar local optimum dan memperlambat waktu pencarian.

Berikut adalah proses penyelesaian Diagonal Magic Cube berukuran $5 \times 5 \times 5$ dengan Hill-climbing with Sideways Move.

- 1) Tentukan initial state dengan menginisialisasi secara acak magic cube dengan susunan angka 1 hingga 53. Setelah itu, hitung magic number kubus.
- 2) Setelah mendapatkan kubus awal dan magic number, evaluasi skor state saat ini dengan menggunakan objective function yang telah ditentukan untuk disimpan sebagai skor saat ini.
- 3) Selanjutnya, bangkitkan semua kemungkinan suksesor dengan menukar setiap pasangan elemen di dalam kubus. Setiap kondisi baru yang dihasilkan kemudian dievaluasi skornya menggunakan objective function yang sama.
- 4) Setelah semua suksesor dihasilkan dan dievaluasi, bandingkan setiap skor suksesor dengan skor state saat ini. Jika ada suksesor yang memiliki skor lebih tinggi, maka *current state* akan berpindah ke suksesor tersebut.
- 5) Jika tidak ada suksesor yang memiliki skor lebih tinggi, tetapi pindahkan *current state* ke suksesor yang memiliki skor sama dengan skor state saat ini.
- 6) Lakukan proses ini hingga semua suksesor memiliki skor lebih rendah dari state saat ini. Kondisi ini dijadikan sebagai kriteria terminasi dimana tidak ada lagi suksesor dengan skor lebih tinggi atau sama dengan skor state saat ini.

Fungsi/Kelas Umum untuk Hill-climbing with Sideways Move

```
from Algorithms.strategy import AlgorithmStrategy
from concurrent.futures import ThreadPoolExecutor, as_completed
from plot import PlotManager
import os
```

```

import time
from itertools import combinations

class SidewaysStrategy(AlgorithmStrategy):
    def execute(self, cube, **kwargs):
        max_sideways = kwargs.get("max_sideways", 10)
        start_time = time.time()

        current_score = cube.getCurrentScore()
        improved = True
        sideways_moves = 0
        scores = [current_score]
        iterations = []

        while improved:
            improved = False
            best_score = current_score
            best_positions = None
            found_better = False

            # Generate all unique position pairs and evaluate scores
            positions = [cube.get_position(i) for i in range(1, cube.n**3)]
            with ThreadPoolExecutor(os.cpu_count()) as executor:
                futures = {
                    executor.submit(cube.evaluate_swap_score, pos1, pos2): (pos1,
pos2)
                    for pos1, pos2 in combinations(positions, 2)
                }

                for future in as_completed(futures):
                    swap_score = future.result()
                    pos1, pos2 = futures[future]

                    if swap_score > best_score:
                        best_score = swap_score
                        best_positions = (pos1, pos2)
                        found_better = True
                        sideways_moves = 0 # Reset sideways moves on improvement
                    elif swap_score == best_score and not found_better:
                        # Allow sideways move if no better score is found
                        best_positions = (pos1, pos2)

# If a swap was selected, apply it and update scores
if best_positions:
    pos1, pos2 = best_positions
    cube.swap_elements(pos1, pos2)
    current_score = best_score
    scores.append(current_score)

    if found_better:
        improved = True
    elif sideways_moves < max_sideways:
        sideways_moves += 1

```

```

        improved = True
    else:
        improved = False # Stop if max sideways moves reached

    # Store the current iteration state
    iterations.append((cube.cube.copy(), current_score))

    # Calculate the time taken for execution
    end_time = time.time()
    total_time = end_time - start_time
    total_iterations = len(scores)
    final_score = current_score

    # Plot the results
    plot_manager = PlotManager(scores)
    plot_manager.plot_sideways_strategy(total_iterations, total_time,
sideways_moves, final_score)

    return cube.cube, current_score, iterations

```

2.2.4 Random Restart Hill-climbing

Random Restart Hill-climbing adalah varian dari algoritma hill-climbing yang mencoba mengatasi jebakan local optimum dengan cara **menggunakan restart dari solusi acak yang baru setiap kali algoritma mencapai local optimum**. Pada setiap iterasi, algoritma akan melakukan pertukaran dua angka dalam kubus, seperti pada hill-climbing biasa, dan terus mencari tetangga terbaik hingga tidak ada lagi tetangga yang lebih baik. Jika algoritma terjebak di local optimum (tidak ada tetangga yang dapat meningkatkan nilai objective function), ia akan melakukan restart dengan solusi acak baru dan mengulang proses hill-climbing dari awal.

- **Kelebihan:** Algoritma ini memberikan kesempatan lebih besar untuk menemukan solusi global optimum karena dengan melakukan restart, algoritma dapat mengeksplorasi ruang solusi yang lebih luas.
- **Kekurangan:** Meskipun bisa menghindari local optimum, proses restart yang berulang-ulang memerlukan lebih banyak waktu pencarian.

Berikut adalah proses penyelesaian Diagonal Magic Cube dengan Random Restart Hill-climbing.

- 1) Tentukan initial state dengan menginisialisasi secara acak magic cube dengan susunan angka 1 hingga 53. Setelah itu, hitung magic number kubus.

- 2) Setelah mendapatkan kubus awal dan magic number, evaluasi skor state saat ini dengan menggunakan objective function yang telah ditentukan untuk disimpan sebagai skor saat ini.
- 3) Selanjutnya, bangkitkan semua kemungkinan suksesor dengan menukar setiap pasangan elemen di dalam kubus. Setiap kondisi baru yang dihasilkan kemudian dievaluasi skornya menggunakan objective function yang sama.
- 4) Setelah semua suksesor dihasilkan dan dievaluasi, bandingkan setiap skor suksesor dengan skor state saat ini. Jika ada suksesor yang memiliki skor lebih tinggi, maka *current state* akan berpindah ke suksesor tersebut.
- 5) Jika tidak ada suksesor yang memiliki skor lebih tinggi, lakukan restart dengan menginisialisasi ulang kubus secara acak.
- 6) Ulangi proses ini hingga jumlah maksimum restart telah tercapai.
- 7) Solusi yang diambil adalah solusi dengan skor terbaik dari seluruh proses restart yang dilakukan.

Fungsi/Kelas Umum untuk Random Restart Hill-climbing

```

import time
from Algorithms.strategy import AlgorithmStrategy
from plot import PlotManager
from Algorithms.steepest_ascent import SteepestAscentStrategy

steepestAscent = SteepestAscentStrategy()

class RandomRestartStrategy(AlgorithmStrategy):
    def execute(self, cube, max_restarts=3):
        start_time_all = time.time()
        best_score = float("-inf")
        best_cube = None
        best_iterations = None
        all_scores = []
        restart_times = []
        restart_iterations = []

        for _ in range(max_restarts):
            cube.initialize_cube(cube.n)
            cube.initialize_sums()

            start_time = time.time()
            _, current_score, iterations = steepestAscent.execute(cube, plot=False)
            end_time = time.time()

            restart_scores = [score for _, score in iterations]
            all_scores.append(restart_scores)

        return best_score, best_cube, best_iterations, start_time_all, end_time, all_scores
    
```

```

        restart_times.append(end_time - start_time)
        restart_iterations.append(len(iterations))

        if current_score > best_score:
            best_score = current_score
            best_cube = cube.cube.copy()
            best_iterations = iterations

        end_time_all = time.time()
        final_exec_time = end_time_all - start_time_all

        plot_manager = PlotManager()
        plot_manager.plot_random_restart_summary(all_scores, restart_times,
        restart_iterations, final_exec_time)

    return best_cube, best_score, best_iterations

```

2.2.5 Stochastic Hill-climbing

Stochastic Hill-climbing bekerja dengan memilih **satu** tetangga secara acak pada setiap iterasi, dan hanya bergerak ke tetangga tersebut jika nilai objective function lebih baik dari state saat ini. Ini berbeda dari steepest ascent yang mengevaluasi semua tetangga. Jadi, jika setelah pertukaran angka, jumlah baris, kolom, tiang, atau diagonal yang sesuai dengan magic number meningkat, algoritma akan berpindah ke tetangga tersebut. Algoritma ini akan terus melakukan iterasi hingga batasan jumlah iterasi maksimum (nmax) tercapai.

- **Kelebihan:** Algoritma ini lebih cepat karena hanya mengevaluasi satu tetangga per iterasi, daripada semua tetangga seperti pada steepest ascent.
- **Kekurangan:** Karena bersifat acak, ada kemungkinan untuk melewatkannya tetangga yang jauh lebih baik, sehingga sulit untuk selalu menemukan solusi terbaik secara cepat.

Proses penyelesaian Diagonal Magic Cube dengan Stochastic Hill-climbing, yakni sebagai berikut.

- 1) Tentukan initial state dengan menginisialisasi secara acak magic cube dengan susunan angka 1 hingga 5^3 . Setelah itu, hitung magic number kubus.
- 2) Setelah mendapatkan kubus awal dan magic number, evaluasi skor state saat ini dengan menggunakan objective function yang telah ditentukan kemudian simpan sebagai skor saat ini.
- 3) Bangkitkan suksesor acak dari state saat ini dengan memilih posisi acak untuk ditukar posisinya.

- 4) Evaluasi kembali hasil pertukaran tersebut menggunakan objective function lalu bandingkan skor saat ini dengan skor suksesor. Jika skor suksesor hasil pertukaran lebih tinggi daripada skor saat ini, maka pertahankan konfigurasi terbaru dan perbarui skor saat ini dengan suksesor. Jika tidak, kembalikan pertukaran tadi ke posisi awal.
- 5) Ulangi proses pemilihan suksesor dan evaluasi sebanyak jumlah iterasi yang ditentukan. Dalam setiap iterasi tersebut dengan suksesor acak, algoritma berusaha mengeksplorasi konfigurasi yang bertujuan menemukan konfigurasi terbaik.

Fungsi/Kelas Umum untuk Stochastic Hill-climbing

```

from Algorithms.strategy import AlgorithmStrategy
from plot import PlotManager
import numpy as np
import time

class StochasticStrategy(AlgorithmStrategy):
    def execute(self, cube, **kwargs):
        max_steps = kwargs.get("max_steps", 10000)
        start_time = time.time()

        current_score = cube.getCurrentScore()
        steps = 0
        scores = [current_score]
        iterations = []

        while steps < max_steps:
            steps += 1
            # Choose two random positions to swap
            pos1 = tuple(np.unravel_index(np.random.randint(0, cube.n**3), (cube.n,
            cube.n, cube.n)))
            pos2 = tuple(np.unravel_index(np.random.randint(0, cube.n**3), (cube.n,
            cube.n, cube.n)))

            cube.swap_elements(pos1, pos2)
            new_score = cube.getCurrentScore()

            # If the swap improves the score, accept it; otherwise, revert it
            if new_score > current_score:
                current_score = new_score
            else:
                cube.swap_elements(pos1, pos2)

            iterations.append((cube.cube.copy(), current_score))
            scores.append(current_score)

            # print(f"Step {steps} | Pos1: {pos1} <-> Pos2: {pos2} | Current Score:
            # {current_score}")

        end_time = time.time()
    
```

```

total_time = end_time - start_time
final_score = current_score
total_iterations = steps

plot_manager = PlotManager(scores)
plot_manager.plot_objective_function(total_iterations=total_iterations,
total_time=total_time, final_score=final_score)

return cube.cube, final_score, iterations

```

2.2.6 Simulated Annealing

Simulated Annealing mengombinasikan efisiensi dari algoritma Hill-climbing dengan *completeness* dari Random Walk, dimulai dengan solusi awal yang acak kemudian pada setiap iterasi dihitung nilai perubahannya untuk diambil solusi yang lebih baik dari saat ini. Jika solusi ternyata lebih buruk, maka terima dengan probabilitas tertentu bergantung pada "temperatur" (T). Secara bertahap, T akan berkurang seiring dengan iterasi sehingga semakin sedikit solusi buruk yang diterima.

- **Kelebihan:** Tidak mudah terjebak di optimum lokal karena adanya probabilitas menerima solusi yang lebih buruk di awal. Hal ini juga memungkinkan cakupan pencarian yang lebih *complete* dibanding algoritma Hill-climbing.
- **Kekurangan:** Karena jumlah iterasi yang bisa saja banyak, waktu komputasi bisa menjadi lebih lama. Kinerja algoritma juga bergantung pada parameter yang digunakan untuk "temperturnya" sehingga masih tidak ada jaminan solusi yang dihasilkan merupakan optimum global.

Proses penyelesaian Diagonal Magic Cube dengan Simulated Annealing, yakni sebagai berikut.

- 1) Tentukan initial state dengan menginisialisasi secara acak magic cube dengan susunan angka 1 hingga 5^3 . Setelah itu, hitung magic number kubus.
- 2) Setelah mendapatkan kubus awal dan magic number, evaluasi skor state saat ini dengan menggunakan objective function yang telah ditentukan kemudian simpan sebagai skor saat ini.
- 3) Pilih parameter Simulated Annealing yang dibutuhkan, yakni suhu temperatur awal dan laju penurunan temperatur. Parameter ini akan menjadi kontrol atas seberapa besar kemungkinan untuk menerima solusi yang lebih buruk, juga seberapa cepat iterasi

selesai. Semakin lambat temperatur menurun, semakin banyak kemungkinan mencari solusi, namun semakin lama waktu komputasinya.

- 4) Bangkitkan suksesor dari *current state* dengan memilih posisi acak untuk ditukar posisinya.
- 5) Evaluasi hasil pertukaran tersebut menggunakan objective function, simpan hasil skornya sebagai skor suksesor. Gunakan hasil skor tersebut untuk menghitung skor delta (selisih), yakni skor suksesor dikurang skor saat ini.
- 6) Jika skor delta bernilai positif (artinya skor suksesor lebih baik), maka *current state* akan berpindah ke suksesor tersebut. Jika tidak (skor suksesor lebih buruk), akan dievaluasi dahulu dengan probabilitas berupa $e^{\Delta \text{skor}/\text{temperatur}}$. Bandingkan probabilitas tersebut dengan nilai statik yaitu 0.5 sebagai titik tengah pilihan baik atau buruk. Jika probabilitas bernilai lebih besar darinya, maka suksesor tersebut masih dapat diterima walaupun skornya lebih buruk.
- 7) Ulangi proses pemilihan suksesor dan evaluasi tersebut, secara bertahap temperatur akan menurun. Iterasi ini diterminasi ketika temperatur sudah bernilai sangat rendah, dari sini solusi terakhir pun diambil.

Fungsi/Kelas Umum untuk Simulated Annealing

```
import numpy as np
from Algorithms.strategy import AlgorithmStrategy
import math
import time
from plot import PlotManager

class SimulatedAnnealingStrategy(AlgorithmStrategy):
    def execute(self, cube, **kwargs):
        initial_temp = kwargs.get("initial_temp", 1000)
        cooling_rate = kwargs.get("cooling_rate", 0.9999)

        start_time = time.time()
        current_score = cube.getCurrentScore()
        # print(f"Initial score: {current_score}")
        temperature = initial_temp
        scores = [current_score]
        acceptance_probs = []
        iterations = []

        iteration = 0
        stuck_frequency = 0

        while temperature > 1e-3 and current_score != 0:
```

```

iteration += 1

pos1, pos2 = self._get_random_positions(cube.n)
cube.swap_elements(pos1, pos2)
new_score = cube.getCurrentScore()
delta_score = new_score - current_score

accepted_with_probability = False
accepted = False

if delta_score > 0:
    accepted = True
else:
    acceptance_probability = math.exp(delta_score / temperature)
    acceptance_probs.append(acceptance_probability)
    accepted_with_probability = np.random.rand() < acceptance_probability

if accepted_with_probability:
    stuck_frequency += 1

if accepted or accepted_with_probability:
    current_score = new_score
else:
    cube.swap_elements(pos1, pos2) ## revert

temperature *= cooling_rate
scores.append(current_score)
iterations.append((cube.cube.copy(), current_score))

end_time = time.time()
total_time = end_time - start_time
total_iterations = iteration
final_score = current_score

plot_manager = PlotManager(scores, acceptance_probs)
plot_manager.plot_objective_function_simulated(
    total_iterations=total_iterations,
    total_time=total_time,
    final_score=final_score,
    stuck_frequency=stuck_frequency
)
plot_manager.plot_acceptance_simulated_annealing()

return cube.cube, current_score, iterations

def _get_random_positions(self, n):
    pos1 = tuple(np.unravel_index(np.random.randint(0, n**3), (n, n, n)))
    pos2 = tuple(np.unravel_index(np.random.randint(0, n**3), (n, n, n)))
    return pos1, pos2

```

2.2.7 Genetic Algorithm

Genetic Algorithm berupaya menemukan solusi optimal dengan menyeleksi, mengkombinasikan, dan memodifikasi solusi potensial. Populasi awal yang telah diinisialisasi dievaluasi menggunakan *fitness function*, kemudian diseleksi sebagai *parent*, lalu dilakukan *crossover* untuk mengkombinasikan dua *parent* untuk menghasilkan anak. State anak ini kemudian dimutasi untuk menghasilkan solusi baru yang mungkin lebih baik.

- **Kelebihan:** Memungkinkan penjelajahan ruang solusi secara global tidak hanya solusi tunggal, termasuk mengatasi optimum lokal. Algoritma ini juga mampu evaluasi secara paralel.
- **Kekurangan:** Lambat mencapai solusi terbaik jika mutasi terlalu sering. Kinerja algoritma ini juga bergantung pada ketepatan skema seleksi dan probabilitas mutasi.

Proses penyelesaian Diagonal Magic Cube dengan Genetic Algorithm adalah sebagai berikut, di bawah ini.

- 1) Tentukan populasi awal dengan menginisialisasi secara acak beberapa magic cube dengan susunan angka 1 hingga 5^3 .
- 2) Selanjutnya, nilai populasi tersebut menggunakan fitness function.
- 3) Seleksi parent (induk) menggunakan **roulette wheel selection**, yang mana individu dalam populasi yang memiliki nilai fitness lebih tinggi memiliki peluang lebih besar untuk dipilih sebagai parent.
- 4) Lakukan crossover (persilangan) dua parent yang. Pada proses ini, satu titik crossover dipilih secara acak, kemudian bagian awal dari satu parent digabungkan dengan bagian akhir dari parent lainnya. Ini akan menghasilkan dua anak baru (offspring).
- 5) Dengan melihat probabilitas mutasi, lakukan mutasi pada offspring.
- 6) Gantikan populasi sebelumnya dengan offspring yang dihasilkan dari proses crossover dan mutasi.
- 7) Ulangi proses ini untuk sebanyak generasi yang ditetapkan.
- 8) Ambil individu dengan fitness tertinggi sebagai solusi.

Fungsi/Kelas Umum untuk Genetic Algorithm

```
import numpy as np
import random

from Algorithms.strategy import AlgorithmStrategy
```

```

from cube import MagicCube

class GeneticAlgorithm(AlgorithmStrategy):
    def execute(self, magic_cube, population_size, generations,n):
        population = [MagicCube(n) for _ in range(population_size)]
        best_individual = None
        best_fitness= float('-inf')
        best_obj_function = float('-inf')
        iterations = []
        scores = []
        avg_fitness = []

        for i in range(generations):
            sumofObjectiveFunctions = self._getSumOfObjectiveFunctions(population)
            fitness_scores = self._get_fitness_array(population)

            # Find best individual in current generation
            current_best_fitness = max(fitness_scores)
            current_best_individual = population[np.argmax(fitness_scores)]
            iterations.append((current_best_individual(cube.copy(),
            current_best_individual.getCurrentScore(),current_best_fitness))

            if current_best_fitness > best_fitness:
                best_fitness = current_best_fitness
                best_individual = current_best_individual(cube.copy())
                best_obj_function = current_best_individual.getCurrentScore()

            scores.append(best_fitness)
            avg_fitness.append(np.mean(fitness_scores))

            new_population = []
            for _ in range(population_size // 2):
                parent1 = self._roulette_wheel_selection(population, fitness_scores)
                parent2 = self._roulette_wheel_selection(population, fitness_scores)
                child1,child2 = self._ordered_crossover(parent1,parent2)

                self._mutate(child1)
                self._mutate(child2)

                new_population.extend([child1,child2])
            population = new_population

            # Final results
            print("\nFinal Results:")
            print(f"Best Individual Fitness: {best_fitness}")
            print(f"Best Obj Function {best_obj_function}")
            print("Iterations (Best Individual, Fitness):")
            for i, (ind,obj, fit) in enumerate(iterations):
                print(f"Iteration {i + 1}: OBJ = {obj} Fitness = {fit}")

            return best_individual,best_fitness,iterations

    def _getSumOfObjectiveFunctions(self, population):

```

```

        scores = [cube.getCurrentScore() for cube in population]
        return abs(sum(scores))

    def _roulette_wheel_selection(self, population, fitness_scores):
        selected_idx = np.random.choice(len(population), p=fitness_scores)
        return population[selected_idx]

    def _ordered_crossover(self, parent1, parent2):
        n = parent1.n

        # Convert to 1D array
        parent1_flatten = parent1.cube.flatten()
        parent2_flatten = parent2.cube.flatten()

        crossover_start = random.randint(0, len(parent1_flatten) - 2)
        crossover_end = random.randint(crossover_start + 1, len(parent1_flatten) - 1)

        #placeholder tracker mana yg blom diisi
        child1_flatten = [-1] * len(parent1_flatten)
        child2_flatten = [-1] * len(parent2_flatten)

        child1_flatten[crossover_start:crossover_end] =
parent1_flatten[crossover_start:crossover_end]
        child2_flatten[crossover_start:crossover_end] =
parent2_flatten[crossover_start:crossover_end]

        self._fill_remaining_values(child1_flatten, parent2_flatten, 0)
        self._fill_remaining_values(child2_flatten, parent1_flatten, 0)

        child1_cube = np.array(child1_flatten).reshape((n, n, n))
        child2_cube = np.array(child2_flatten).reshape((n, n, n))

        child1 = MagicCube(n)
        child2 = MagicCube(n)
        child1(cube = child1_cube)
        child2(cube = child2_cube)

        child1.initialize_sums()
        child2.initialize_sums()
        child1.update_position_map()
        child2.update_position_map()

        return child1, child2

    def _fill_remaining_values(self, child, source, pos):
        for gene in source:
            if gene not in child:
                while child[pos] != -1:
                    pos += 1
                child[pos] = gene

    def _mutate(self, cube):
        if random.random() < 0.3:

```

```

num1 = random.randint(1, 125)
num2 = random.randint(1, 125)
while num2 == num1:
    num2 = random.randint(1, 125)

cube.swap_number(num1, num2)

def _get_fitness_array(self, population):
    sum_total = sum(abs(cube.getCurrentScore()) for cube in population)

    real_fitness = [sum_total + cube.getCurrentScore() for cube in population]
    fitness_total = sum(real_fitness)

    return [x/fitness_total for x in real_fitness]

```

Penjelasan fungsi:

- `getSumOfObjectiveFunctions`

Menghitung jumlah dari semua skor dalam populasi. Ini digunakan untuk mengetahui total tujuan yang dicapai oleh populasi dan memberikan dasar untuk menghitung fitness dari setiap individu dalam populasi.

- `roulette_wheel_selection`

Memilih satu individu dari populasi berdasarkan kebugarannya menggunakan metode "roulette wheel". Individu dengan nilai fitness lebih tinggi memiliki peluang lebih besar untuk dipilih, sehingga memberikan prioritas pada solusi yang lebih baik dalam proses seleksi.

- `ordered_crossover`

Menghasilkan dua anak dari dua induk dengan metode "ordered crossover". Dalam metode ini, sebagian urutan dari satu induk dipertahankan, sementara urutan lain diambil dari induk kedua, menciptakan kombinasi gen baru pada setiap anak untuk mempertahankan variasi dalam populasi.

- `fill_remaining_values`

Mengisi nilai-nilai yang belum diisi dalam anak hasil crossover dengan nilai dari induk lain, memastikan tidak ada duplikasi dan seluruh elemen dari induk tetap ada dalam anak tersebut.

- `mutate`

Melakukan mutasi pada individu cube dengan kemungkinan tertentu. Dalam mutasi ini, dua angka acak dalam kubus akan ditukar, membantu menciptakan variasi yang lebih besar dalam populasi dan menghindari local optimum dalam proses evolusi.

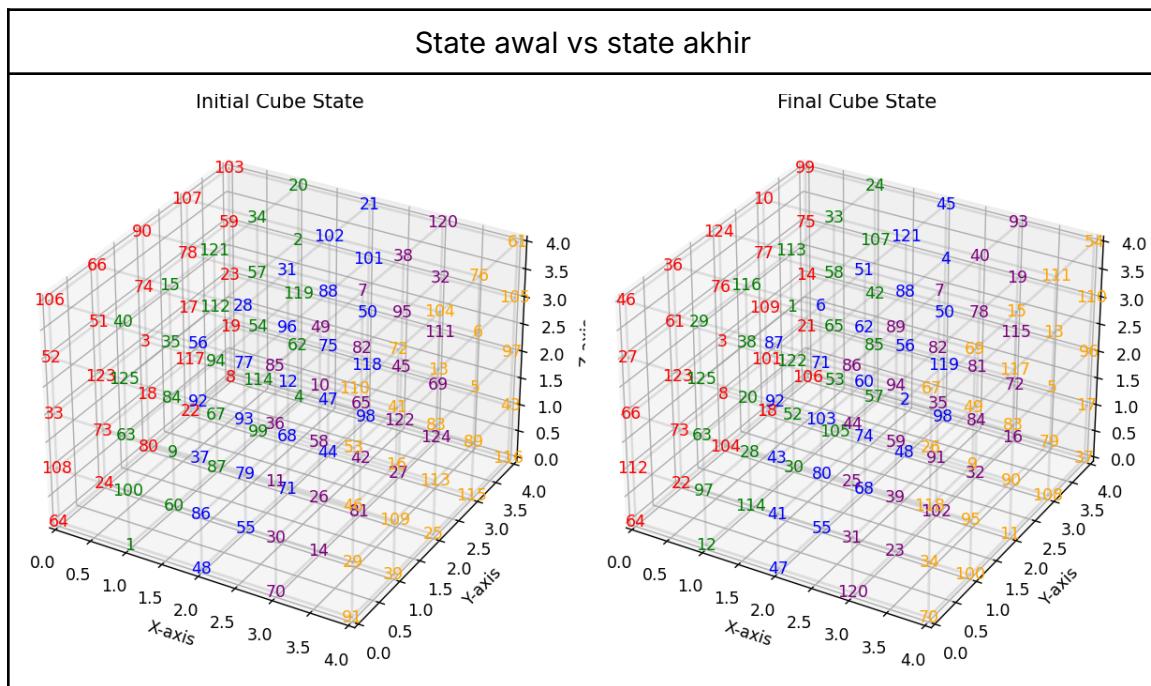
- get_fitness_array

Menghitung nilai fitness (array) untuk setiap individu dalam populasi. Nilai fitness diperoleh berdasarkan jumlah total skor dalam populasi, di mana setiap individu mendapatkan nilai fitness yang relatif dan menentukan peluangnya untuk terpilih dalam proses seleksi.

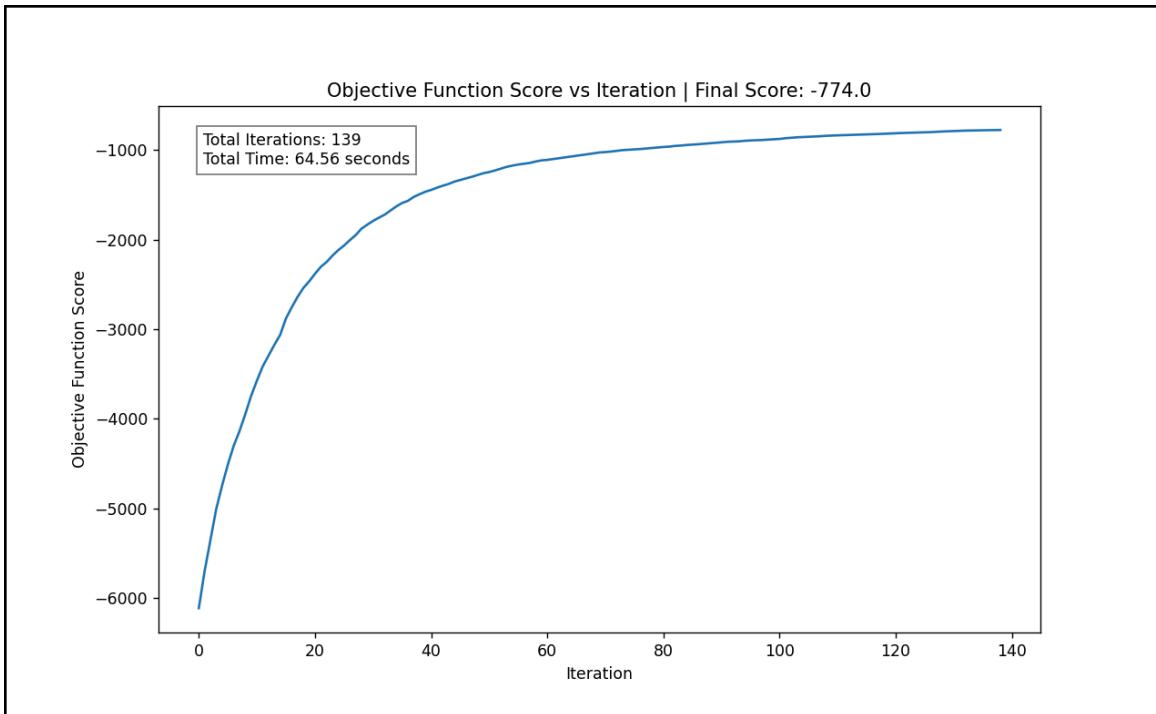
2.3 Hasil Eksperimen dan Analisis Setiap Algoritma

2.3.1 Steepest Ascent Hill-climbing

1. Percobaan 1



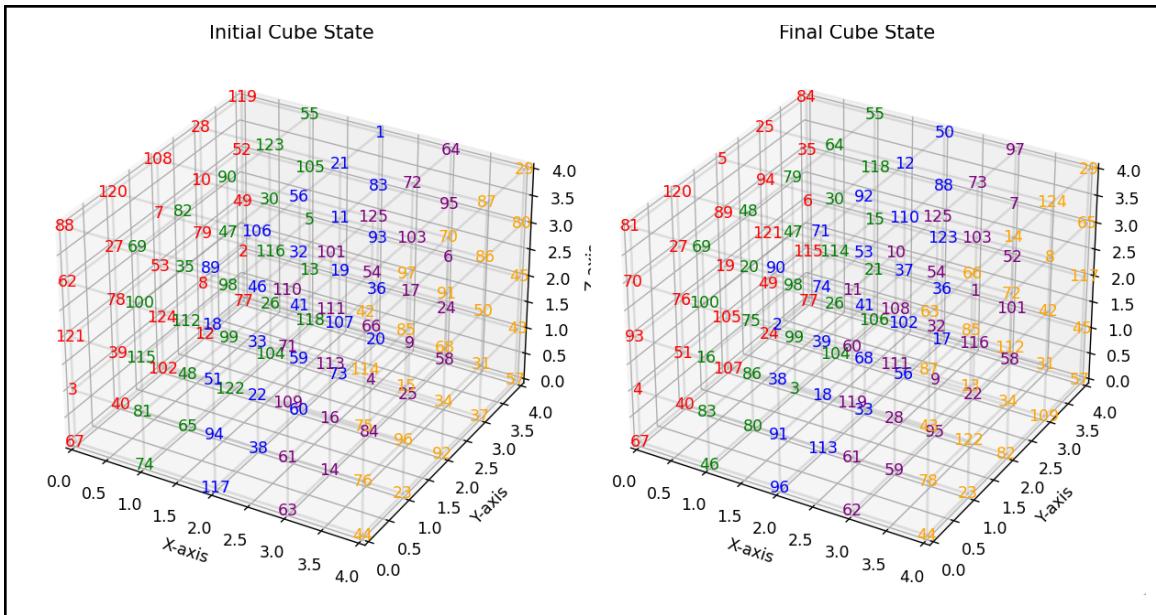
Plot nilai objective function terhadap banyak iterasi yang telah dilewati



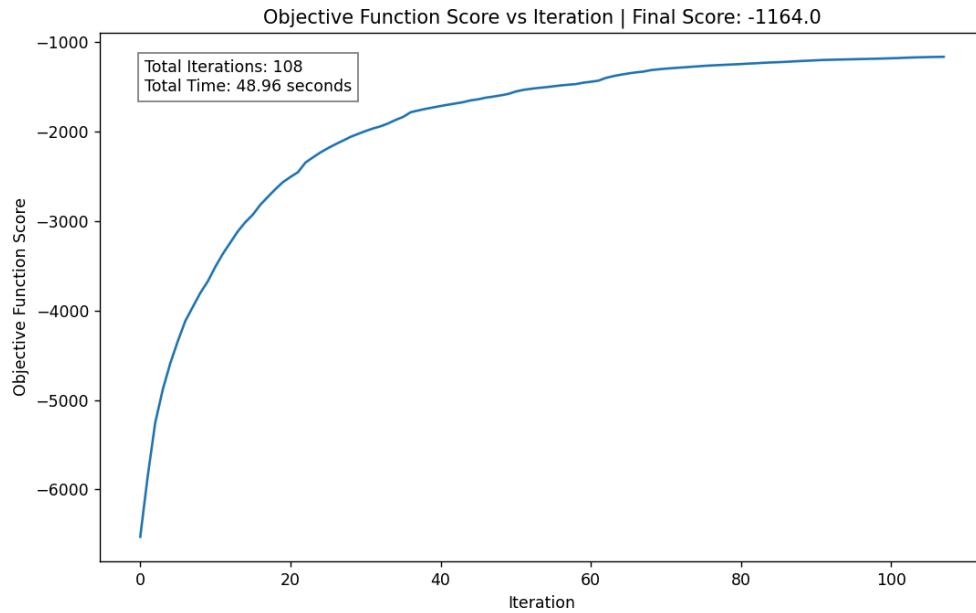
Nilai objective function akhir yang dicapai	-774.0
Durasi proses pencarian	64.56 detik
Banyak iterasi hingga proses pencarian berhenti	139

2. Percobaan 2

State awal vs state akhir

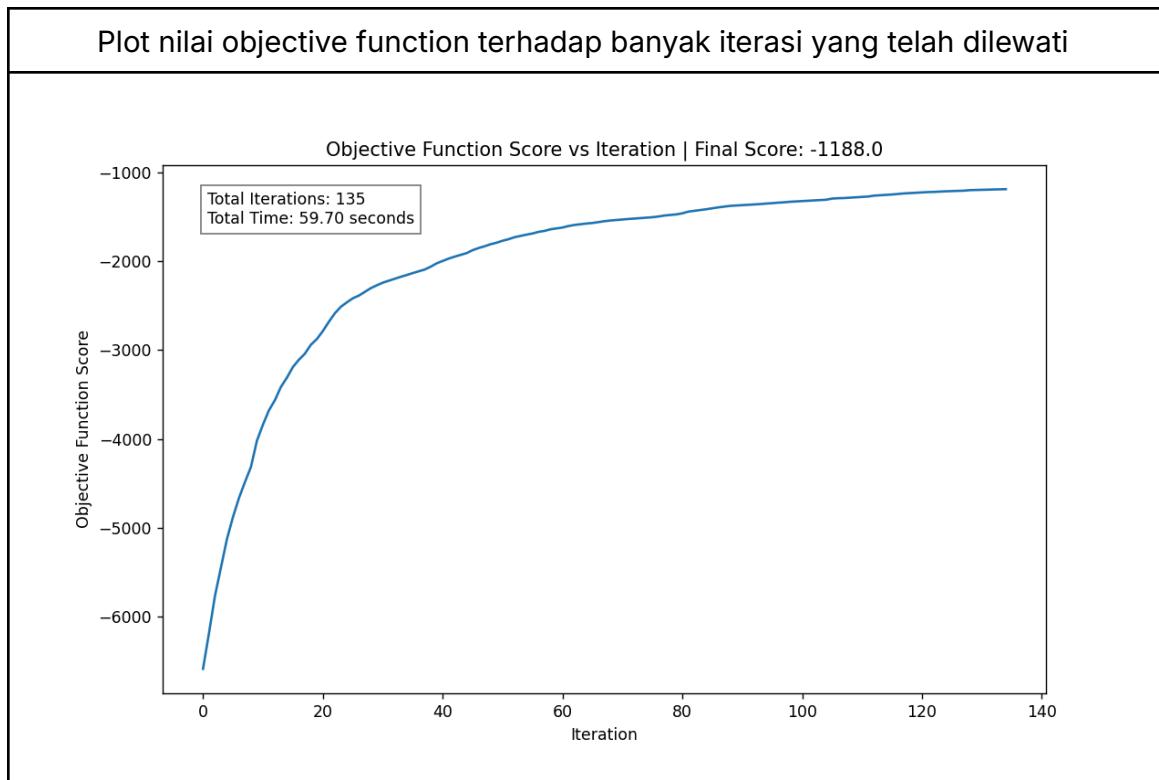
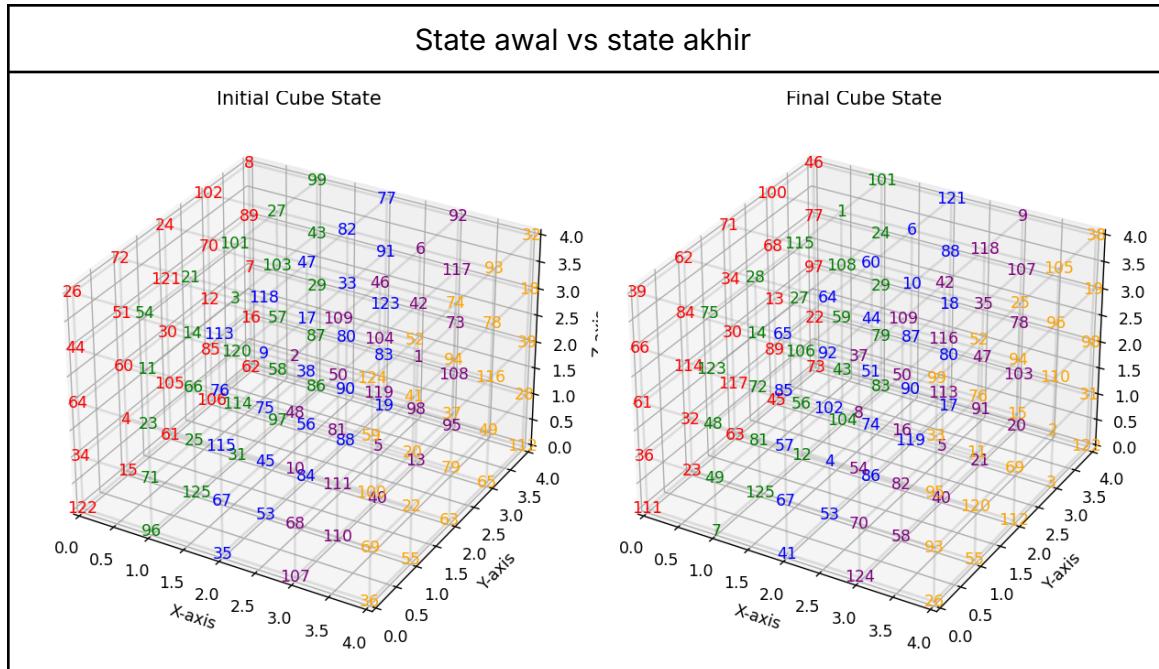


Plot nilai objective function terhadap banyak iterasi yang telah dilewati



Nilai objective function akhir yang dicapai	-1164.0
Durasi proses pencarian	48.96 detik
Banyak iterasi hingga proses pencarian berhenti	108

3. Percobaan 3



Nilai objective function akhir yang dicapai	-1188.0
---	---------

Durasi proses pencarian	59.70 detik
Banyak iterasi hingga proses pencarian berhenti	135

Nilai objective function dalam percobaan ini berkisar antara -774.0 hingga -1188.0, yang cukup jauh dari target global optima (nilai 0). Hal ini menunjukkan bahwa algoritma belum mencapai solusi global optimal dan hanya mencapai solusi lokal optimal. Hal ini dapat terjadi karena steepest ascent cenderung bergerak ke arah perbaikan tertinggi dalam ruang pencarian, yang dapat menyebabkan algoritma terjebak pada solusi lokal. Dalam konteks ini, setiap percobaan cenderung mencapai solusi lokal yang berbeda-beda, namun masih cukup jauh dari solusi global (nilai 0).

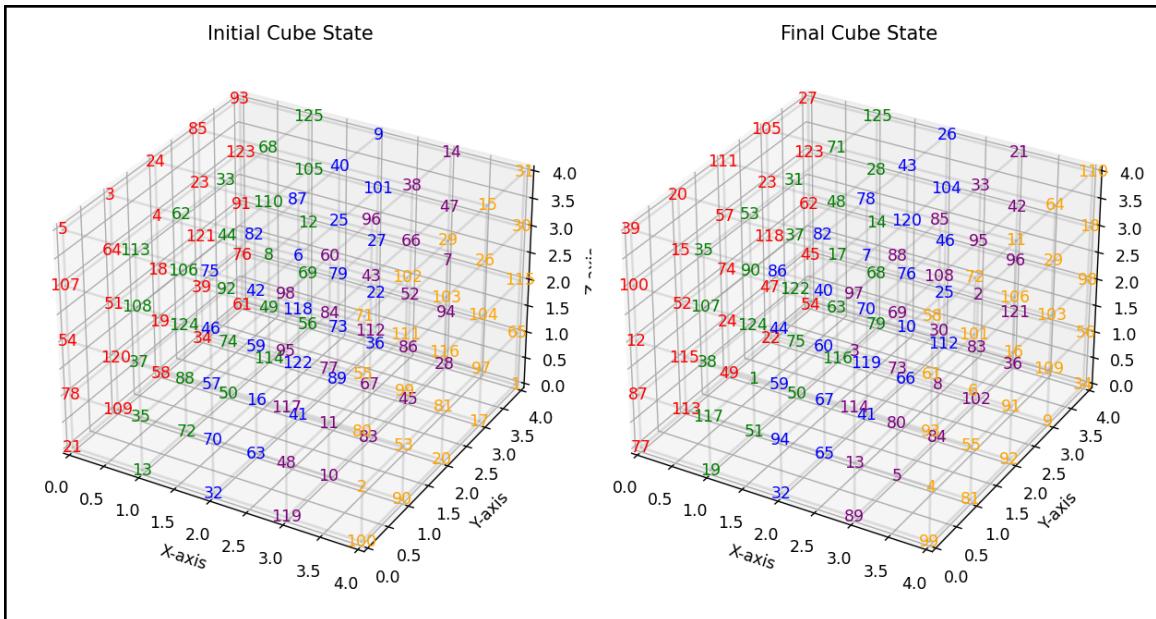
Terdapat perbedaan signifikan antara nilai akhir objective function maksimum dari ketiga percobaan. Percobaan 1 menghasilkan nilai -774.0, sedangkan percobaan 2 dan 3 mencapai nilai yang lebih rendah, yaitu -1164.0 dan -1188.0 (**standar deviasi: 189.76**). Hal ini menunjukkan bahwa hasil metode steepest ascent ini tidak sepenuhnya konsisten dan sangat bergantung pada titik awal atau jalur pencarian yang diambil dalam ruang solusi. Durasi pencarian dan banyaknya iterasi juga menunjukkan variabilitas antar percobaan. Misalnya, percobaan 1 membutuhkan waktu lebih lama (64.56 detik) dan iterasi lebih banyak (139 iterasi) dibandingkan dengan percobaan 2 yang lebih singkat (48.96 detik dan 108 iterasi). Variabilitas ini bisa disebabkan oleh perbedaan kondisi awal yang menyebabkan algoritma melalui jalur berbeda setiap kali dijalankan.

2.3.2 Hill-climbing with Sideways Move

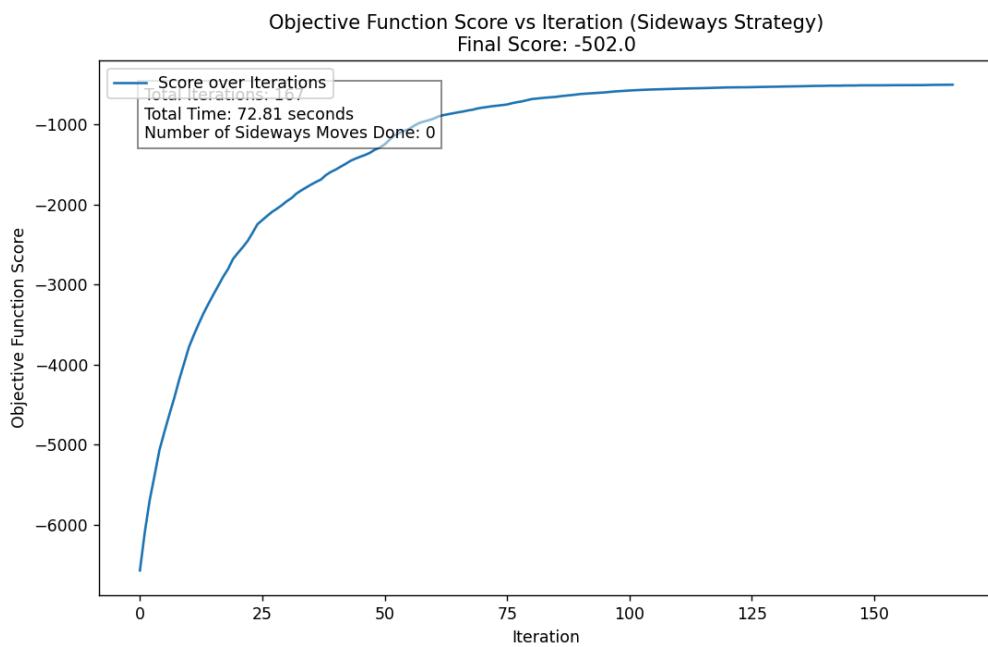
Dengan parameter `max_sideways` diset pada angka 10.

1. Percobaan 1

State awal vs state akhir



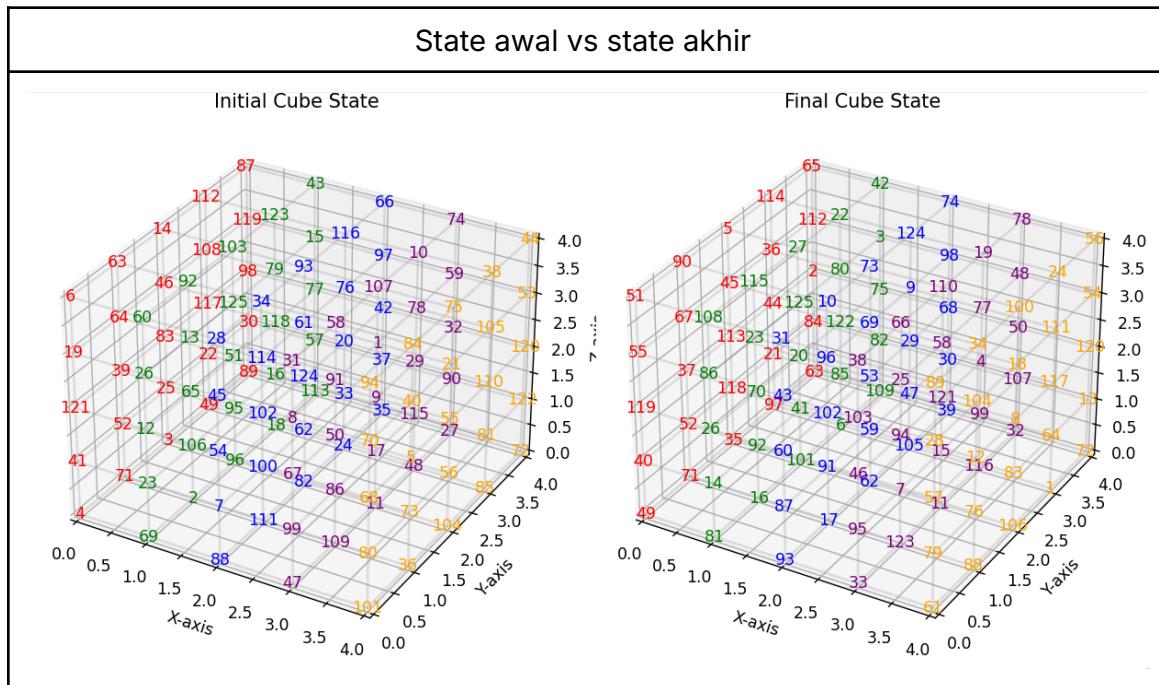
Plot nilai objective function terhadap banyak iterasi yang telah dilewati



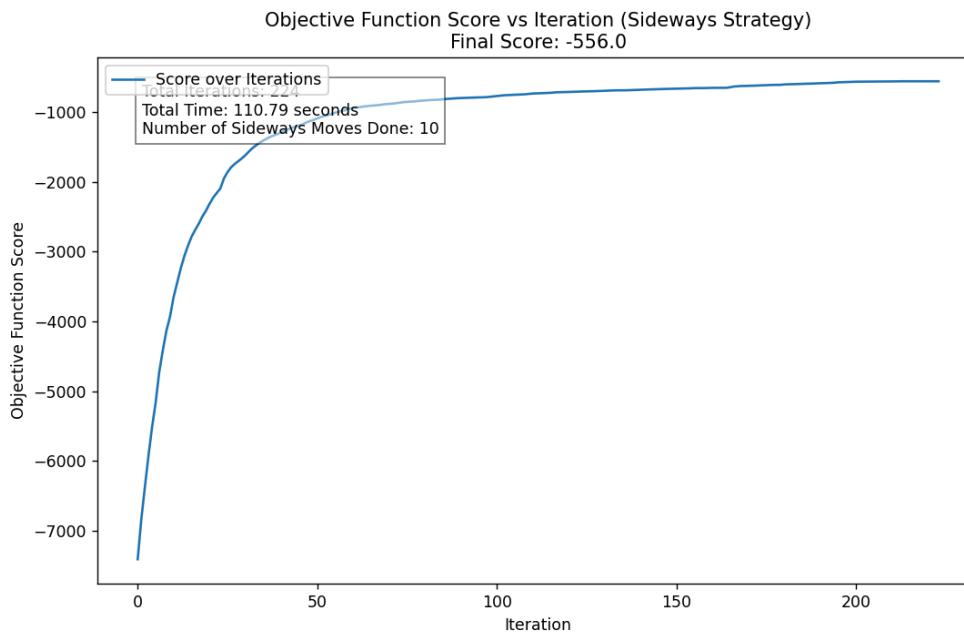
Nilai objective function akhir yang dicapai	-502.0
Durasi proses pencarian	72.81 detik
Banyak iterasi hingga proses pencarian berhenti	167

Jumlah sideways move yang terjadi	0
-----------------------------------	---

2. Percobaan 2

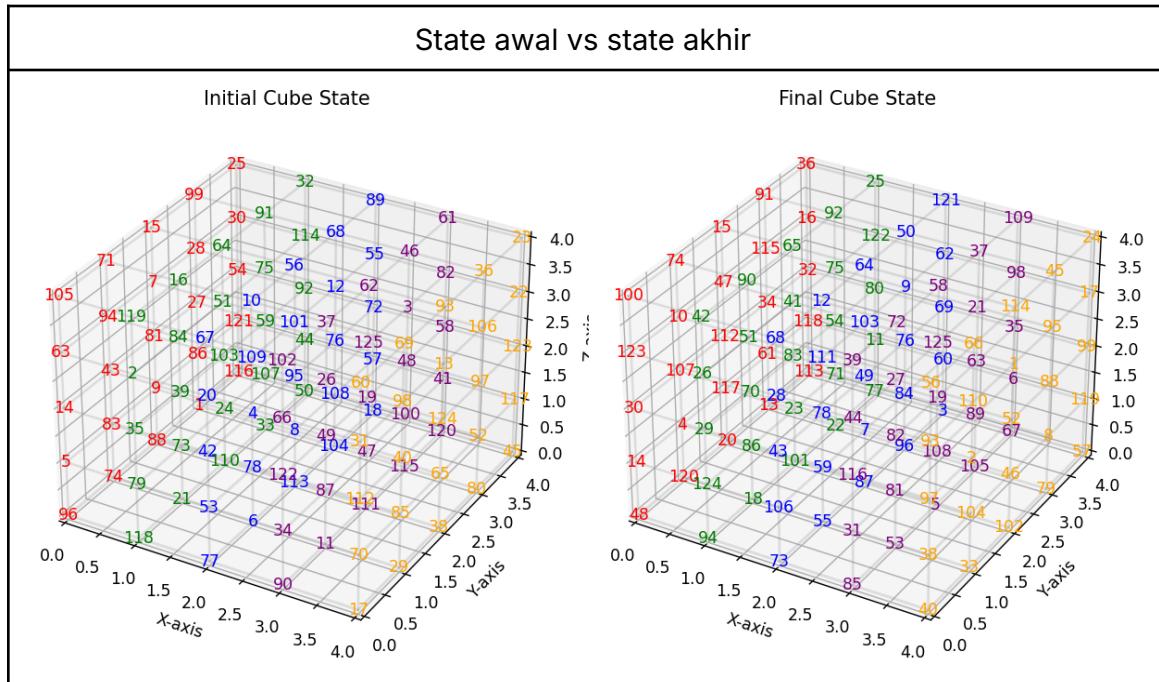


Plot nilai objective function terhadap banyak iterasi yang telah dilewati

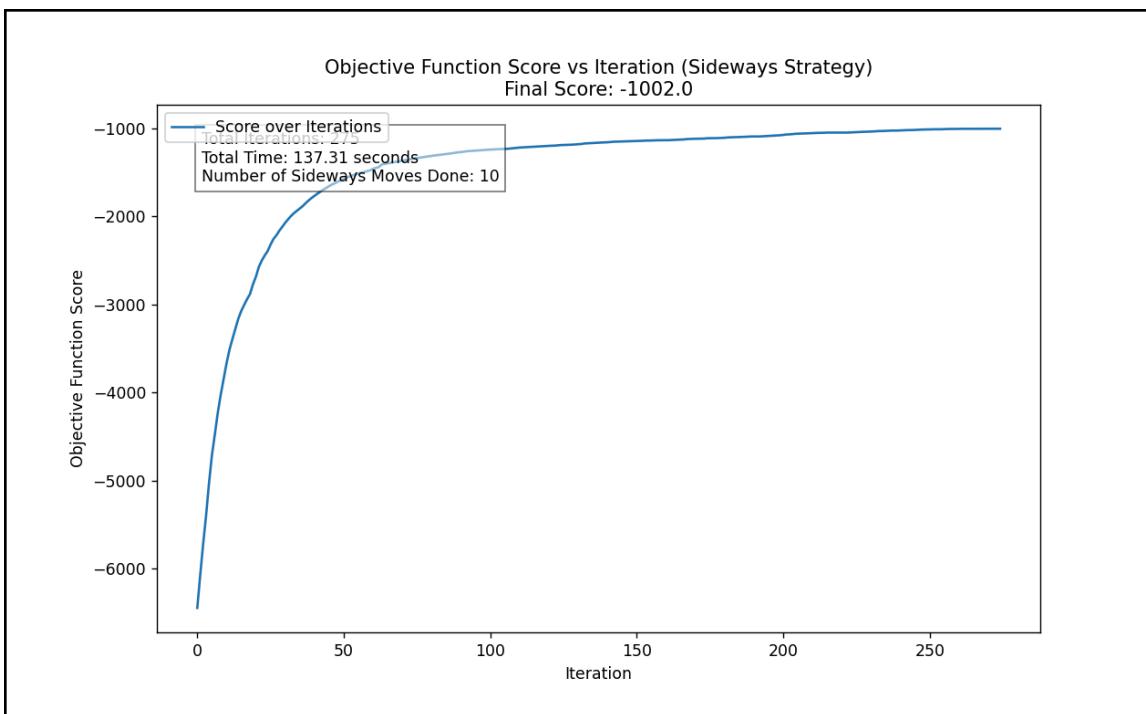


Nilai objective function akhir yang dicapai	-556.0
Durasi proses pencarian	110.79 detik
Banyak iterasi hingga proses pencarian berhenti	224
Jumlah sideways move yang terjadi	10

3. Percobaan 3



Plot nilai objective function terhadap banyak iterasi yang telah dilewati



Nilai objective function akhir yang dicapai	-1002.0
Durasi proses pencarian	137.31 detik
Banyak iterasi hingga proses pencarian berhenti	275
Jumlah sideways move yang terjadi	10

Nilai objective function akhir berkisar antara -502.0 hingga -1002.0, masih cukup jauh dari target global optimal, yaitu nilai 0. Hal ini dapat terjadi karena perjalanan "sideway" tidak selalu terjadi dan bergantung pada nilai objective function tetangga, dimana dalam percobaan ini, hanya percobaan kedua dan ketiga yang menggunakan sideways move, yaitu 10 kali masing-masing, padahal fitur sideways move dapat membantu algoritma melewati local optima. Selain itu, karena terdapat variasi yang cukup tinggi dari nilai objective function, sulit untuk mendapatkan nilai yang sama persis dengan current state yang memungkinkan terjadinya sideways move.

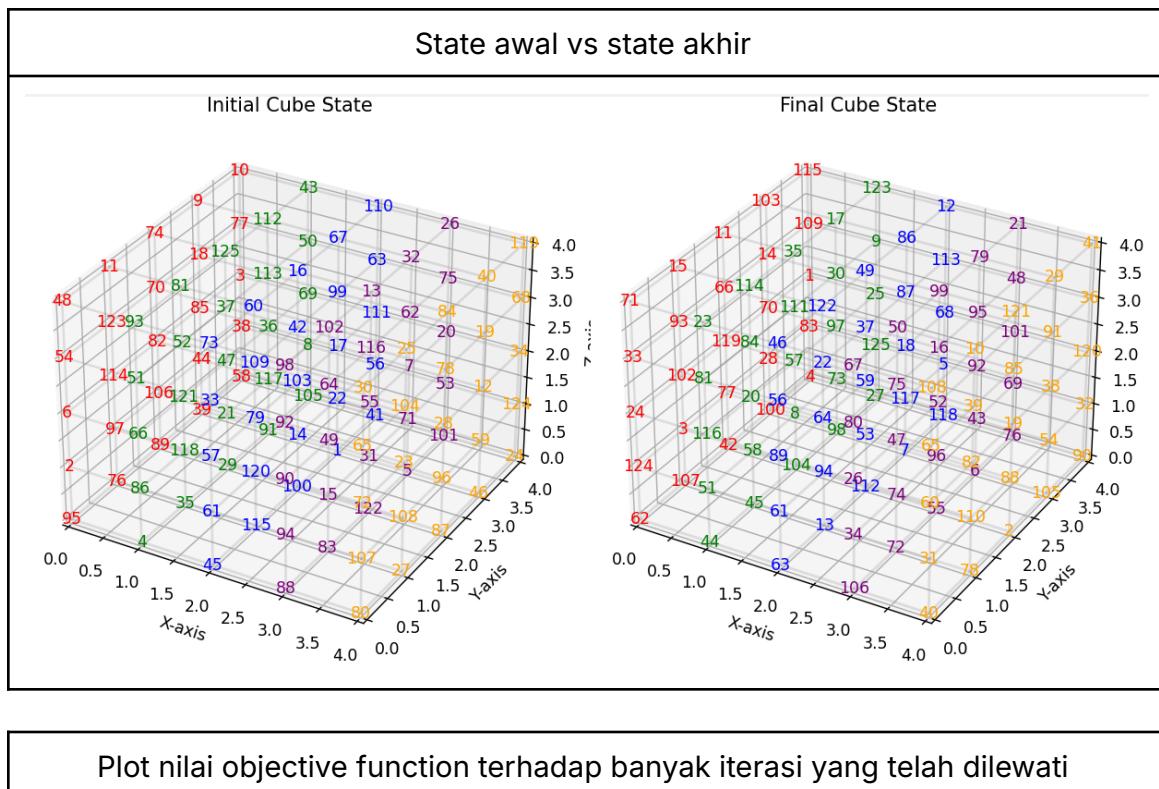
Terdapat variasi yang cukup signifikan pada nilai objective akhir maksimum antara tiga percobaan ini. Percobaan pertama mencapai nilai yang relatif lebih tinggi (-502.0), sedangkan percobaan ketiga mencapai nilai yang lebih rendah (-1002.0) (**standar deviasi: 224.06**). Hal ini menunjukkan bahwa hasil yang dicapai tidak konsisten antar percobaan.

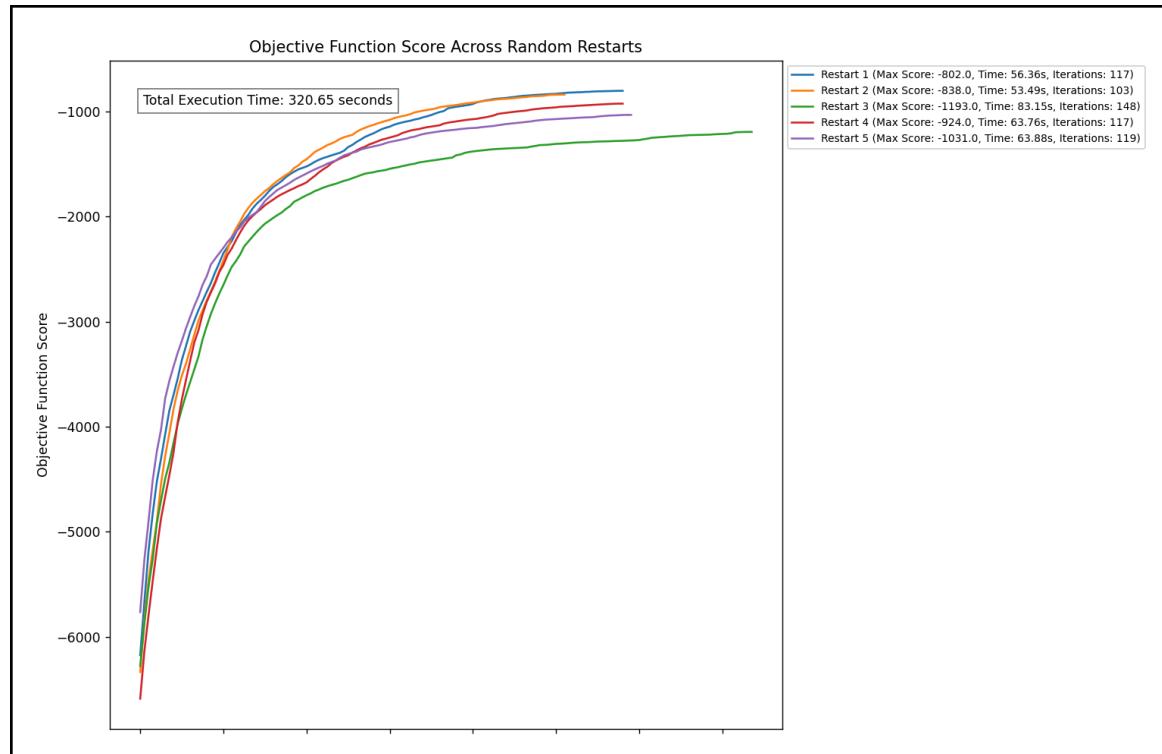
Jumlah iterasi dan durasi pencarian juga menunjukkan perbedaan yang cukup besar antar percobaan. Percobaan pertama selesai lebih cepat dengan 167 iterasi dalam waktu 72.81 detik tanpa sideways move. Percobaan kedua dan ketiga, yang masing-masing menggunakan 10 sideways moves, membutuhkan lebih banyak iterasi dan waktu, masing-masing mencapai 224 dan 275 iterasi serta durasi 110.79 detik dan 137.31 detik. Hal ini mengindikasikan bahwa sideways move berdampak pada penambahan waktu dan iterasi. Dari dua percobaan yang menggunakan sideways move, terlihat bahwa perbedaan hasil tetap ada (nilai akhir -556.0 dan -1002.0), meskipun keduanya menggunakan jumlah sideways move yang sama (10 kali). Ini menunjukkan bahwa efek sideways move bergantung pada titik awal dan tidak selalu berhasil membawa algoritma lebih dekat ke global optimal.

2.3.3 Random Restart Hill-climbing

Dengan parameter **max_restarts** diset pada angka 5.

1. Percobaan 1

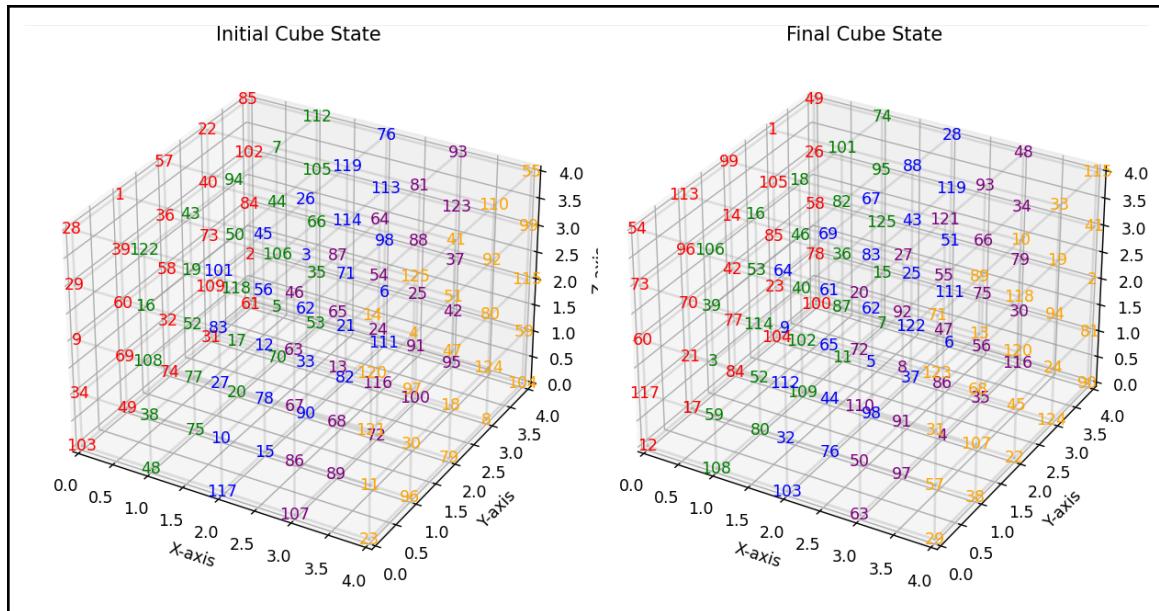




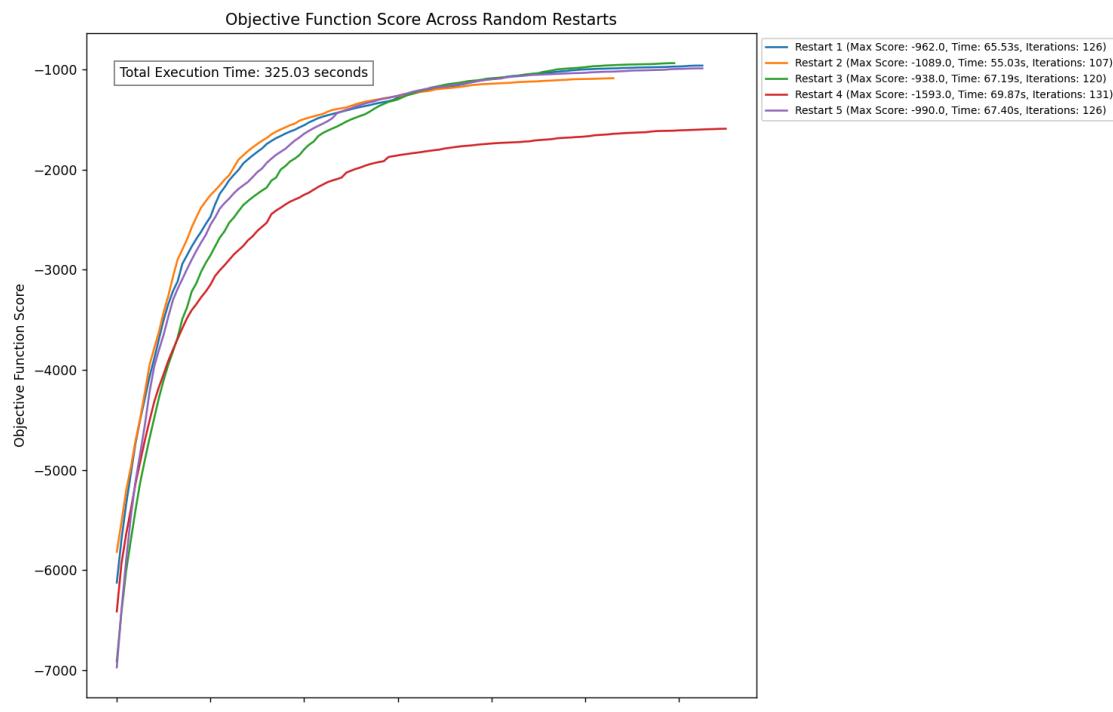
Nilai objective function akhir yang dicapai	-802.0
Durasi proses pencarian	320.65 detik
Banyak restart	5
Banyak iterasi per restart	1. 117 2. 103 3. 148 4. 117 5. 119

2. Percobaan 2

State awal vs state akhir



Plot nilai objective function terhadap banyak iterasi yang telah dilewati



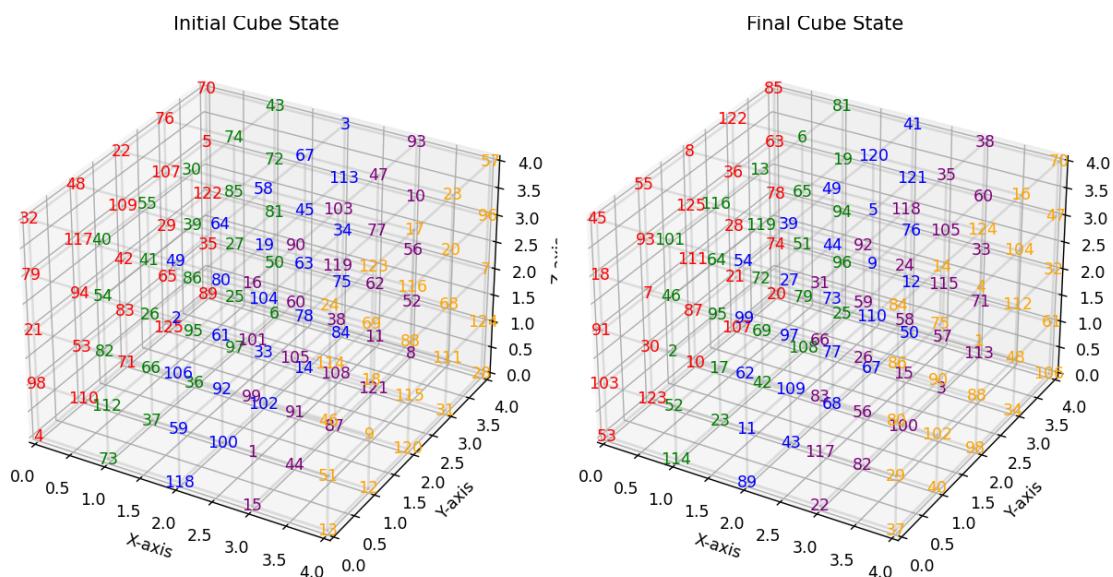
Nilai objective function akhir yang dicapai	-938.0
Durasi proses pencarian	325.03 detik
Banyak restart	5

Banyak iterasi per restart

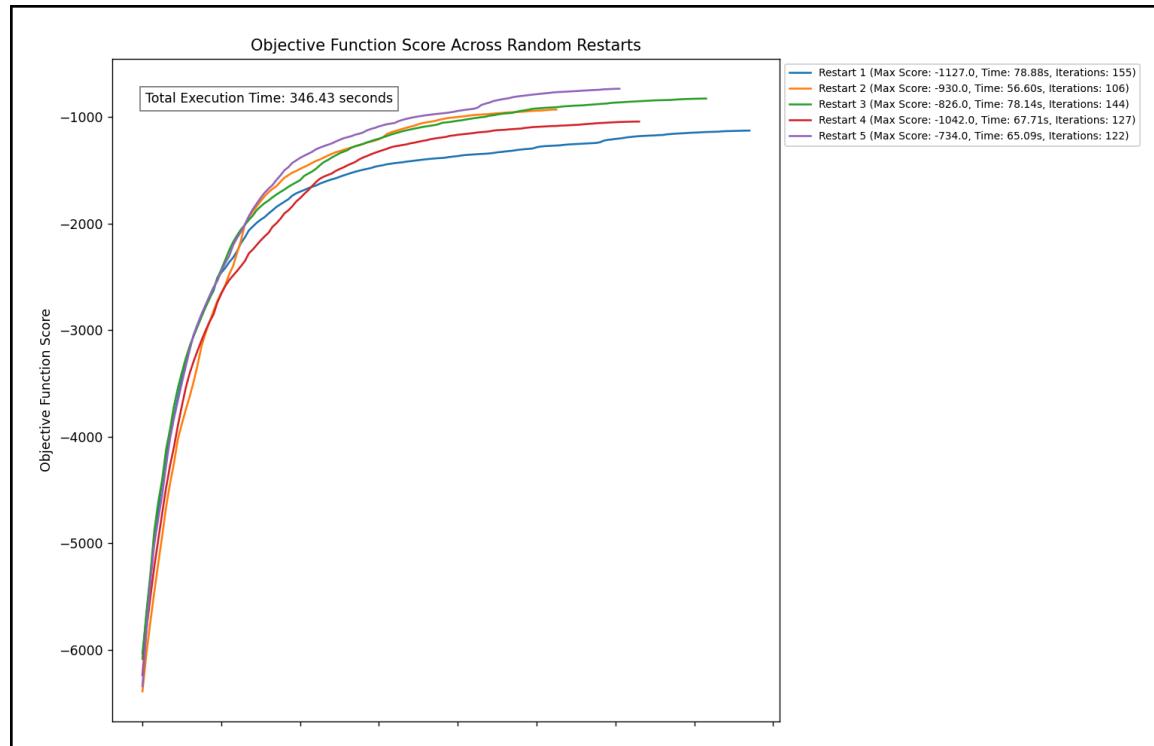
1. 126
2. 107
3. 120
4. 131
5. 126

3. Percobaan 3

State awal vs state akhir



Plot nilai objective function terhadap banyak iterasi yang telah dilewati



Nilai objective function akhir yang dicapai	-734.0
Durasi proses pencarian	346.43 detik
Banyak restart	5
Banyak iterasi per restart	1. 155 2. 106 3. 144 4. 127 5. 122

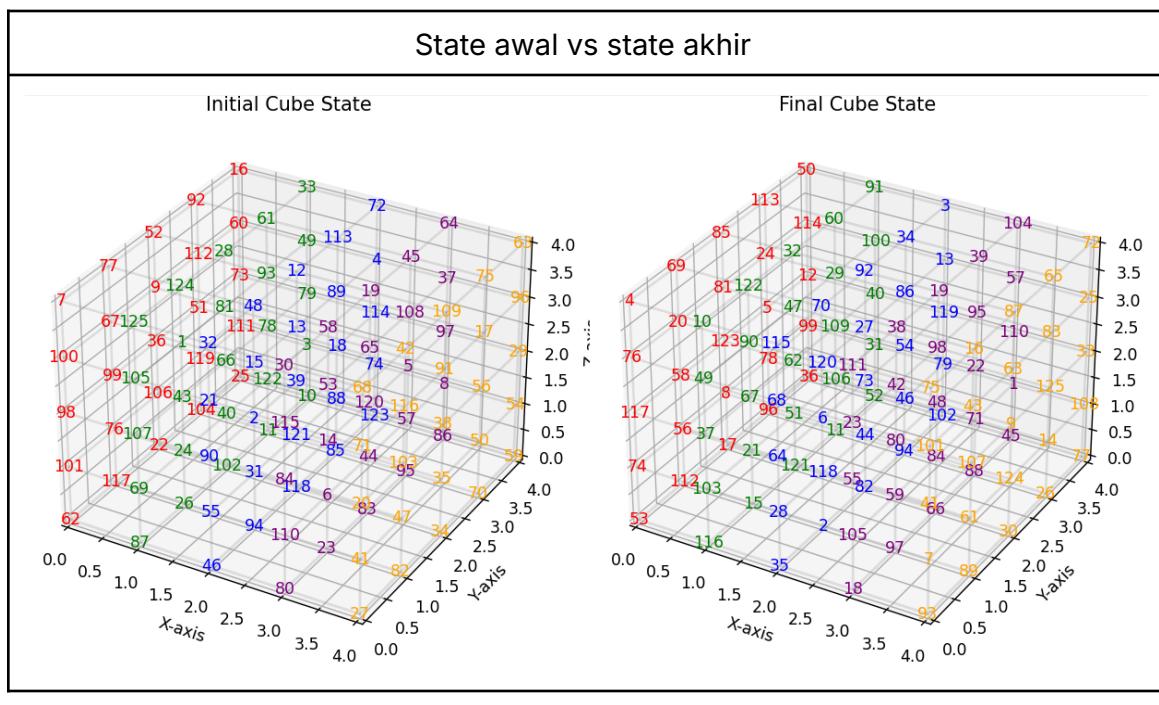
Hasil akhir dari metode random restart hill climbing masih berada cukup jauh dari global optimal (nilai 0), dengan nilai objective function akhir berkisar antara -734.0 hingga -938.0. Random restart memberikan titik awal baru dalam ruang pencarian setiap kali algoritma mengalami restart, yang seharusnya dapat mengurangi risiko terjebak di local optima. Meskipun demikian, hasil percobaan menunjukkan bahwa setiap restart tidak selalu mendekatkan algoritma pada global optimum.

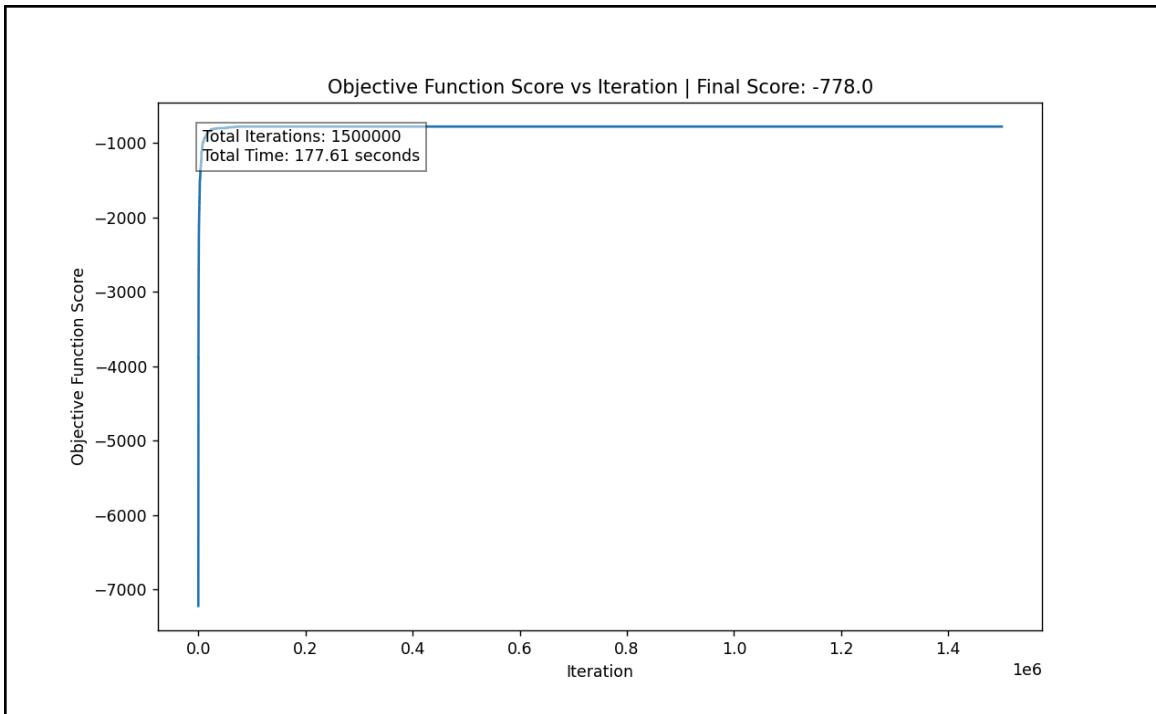
Nilai objective function akhir dari setiap percobaan cukup beragam, dengan nilai -802.0 pada percobaan pertama, -938.0 pada percobaan kedua, dan -734.0 pada percobaan ketiga (**standar deviasi: 84.81**). Hal ini menunjukkan bahwa algoritma menghasilkan hasil

yang bervariasi, yang dapat dipengaruhi oleh variasi titik awal setiap kali restart. Random restart memang bertujuan untuk meningkatkan eksplorasi ruang pencarian, namun hasil ini menunjukkan bahwa algoritma tetap cenderung mendekati solusi suboptimal secara acak. Durasi total pencarian di antara percobaan bervariasi meskipun tidak terlalu jauh berbeda, yaitu antara 320.65 detik hingga 346.43 detik, meskipun jumlah restart konstan sebanyak lima kali. Selain itu, jumlah iterasi per restart bervariasi pada setiap percobaan, yang menunjukkan bahwa waktu pencarian yang diperlukan bergantung pada tetangga yang ditemukan pada setiap iterasi. Waktu komputasi yang cukup lama pada algoritma ini dapat menjadi kelemahan jika banyak restart dilakukan tanpa perbaikan signifikan pada hasil akhir.

2.3.4 Stochastic Hill-climbing

1. Percobaan 1

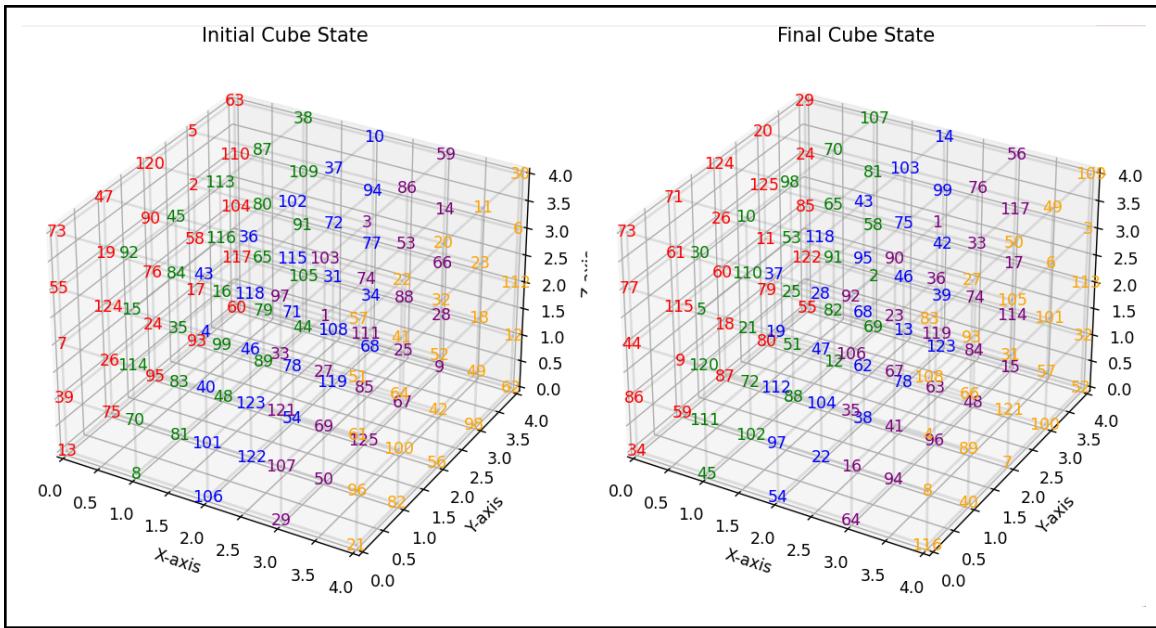




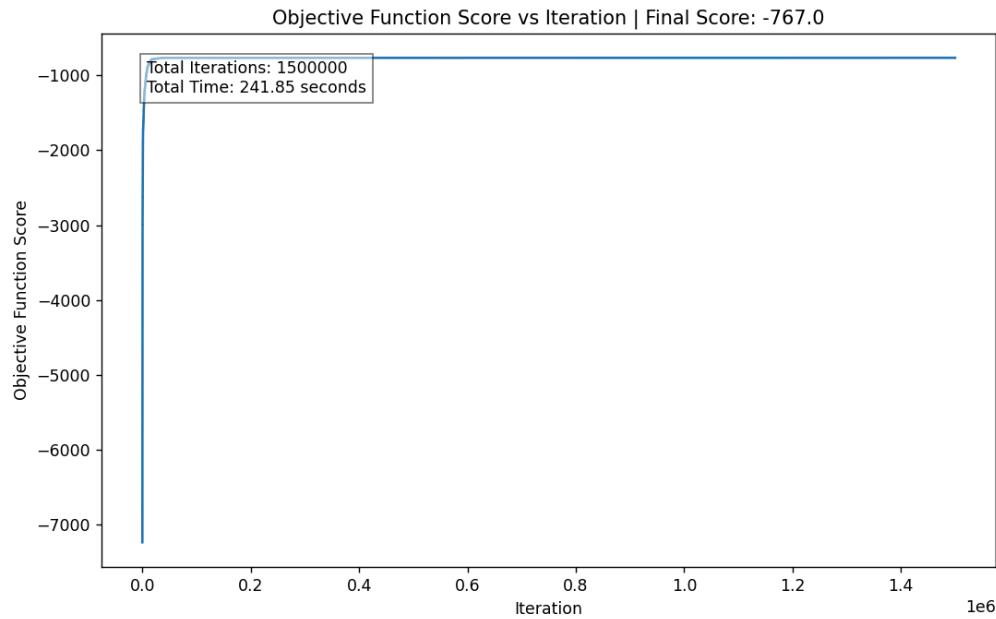
Nilai objective function akhir yang dicapai	-778.0
Durasi proses pencarian	177.61 detik
Banyak iterasi hingga proses pencarian berhenti	1.500.000

2. Percobaan 2

State awal vs state akhir

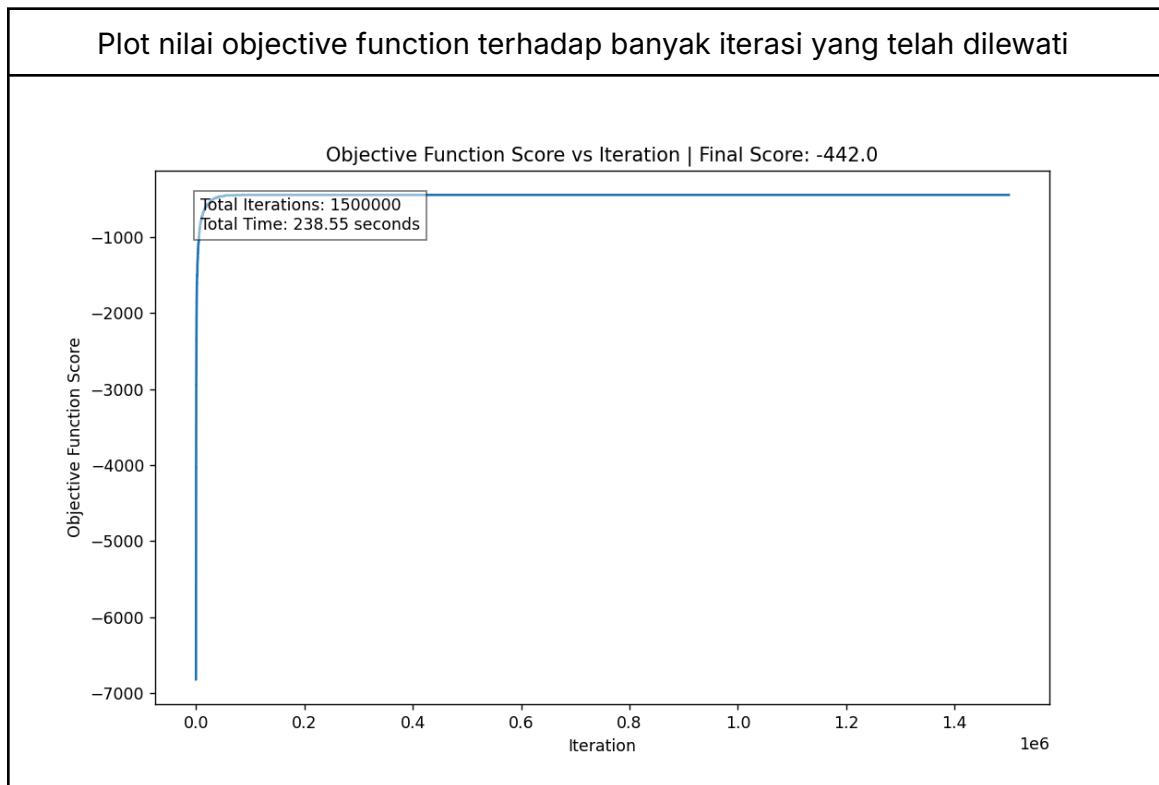
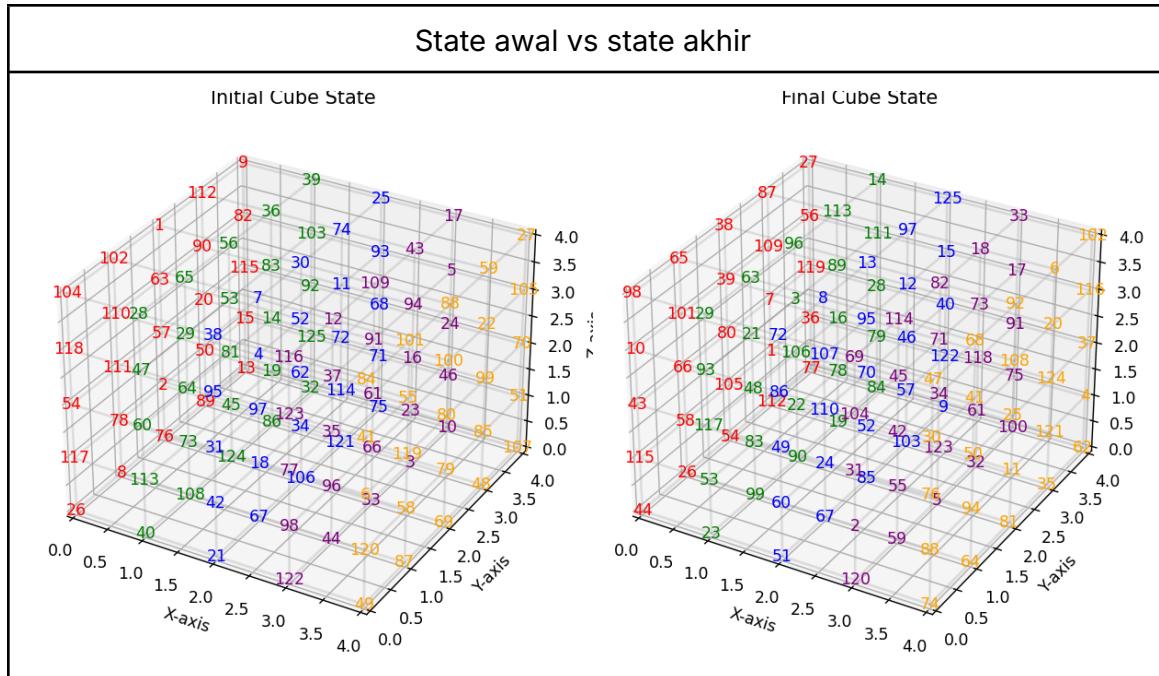


Plot nilai objective function terhadap banyak iterasi yang telah dilewati



Nilai objective function akhir yang dicapai	-767.0
Durasi proses pencarian	241.85 detik
Banyak iterasi hingga proses pencarian berhenti	1.500.000

3. Percobaan 3



Nilai objective function akhir yang dicapai

-442.0

Durasi proses pencarian	238.55 detik
Banyak iterasi hingga proses pencarian berhenti	1.500.000

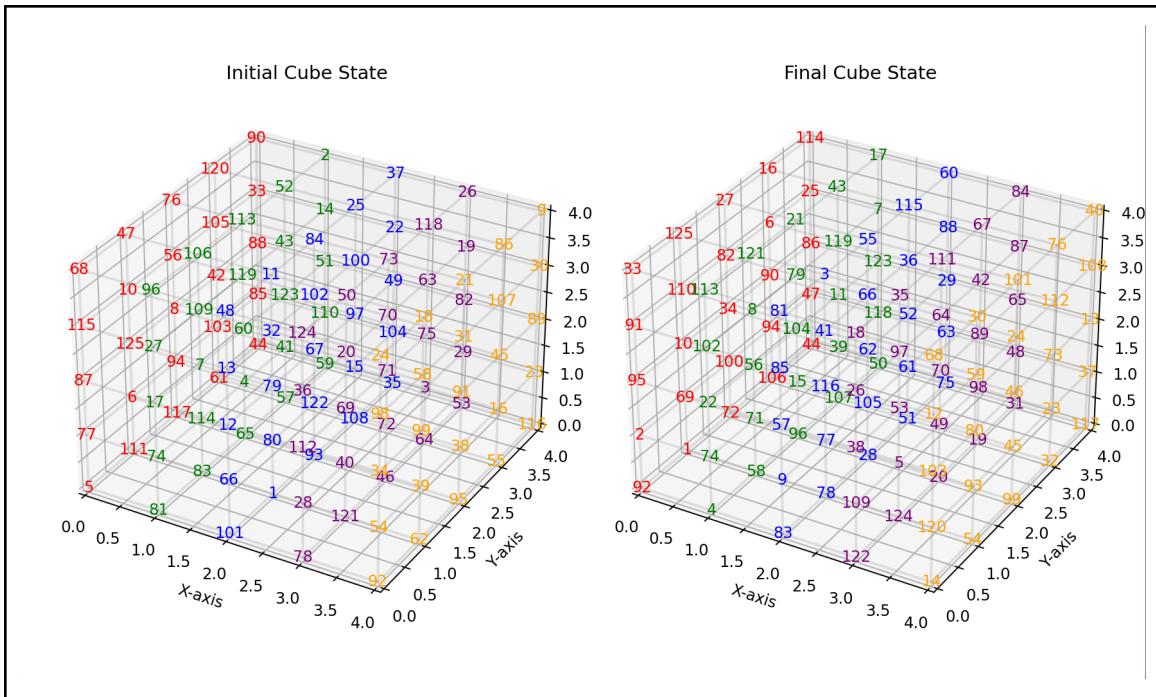
Dari ketiga percobaan, diperoleh hasil dari percobaan ini masih jauh dari nilai 0 (nilai global optima). Hal ini dapat disebabkan karena algoritma hanya memilih satu langkah secara acak, ada kemungkinan besar bahwa langkah yang dipilih tidak selalu membawa algoritma lebih dekat ke nilai optimal. Hal ini berarti bergantung kepada kebetulan dalam memilih langkah perbaikan yang efektif di setiap iterasi.

Dari ketiga percobaan, nilai objective function maksimum yang dicapai bervariasi secara signifikan, yaitu -778.0, -767.0, dan -442.0 (**standar deviasi: 155.86**). Ini menunjukkan adanya variabilitas yang tinggi dalam hasil algoritma stochastic hill climbing. Percobaan ketiga mencapai hasil yang terbaik (-442.0), namun masih jauh dari nilai optimal global (0). Perbedaan hasil ini mengindikasikan bahwa pencarian acak dalam algoritma tidak selalu menghasilkan solusi yang konsisten. Durasi waktu juga berbeda di setiap percobaan, dengan waktu pencarian berkisar antara 177.61 detik hingga 241.85 detik. Semua percobaan mencapai batas iterasi maksimum (1.500.000 iterasi), yang menunjukkan bahwa algoritma tidak menemukan perbaikan lebih lanjut sebelum batas ini tercapai, meskipun percobaan memiliki hasil akhir yang berbeda.

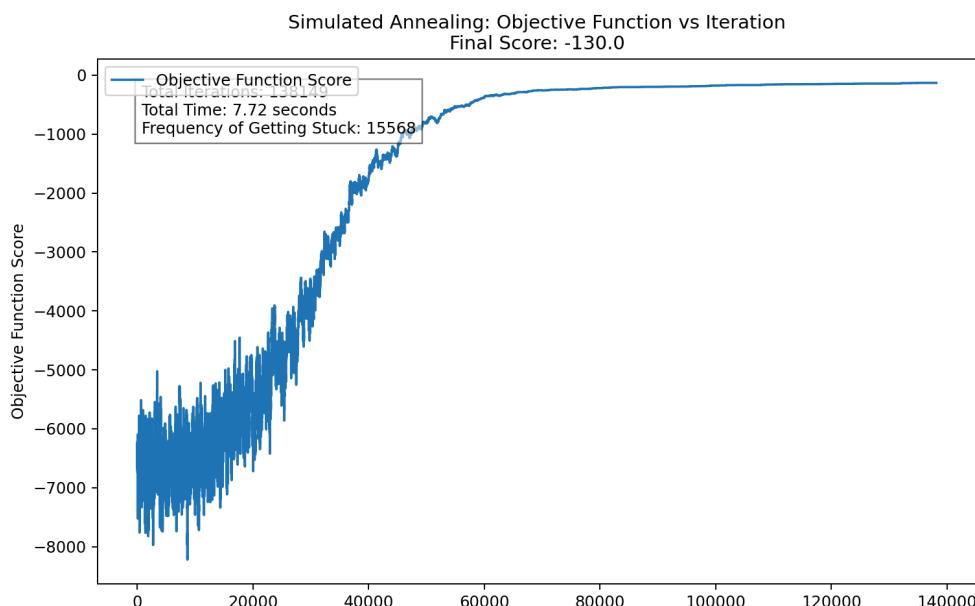
2.3.5 Simulated Annealing

1. Percobaan 1

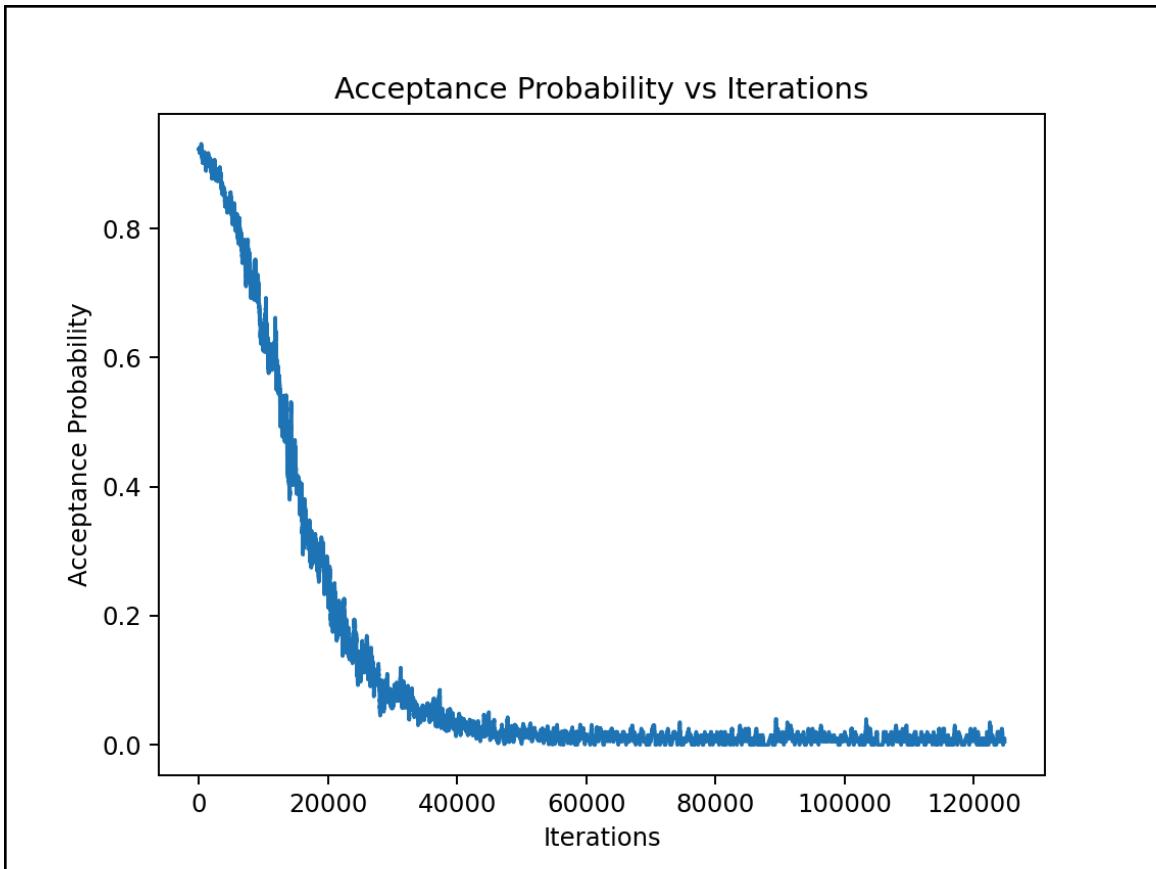
State awal vs state akhir



Plot nilai objective function terhadap banyak iterasi yang telah dilewati



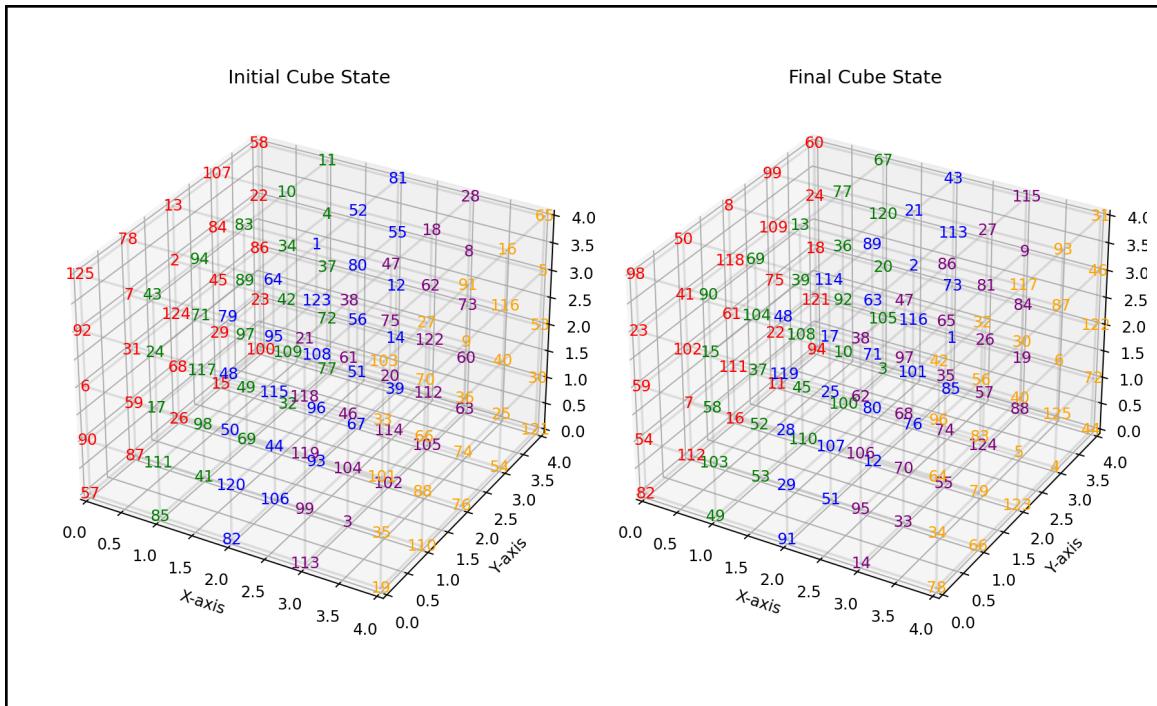
Plot nilai acceptance probability terhadap banyak iterasi yang telah dilewati



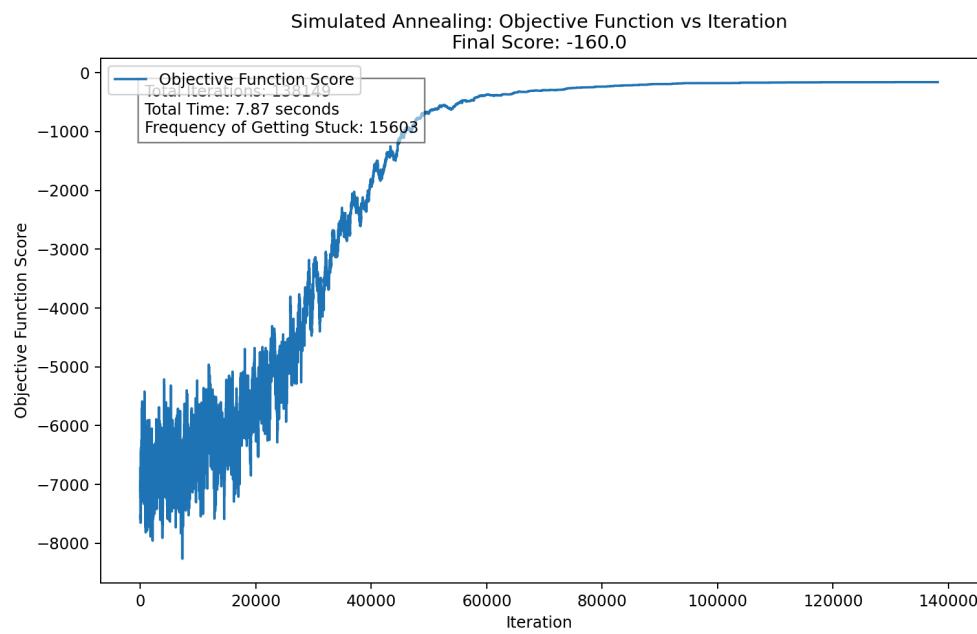
Nilai objective function akhir yang dicapai	-130.0
Durasi proses pencarian	7.72 detik
Banyak iterasi hingga proses pencarian berhenti	138149
Banyak terjebak pada lokal optima	15568

2. Percobaan 2

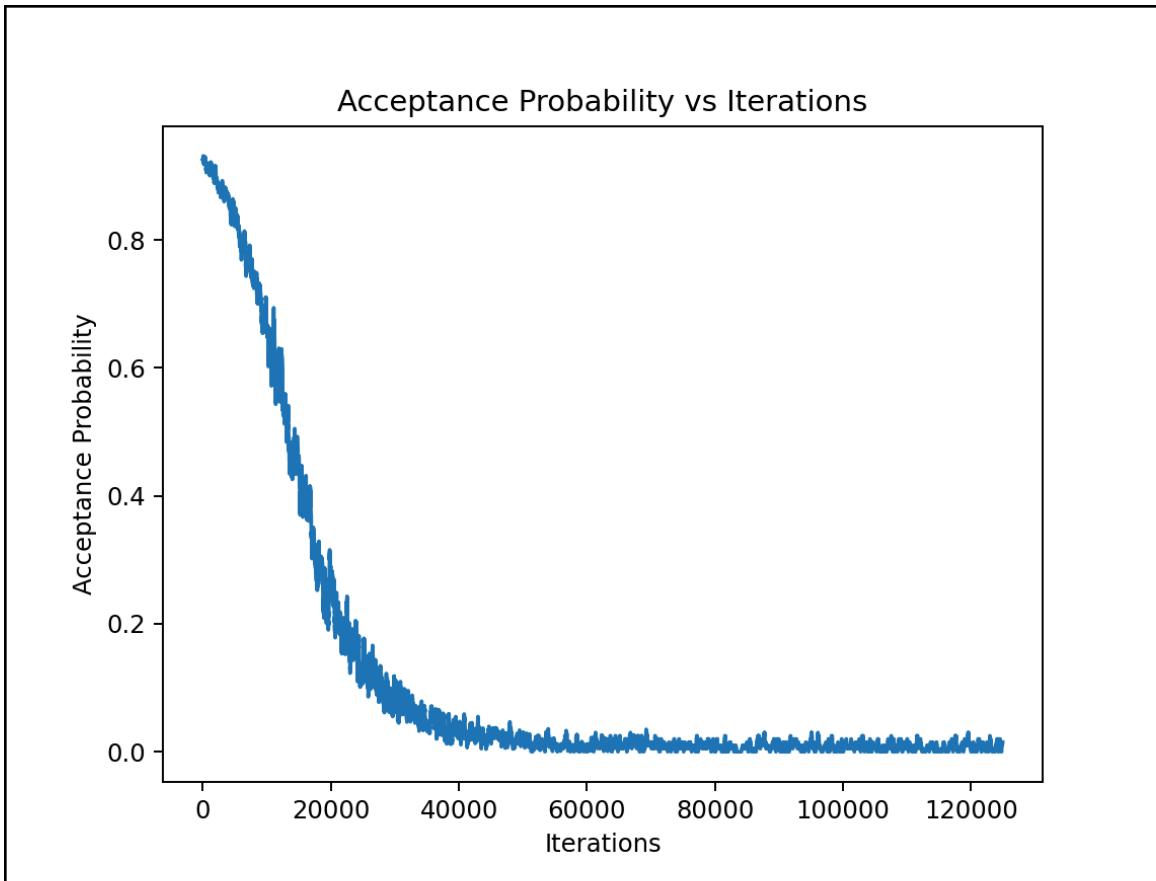
State awal vs state akhir



Plot nilai objective function terhadap banyak iterasi yang telah dilewati



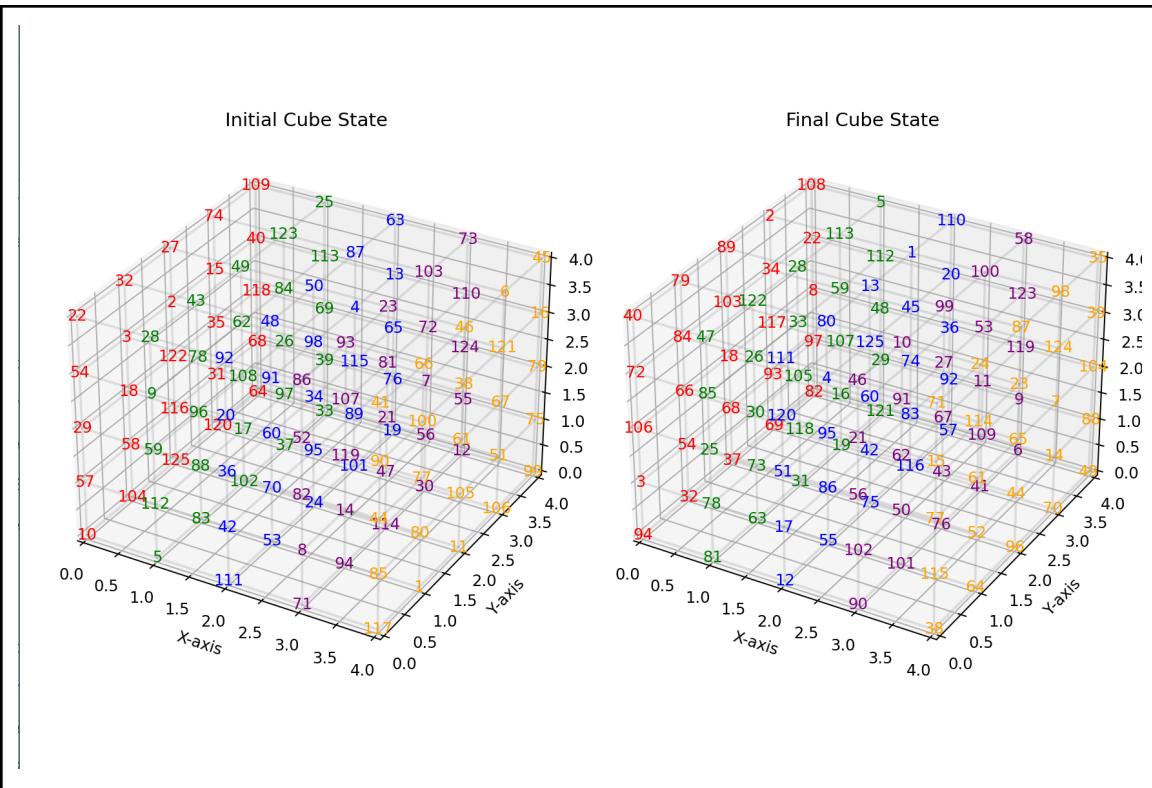
Plot nilai acceptance probability terhadap banyak iterasi yang telah dilewati



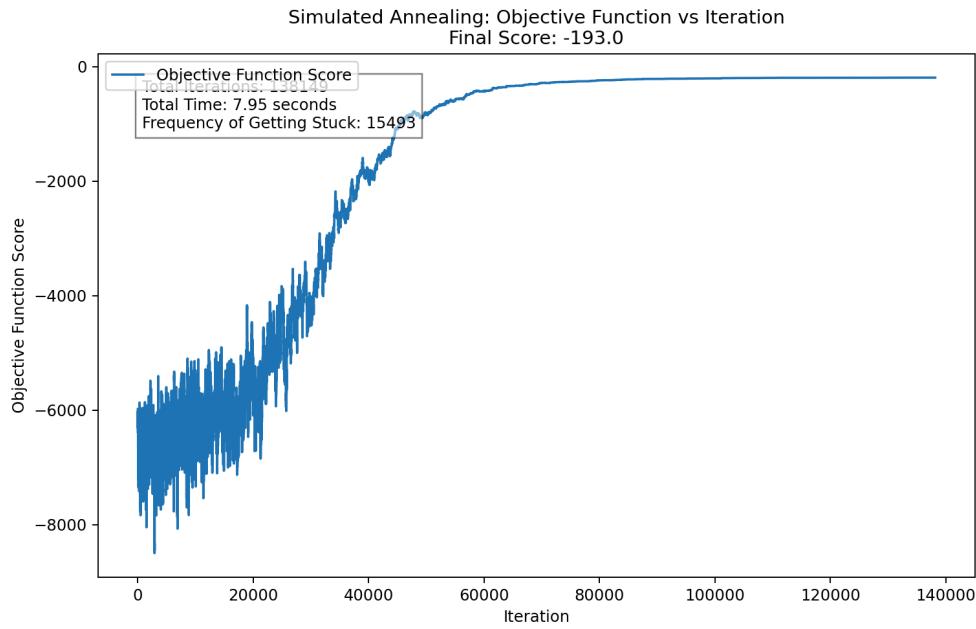
Nilai objective function akhir yang dicapai	-160.0
Durasi proses pencarian	7.87 detik
Banyak iterasi hingga proses pencarian berhenti	138149
Banyak terjebak pada lokal optima	15603

3. Percobaan 3

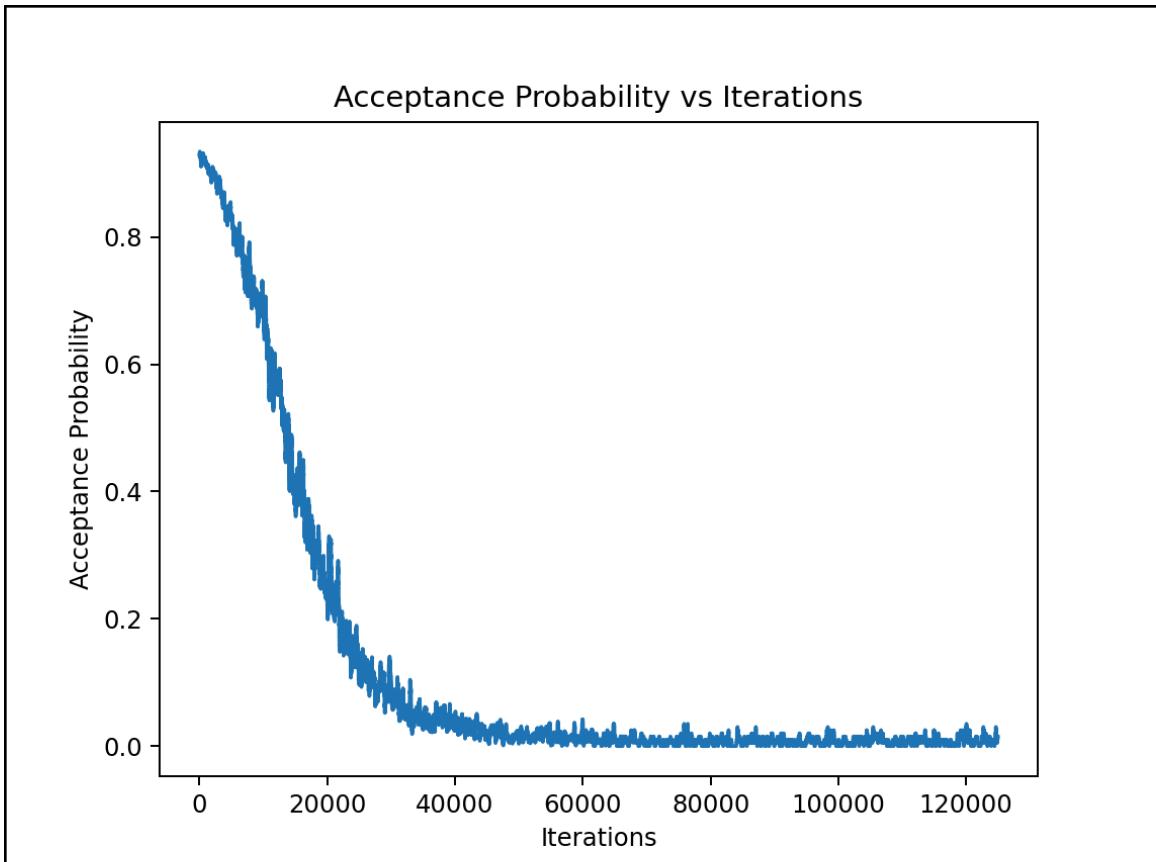
State awal vs state akhir



Plot nilai objective function terhadap banyak iterasi yang telah dilewati



Plot nilai acceptance probability terhadap banyak iterasi yang telah dilewati



Nilai objective function akhir yang dicapai	-193.0
Durasi proses pencarian	7.95 detik
Banyak iterasi hingga proses pencarian berhenti	138149
Banyak terjebak pada lokal optima	15493

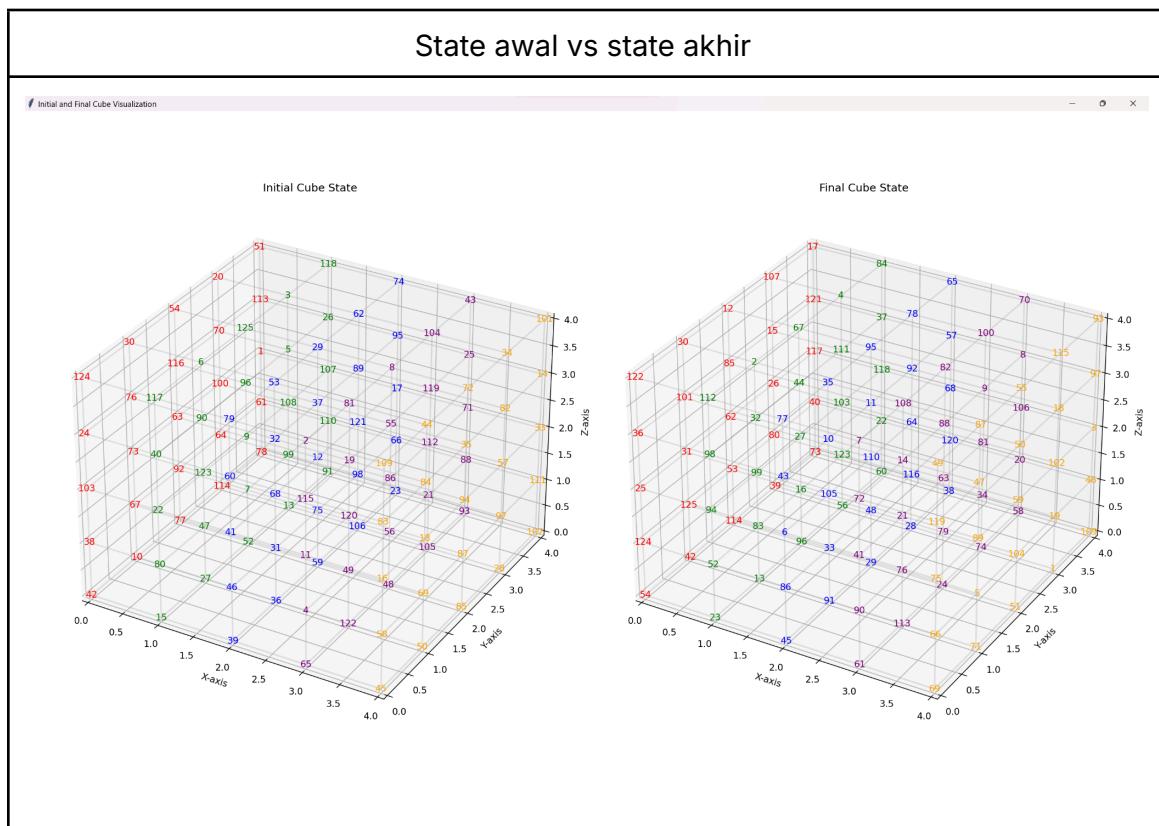
Metode simulated annealing berhasil menghasilkan nilai akhir objective function yang mendekati optimal global dibandingkan dengan metode lain yang telah diuji sebelumnya. Hasil akhir berkisar antara -130.0 hingga -193.0, yang menunjukkan bahwa algoritma ini berhasil mencapai solusi yang lebih baik dan lebih dekat dengan global optima (nilai 0). Hal ini dapat disebabkan oleh cooling schedule yang membantu menghindari local optima.

Hasil akhir dari metode ini juga cukup konsisten, dengan variasi dari -130.0 pada percobaan pertama, -160.0 pada percobaan kedua, dan -193.0 pada percobaan ketiga. Ini menunjukkan bahwa algoritma simulated annealing memberikan hasil yang relatif stabil dan cukup dekat satu sama lain, meskipun hasil akhirnya sedikit berbeda di setiap percobaan.

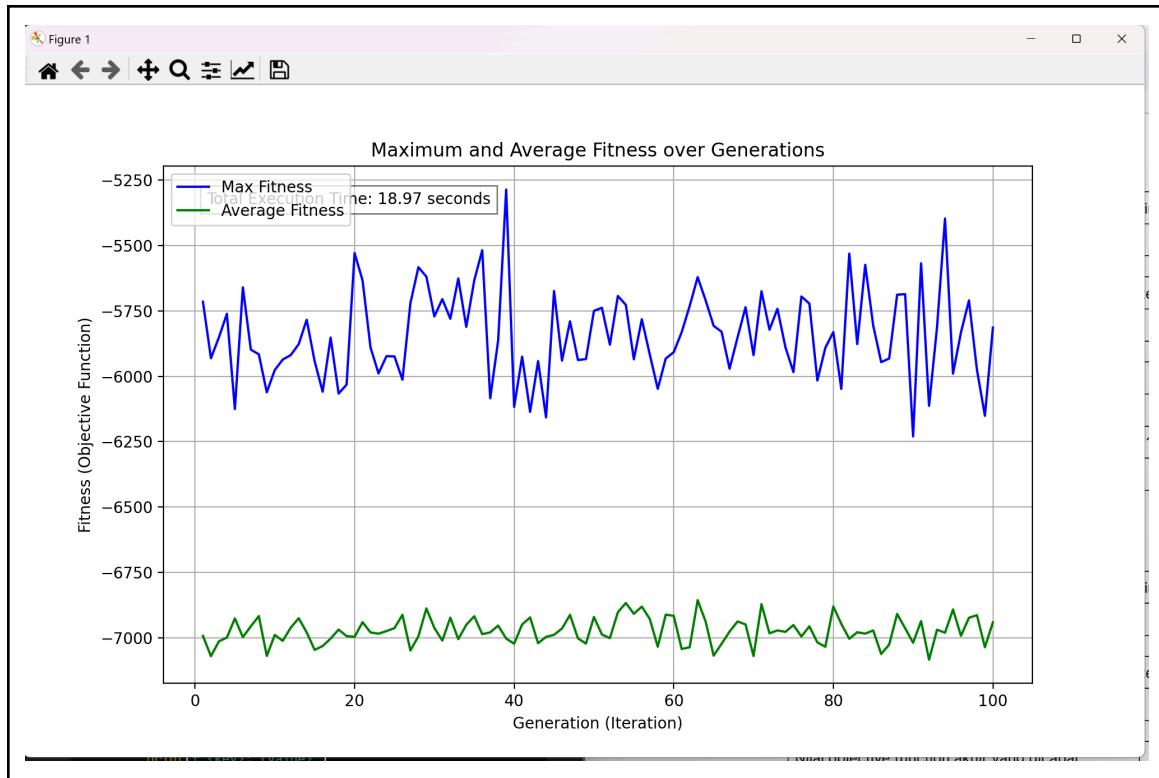
Durasi proses pencarian dan jumlah iterasi hingga berhenti juga hampir identik pada ketiga percobaan, menunjukkan bahwa algoritma berhenti secara konsisten setelah jumlah iterasi yang sama. Waktu pencarian yang berkisar antara 7.72 hingga 7.95 detik serta iterasi yang sama di setiap percobaan (138,149 iterasi) menunjukkan efisiensi metode ini, yang stabil di berbagai kondisi awal. Durasi pencarian juga hanya sekitar 7-8 detik, yang jauh lebih singkat dari beberapa algoritma lain yang diuji.

2.3.6 Genetic Algorithm

- A. Jumlah populasi sebagai kontrol (variasi iterasi)
 - a. Jumlah iterasi = 100, populasi = 100
 - i. Percobaan 1



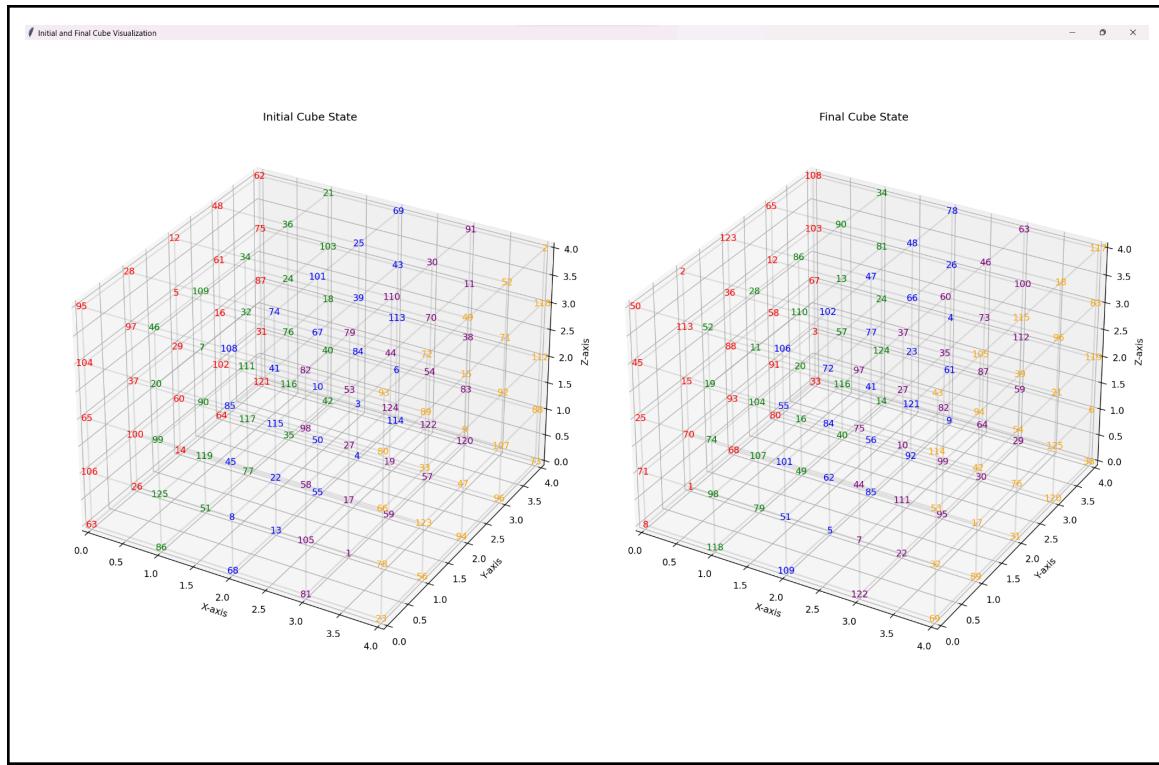
Plot nilai objective function terhadap banyak iterasi yang telah dilewati



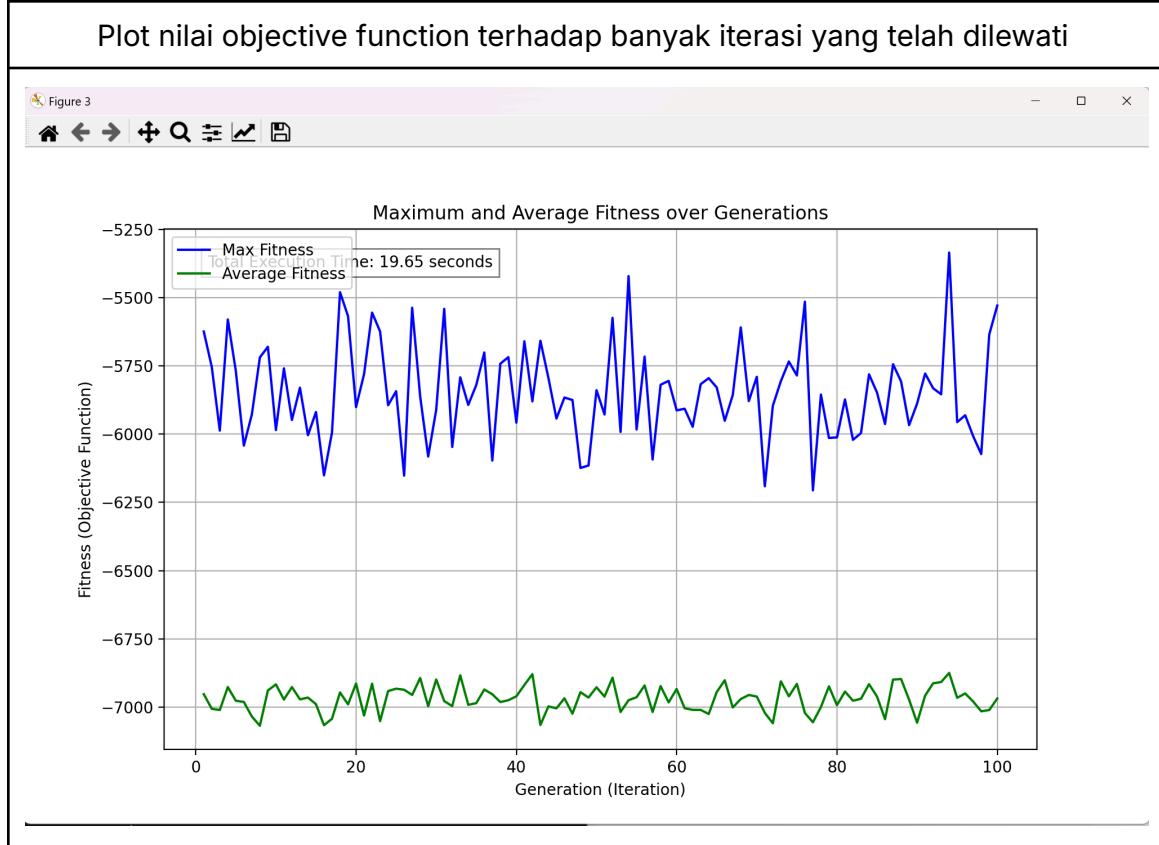
Nilai objective function akhir yang dicapai	-5286 (Terbaik)
Jumlah populasi	100
Banyak iterasi	100
Durasi proses pencarian	18.97 detik

ii. Percobaan 2

State awal vs state akhir

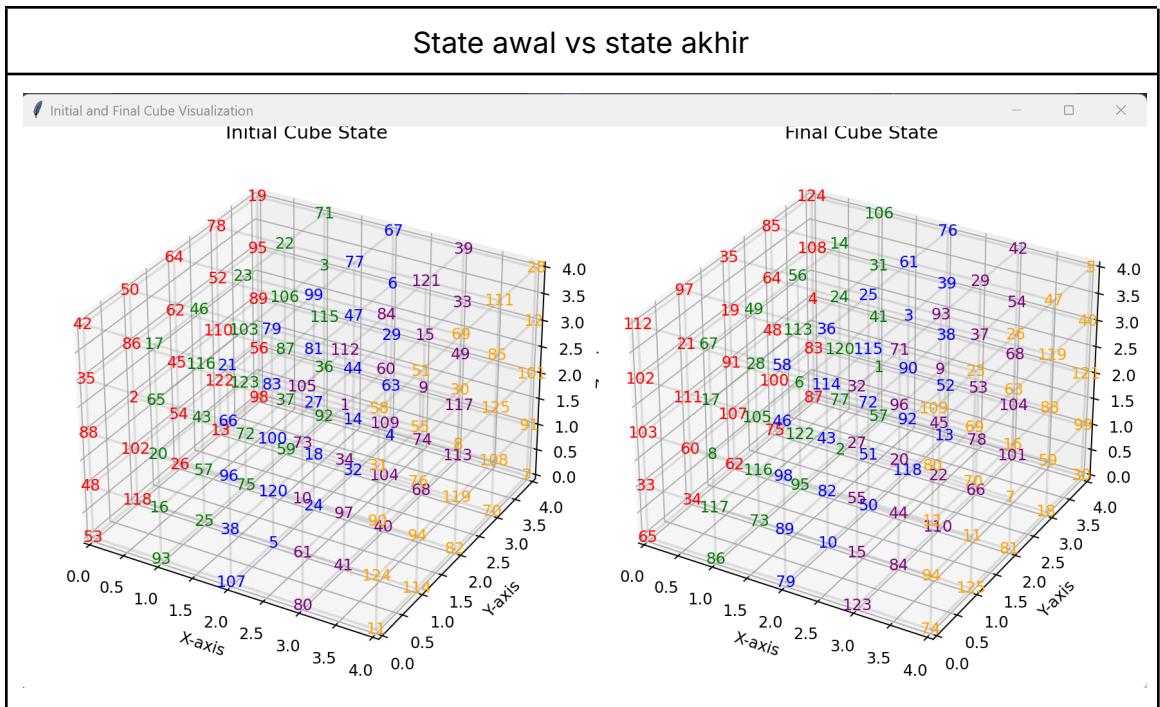


Plot nilai objective function terhadap banyak iterasi yang telah dilewati

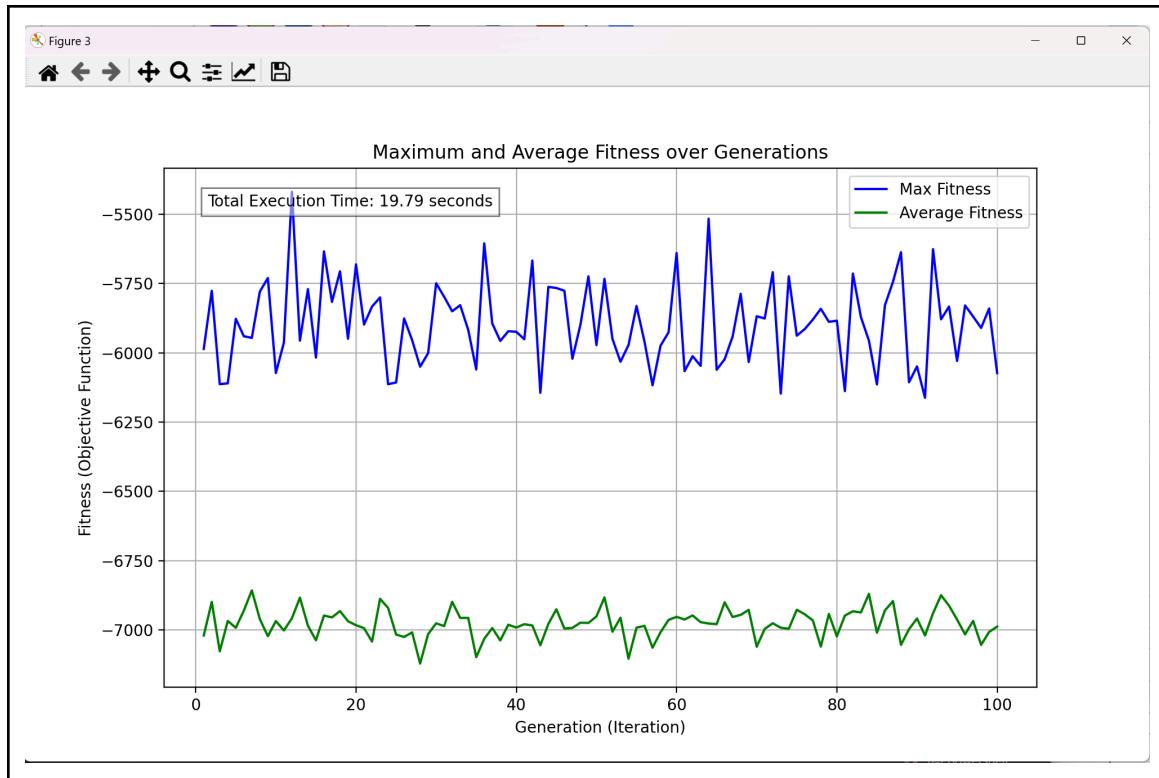


Nilai objective function akhir yang dicapai	-5335
Jumlah populasi	100
Banyak iterasi	100
Durasi proses pencarian	19.65

iii. Percobaan 3



Plot nilai objective function terhadap banyak iterasi yang telah dilewati

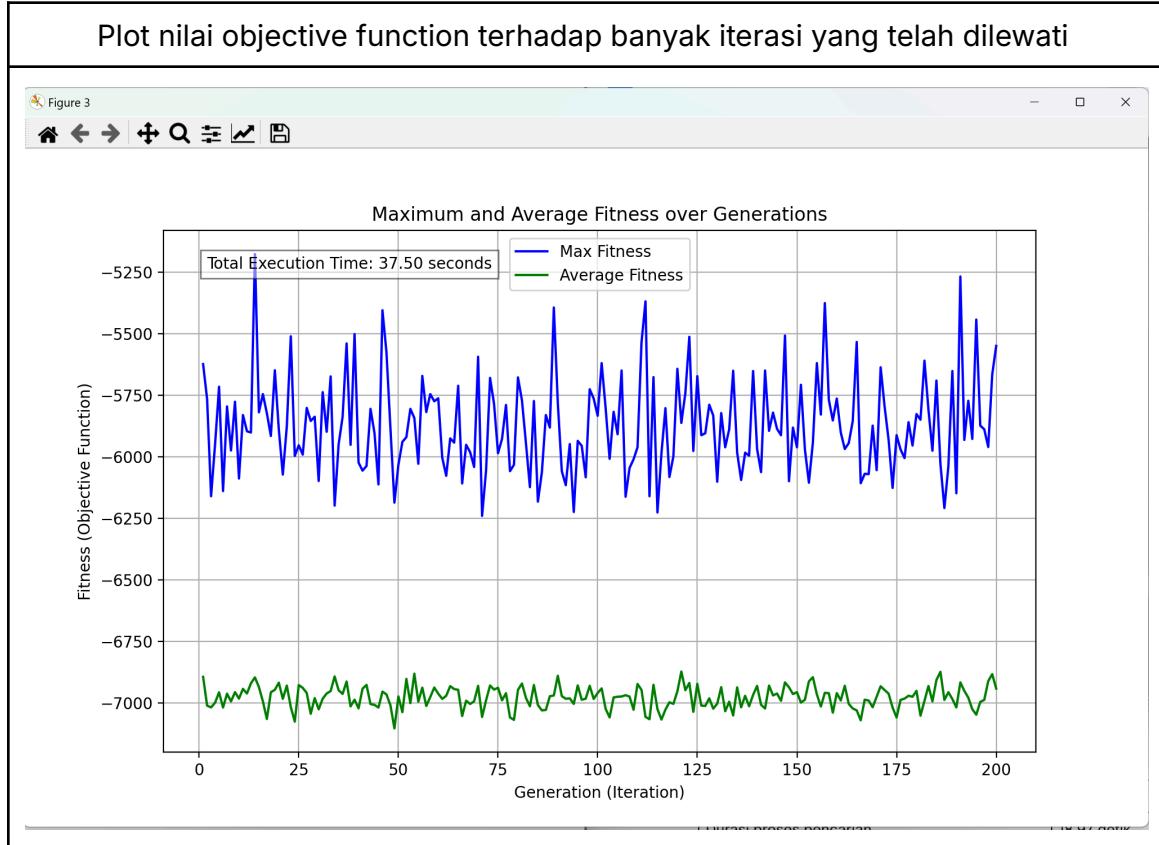
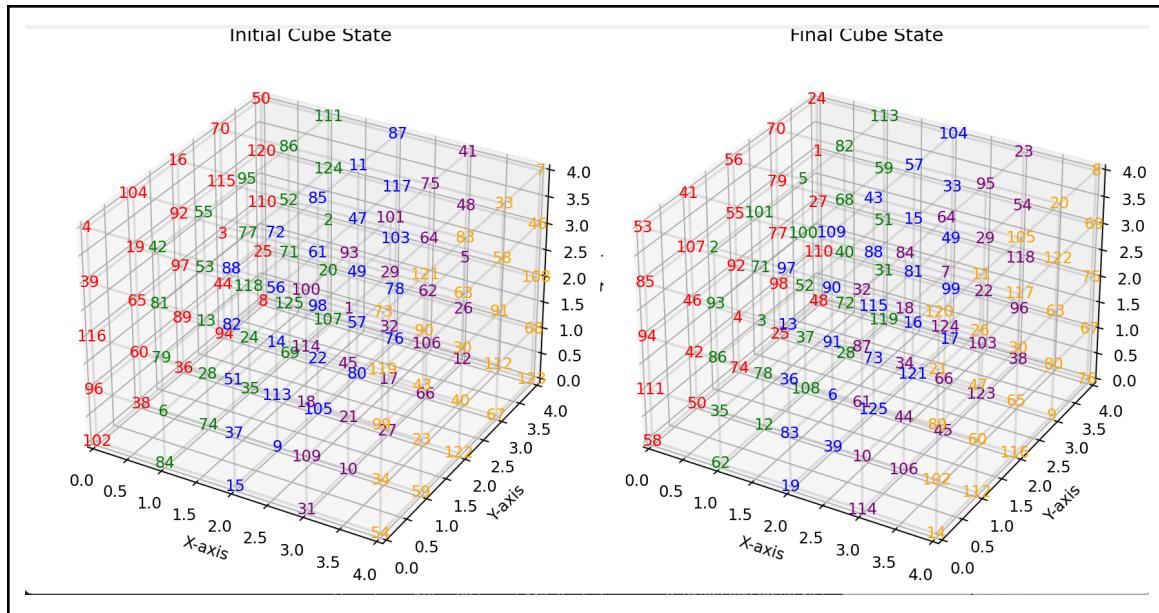


Nilai objective function akhir yang dicapai	-5419
Jumlah populasi	100
Banyak iterasi	100
Durasi proses pencarian	19.79

b. Jumlah iterasi = 200, populasi = 100

i. Percobaan 1

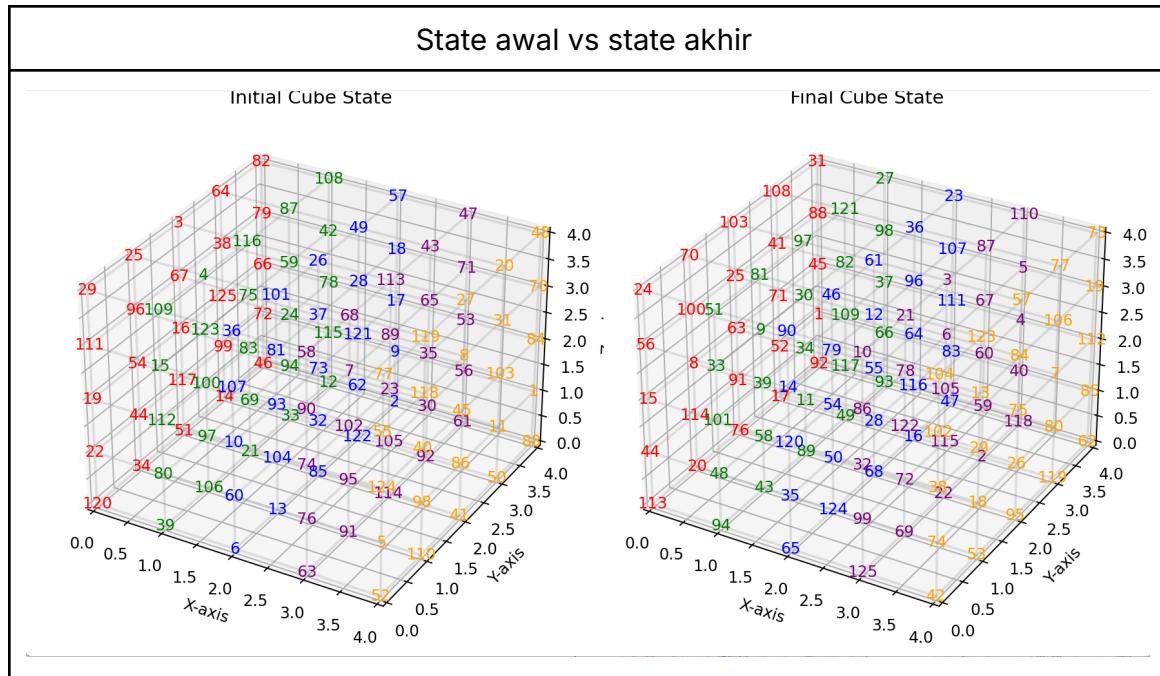
State awal vs state akhir



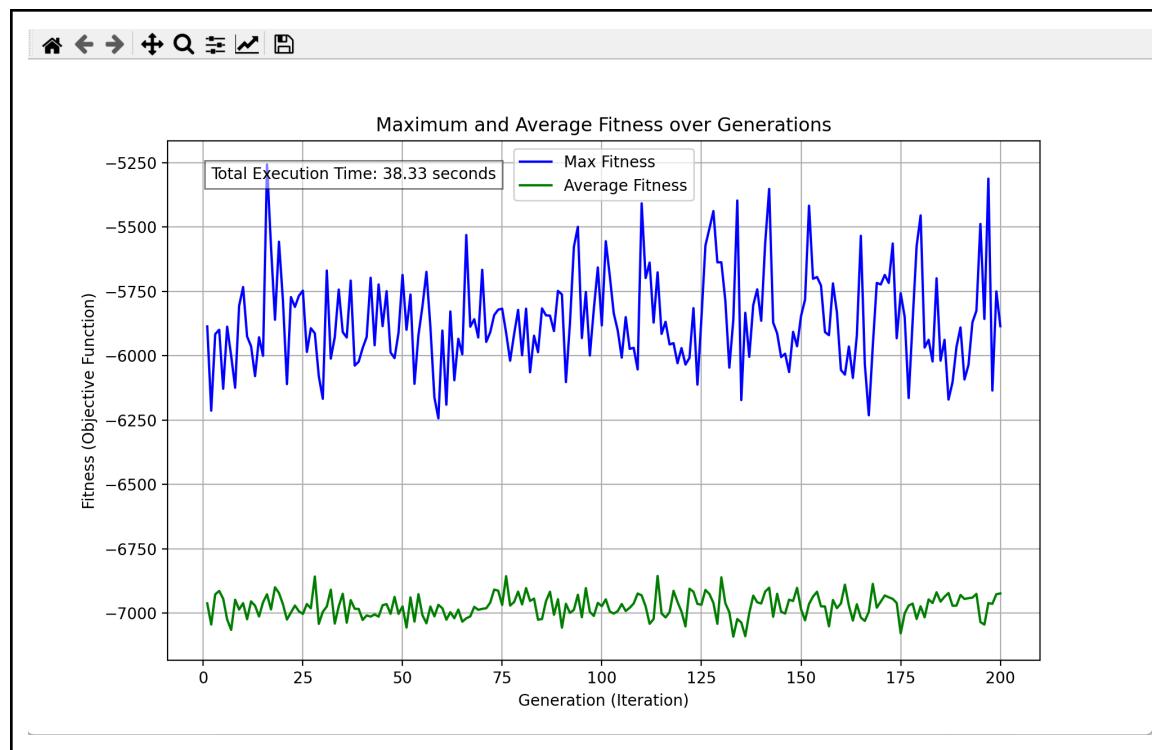
Nilai objective function akhir yang dicapai	-5176 (Terbaik)
Jumlah populasi	100

Banyak iterasi	200
Durasi proses pencarian	37.50 detik

ii. Percobaan 2



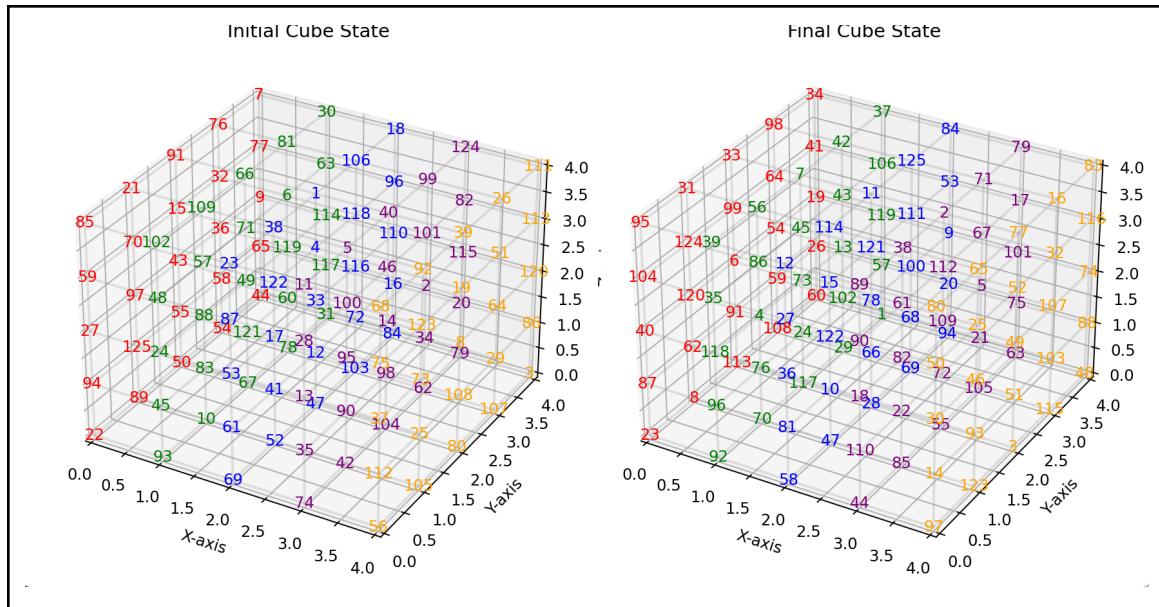
Plot nilai objective function terhadap banyak iterasi yang telah dilewati



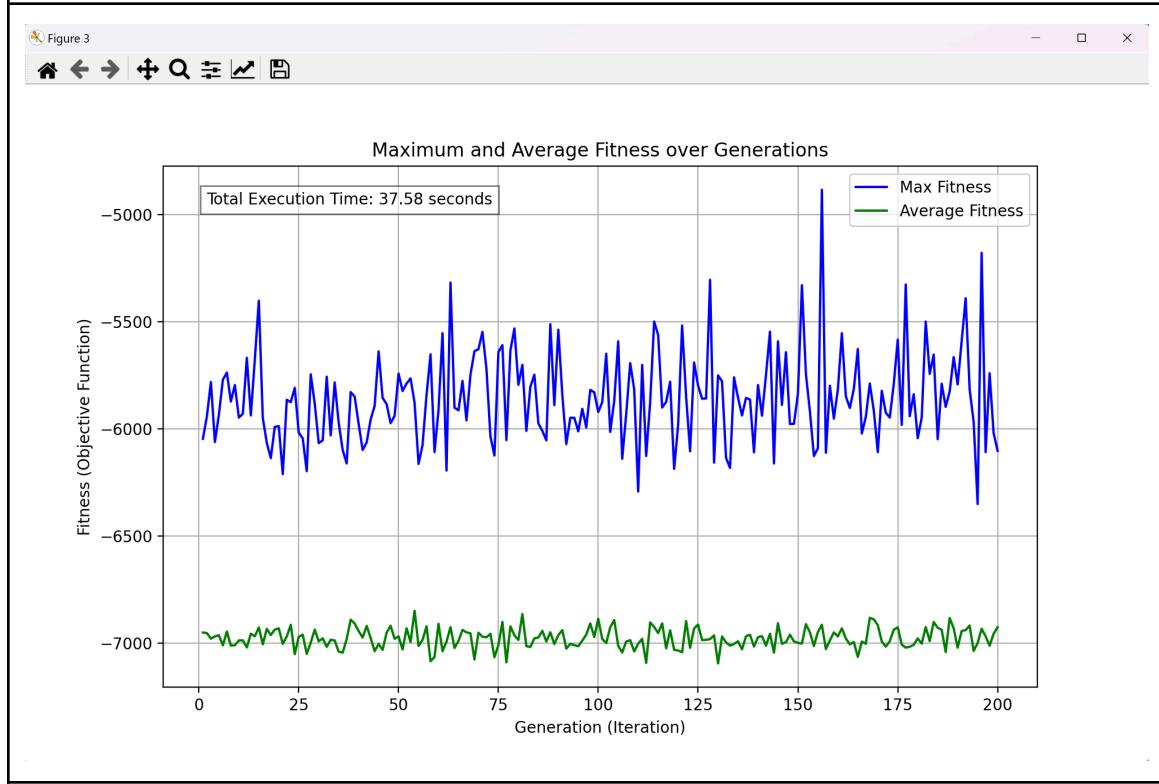
Nilai objective function akhir yang dicapai	-5257 (terbaik)
Jumlah populasi	100
Banyak iterasi	200
Durasi proses pencarian	37.58 detik

iii. Percobaan 3

State awal vs state akhir



Plot nilai objective function terhadap banyak iterasi yang telah dilewati

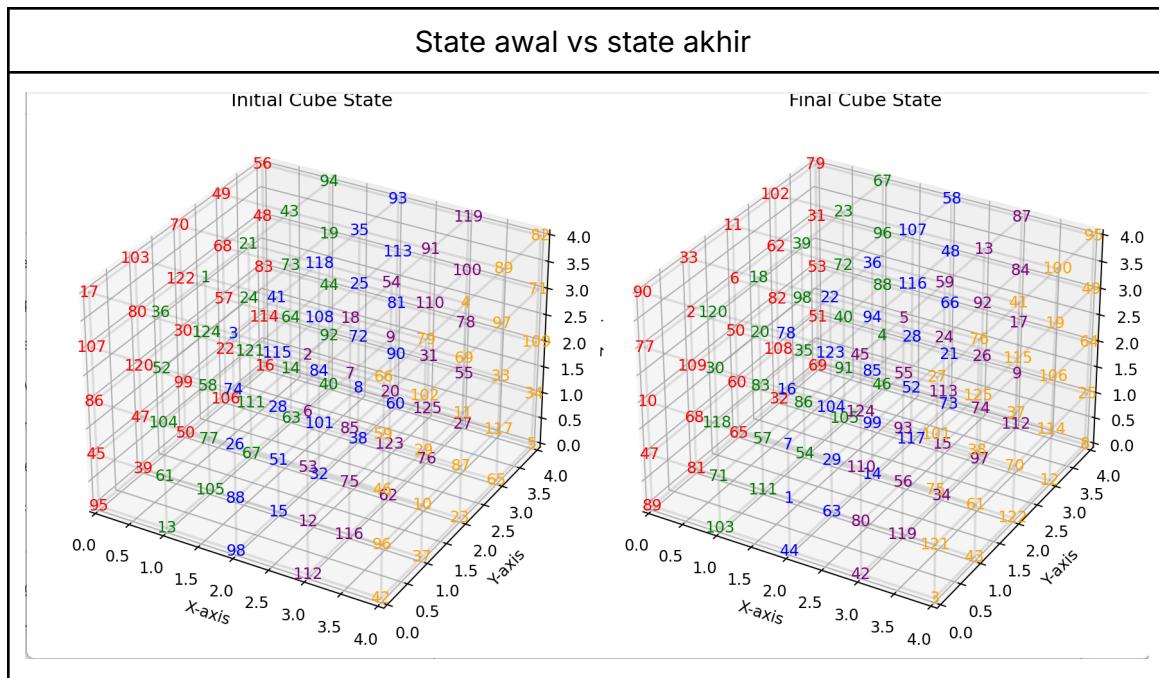


Nilai objective function akhir yang dicapai	-4884 (terbaik)
Jumlah populasi	100

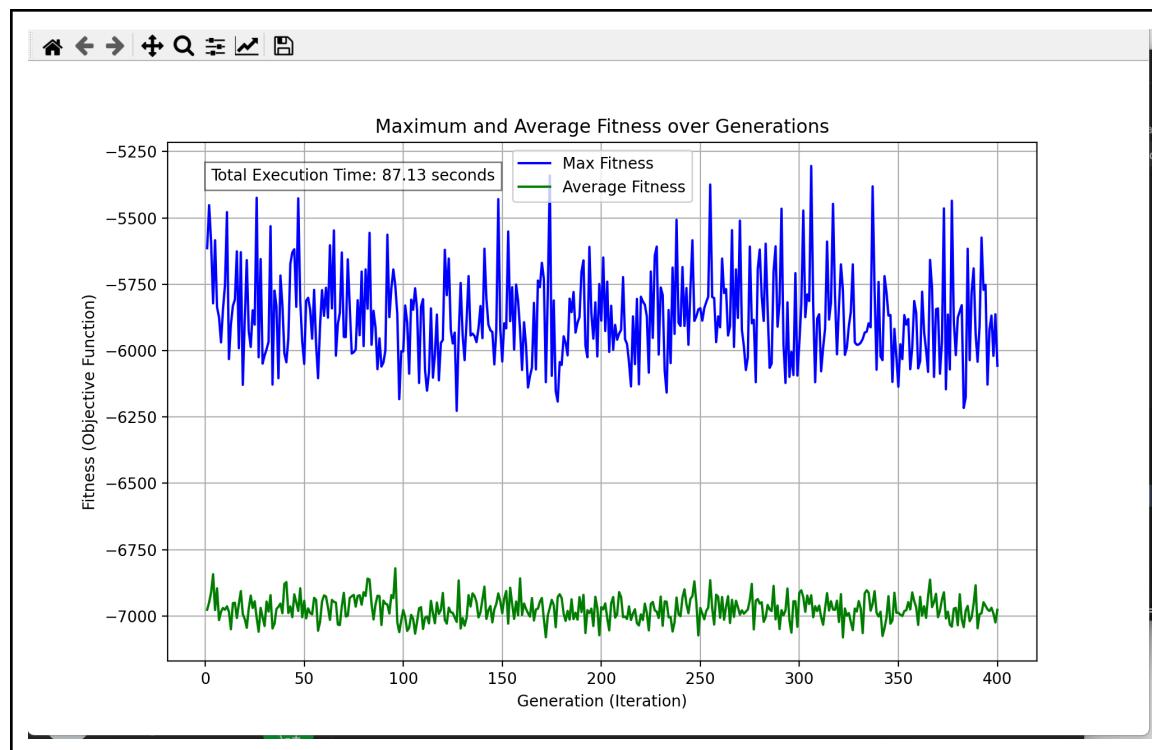
Banyak iterasi	200
Durasi proses pencarian	37.58 detik

c. Jumlah iterasi = 400, populasi = 100

i. Percobaan 1



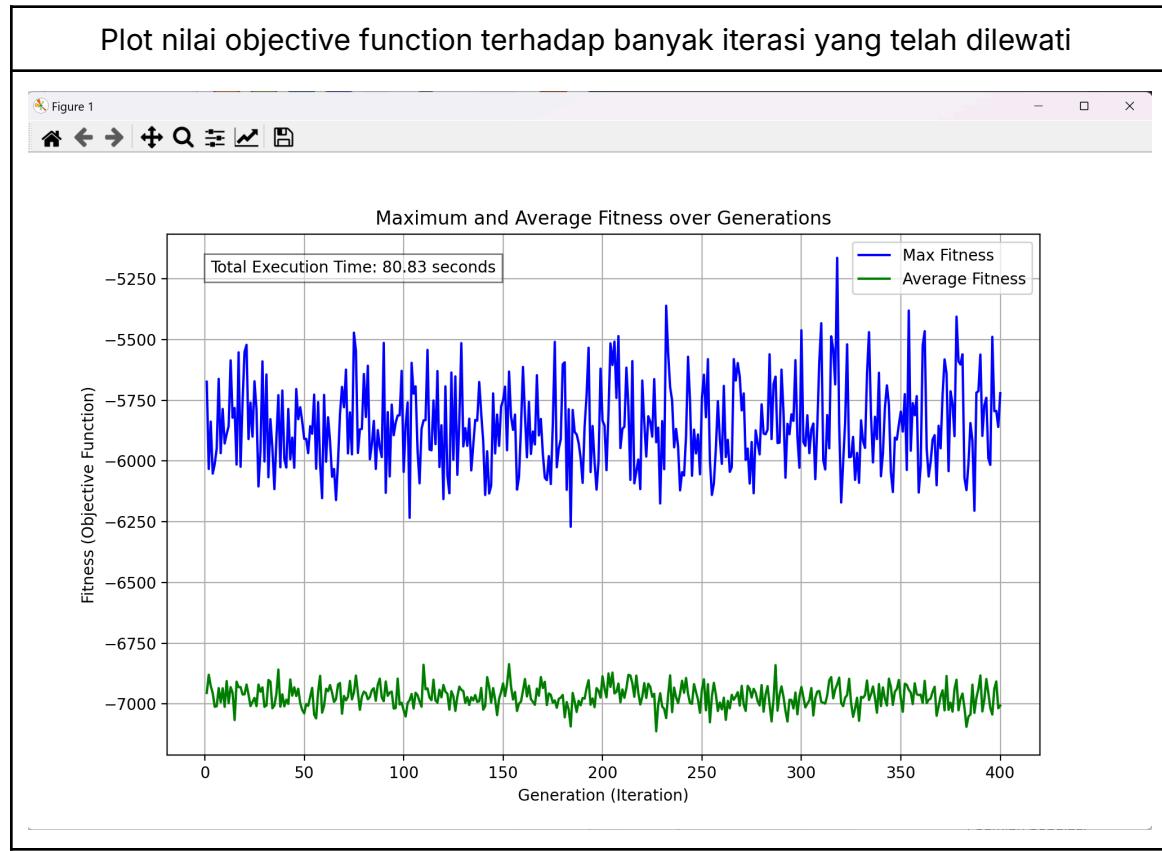
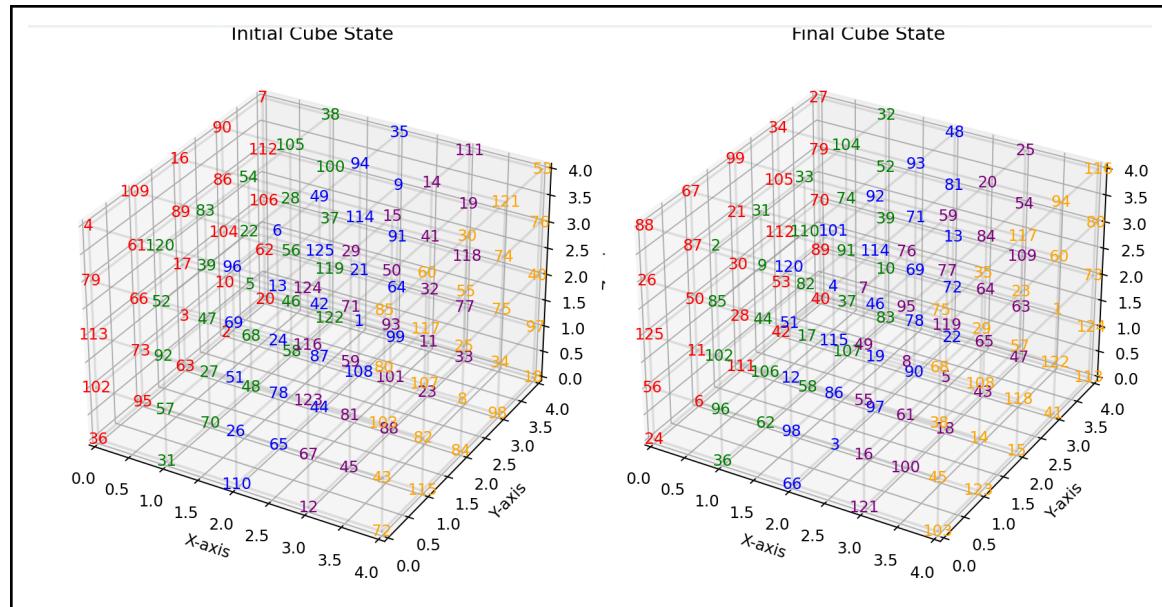
Plot nilai objective function terhadap banyak iterasi yang telah dilewati



Nilai objective function akhir yang dicapai	-5340 (terbaik)
Jumlah populasi	100
Banyak iterasi	400
Durasi proses pencarian	87.13 detik

ii. Percobaan 2

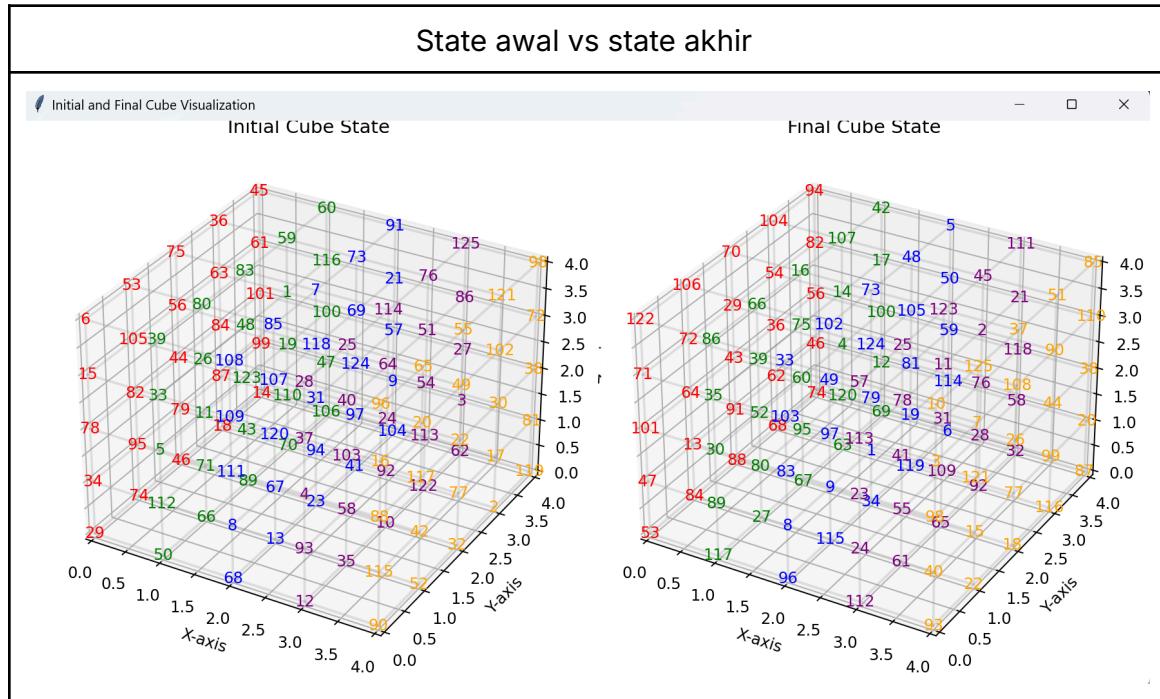
State awal vs state akhir



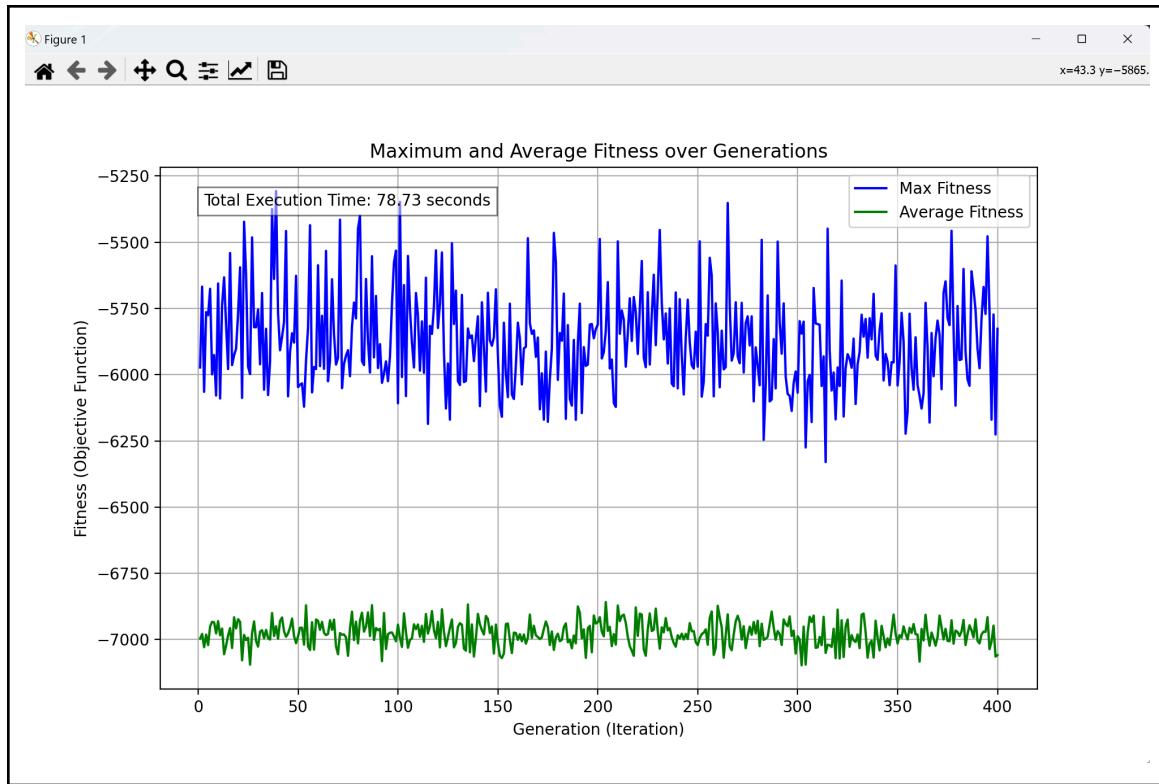
Nilai objective function akhir yang dicapai	-5164 (terbaik)
Jumlah populasi	100

Banyak iterasi	400
Durasi proses pencarian	80.83 detik

iii. Percobaan 3



Plot nilai objective function terhadap banyak iterasi yang telah dilewati



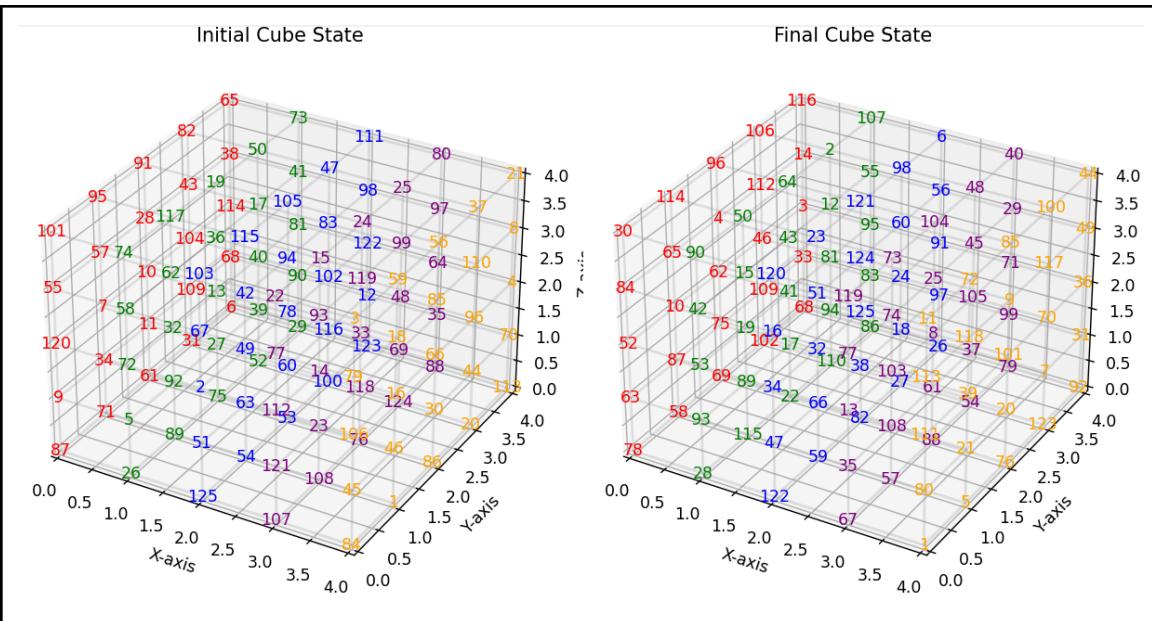
Nilai objective function akhir yang dicapai	-5307 (terbaik)
Jumlah populasi	100
Banyak iterasi	400
Durasi proses pencarian	78.73 detik

B. Jumlah iterasi sebagai kontrol (variasi populasi)

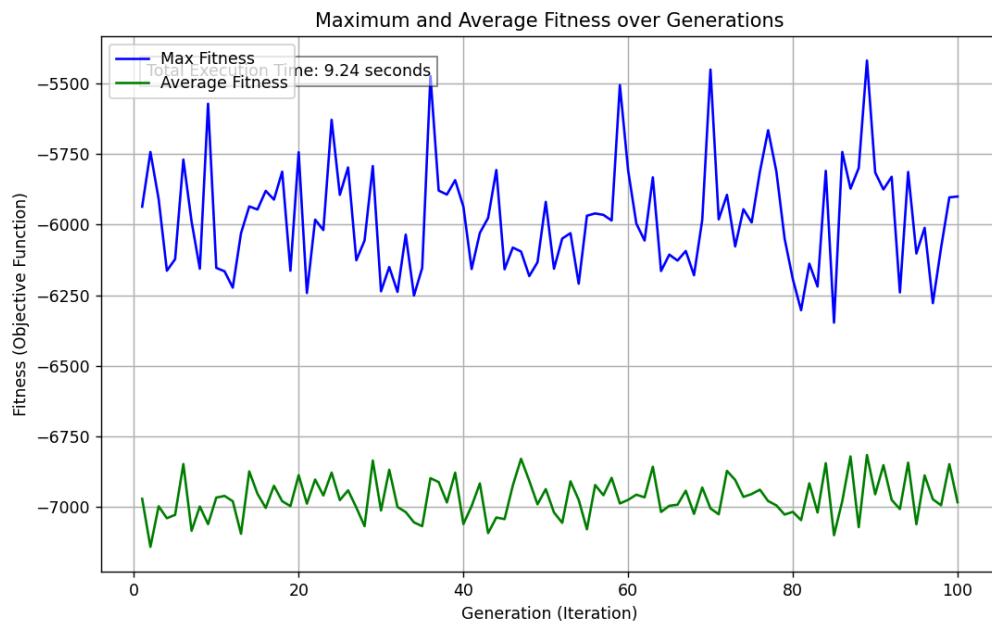
a. Jumlah iterasi = 100, populasi = 50

i. Percobaan 1

State awal vs state akhir



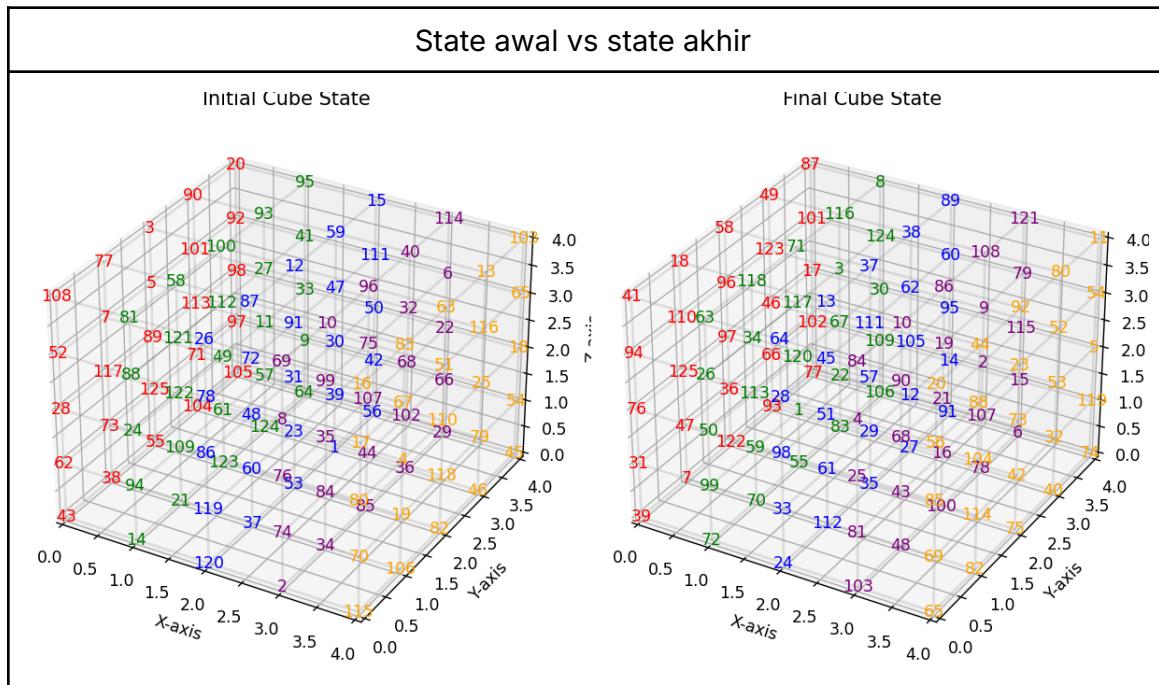
Plot nilai objective function terhadap banyak iterasi yang telah dilewati



Nilai objective function akhir yang dicapai	-5410.0
Jumlah populasi	50
Banyak iterasi	100

Durasi proses pencarian	9.24 detik
-------------------------	------------

ii. Percobaan 2

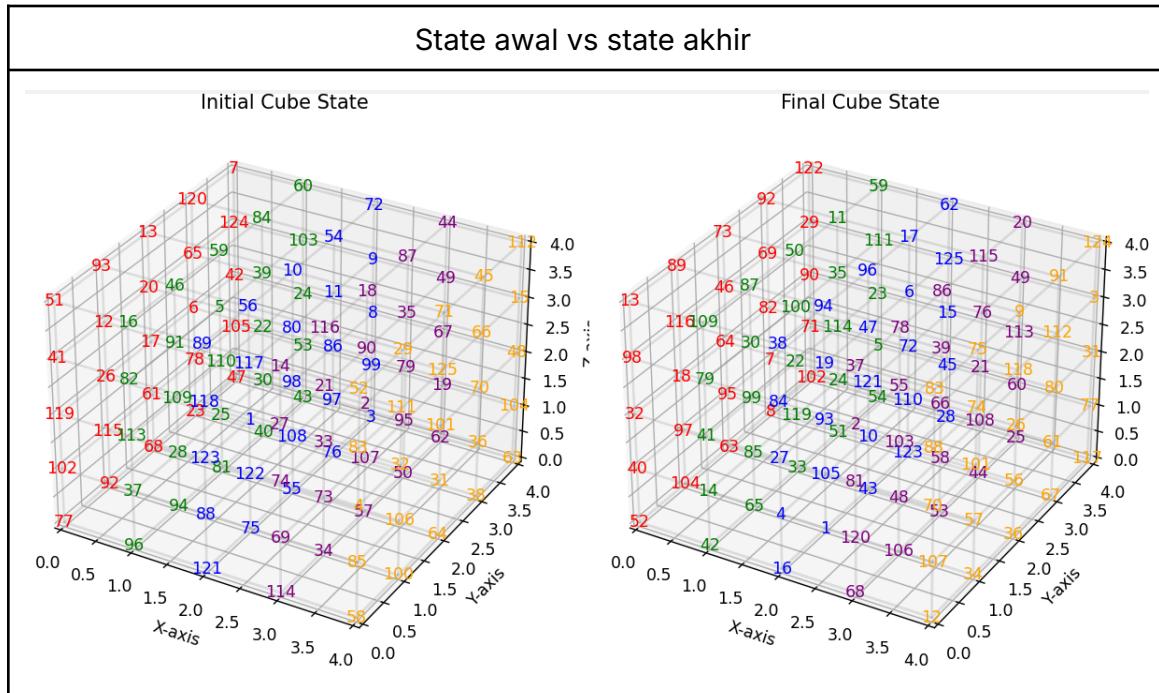


Plot nilai objective function terhadap banyak iterasi yang telah dilewati

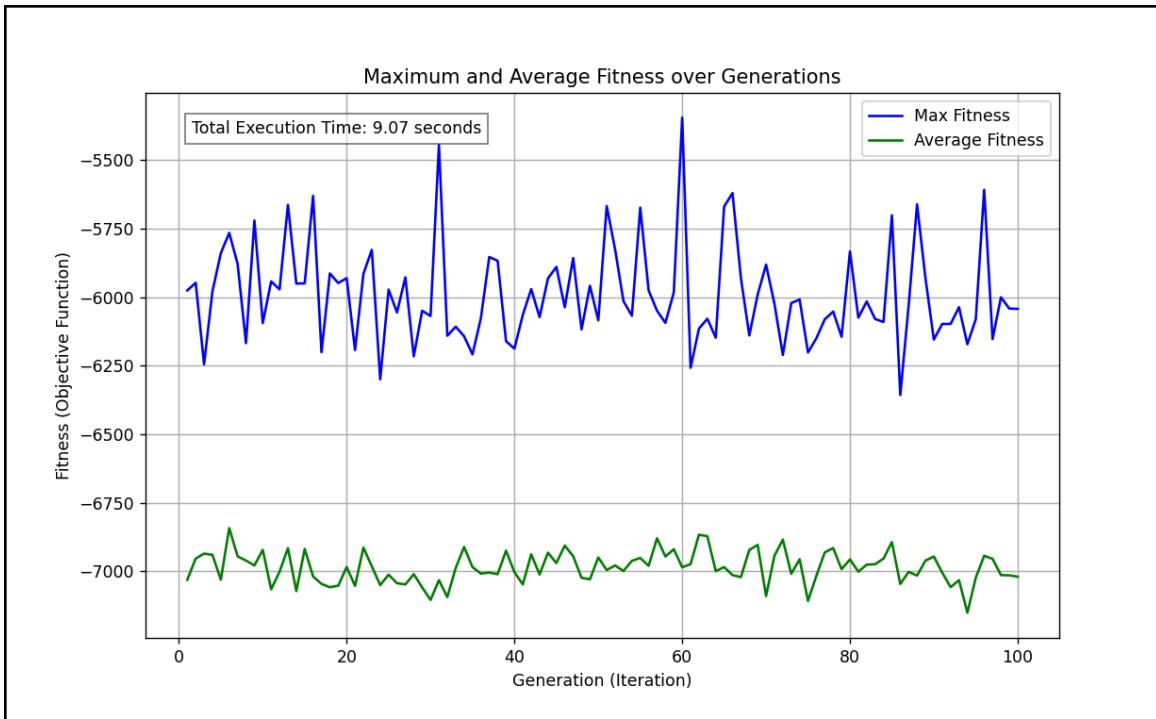


Nilai objective function akhir yang dicapai	-5398.0
Jumlah populasi	50
Banyak iterasi	100
Durasi proses pencarian	9.18 detik

iii. Percobaan 3



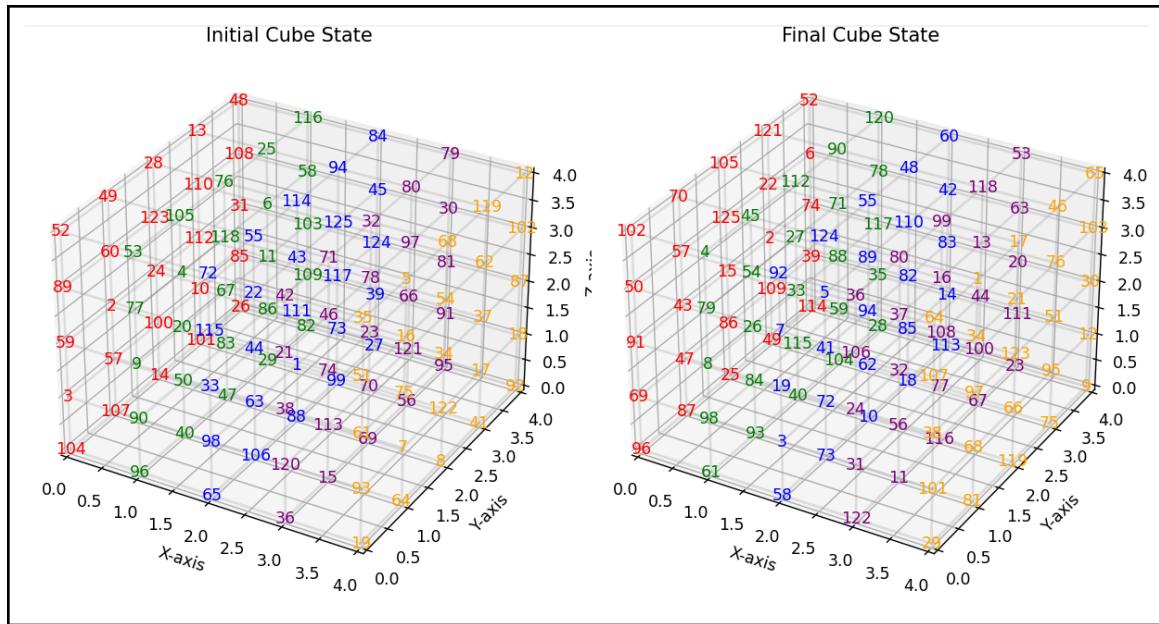
Plot nilai objective function terhadap banyak iterasi yang telah dilewati



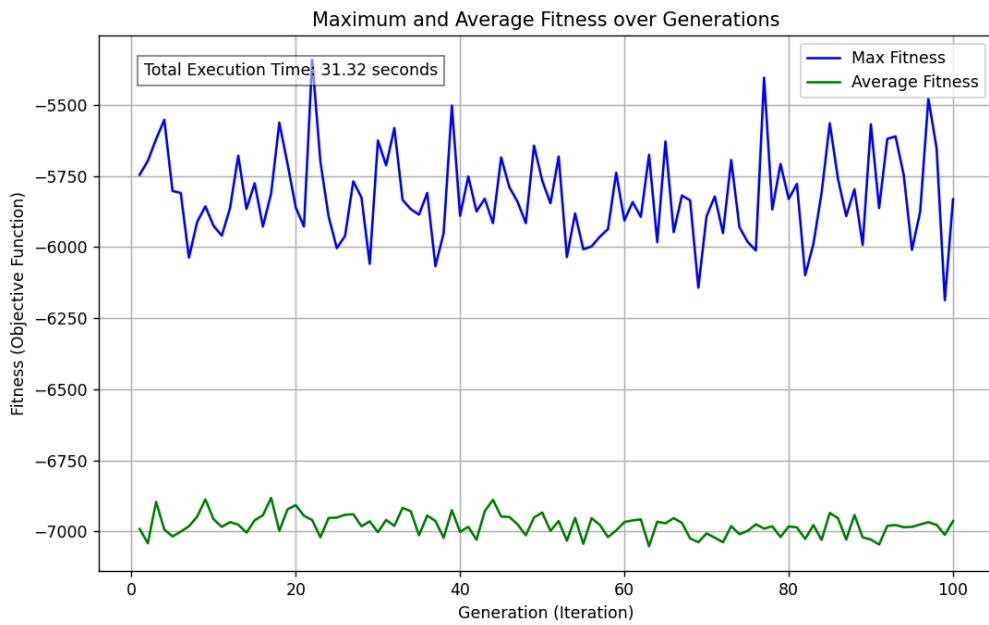
Nilai objective function akhir yang dicapai	-5312.0
Jumlah populasi	50
Banyak iterasi	100
Durasi proses pencarian	9.07 detik

- b. Jumlah iterasi = 100, populasi = 150
 i. Percobaan 1

State awal vs state akhir



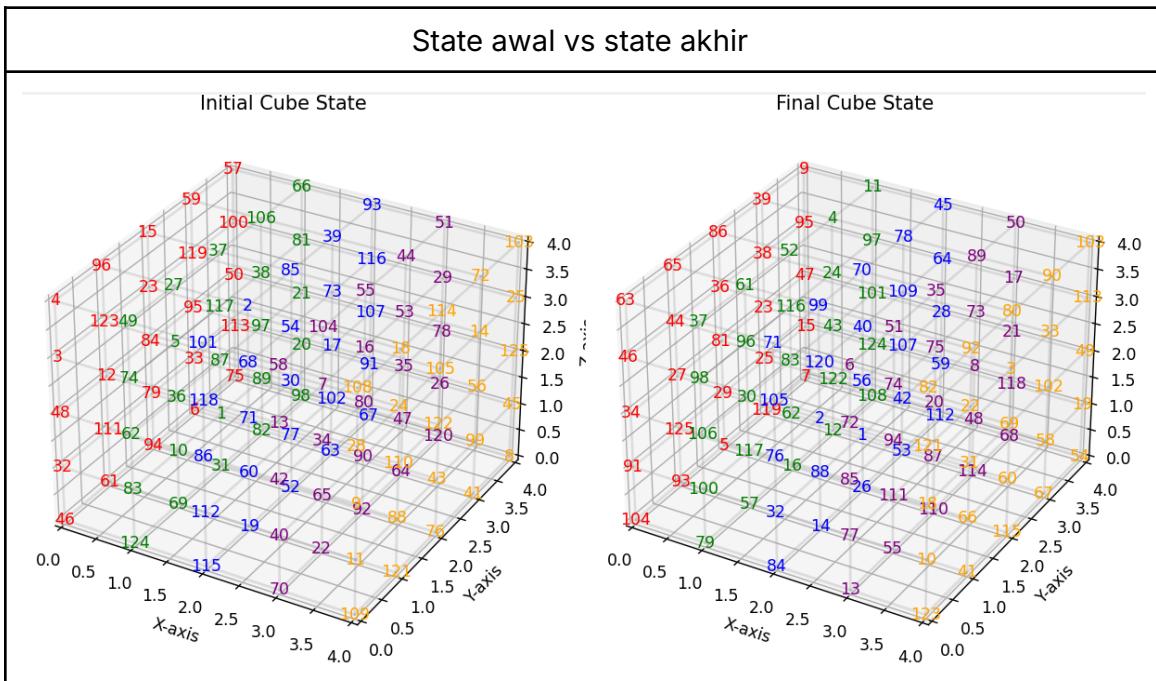
Plot nilai objective function terhadap banyak iterasi yang telah dilewati



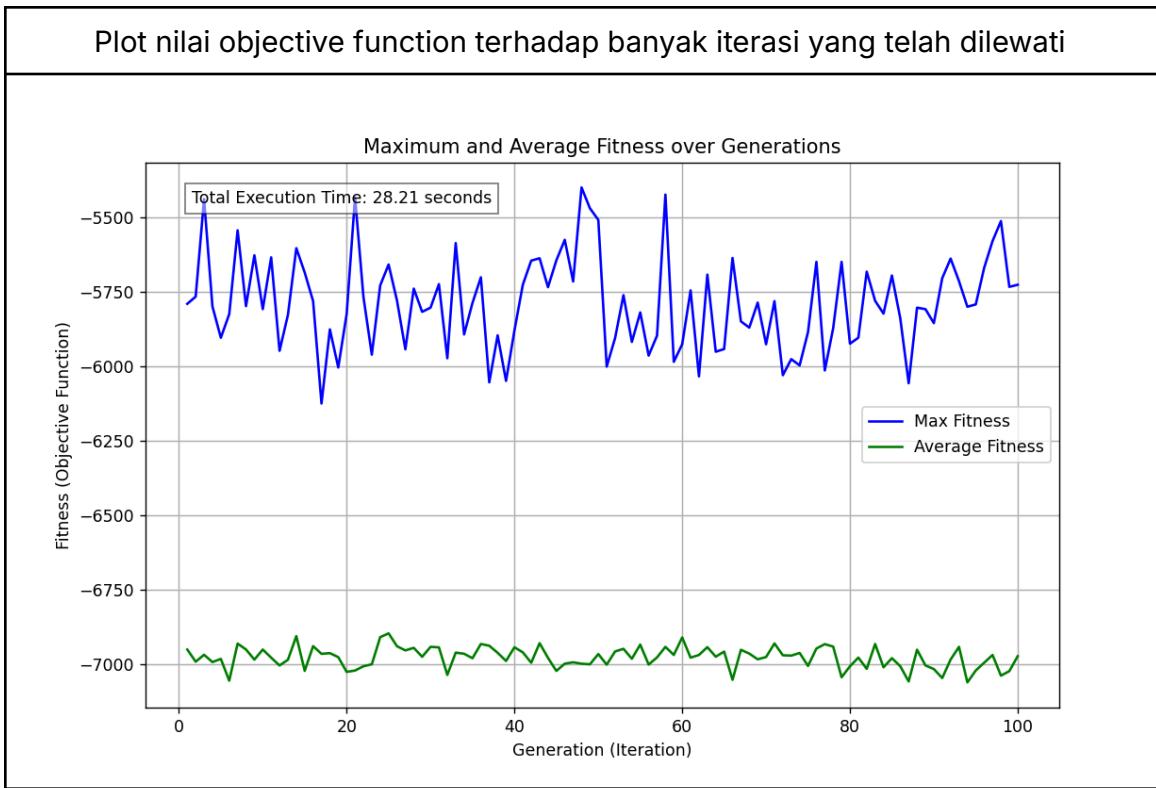
Nilai objective function akhir yang dicapai	5289.0
Jumlah populasi	150
Banyak iterasi	100

Durasi proses pencarian	31.32 detik
-------------------------	-------------

ii. Percobaan 2

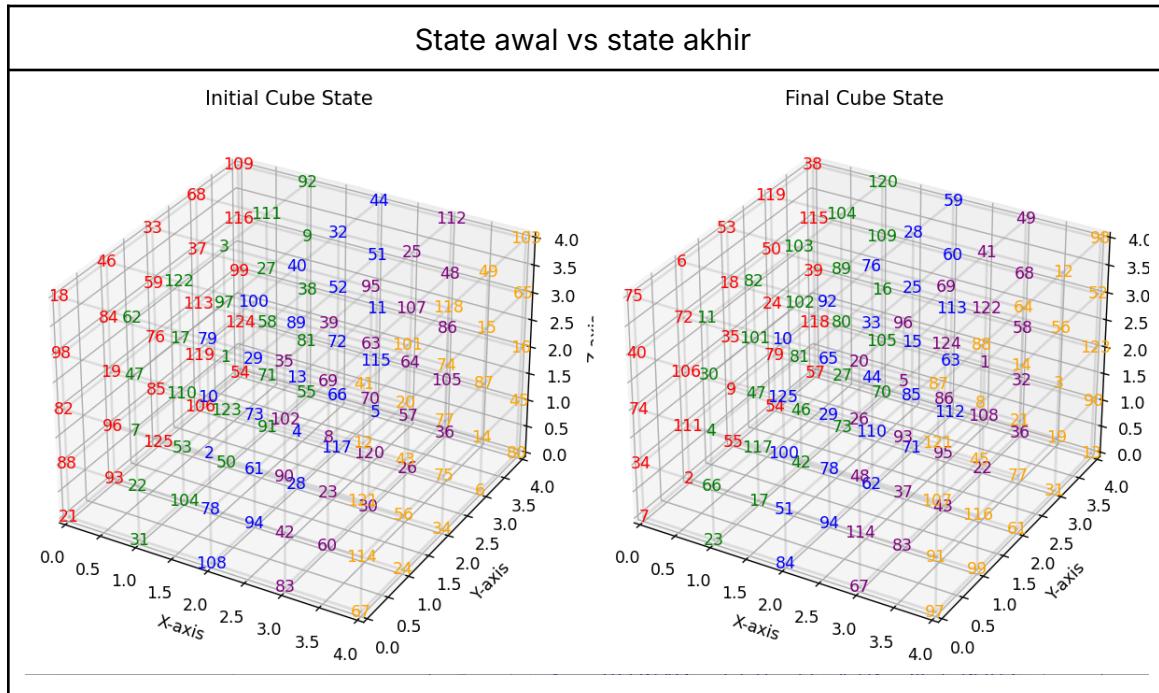


Plot nilai objective function terhadap banyak iterasi yang telah dilewati

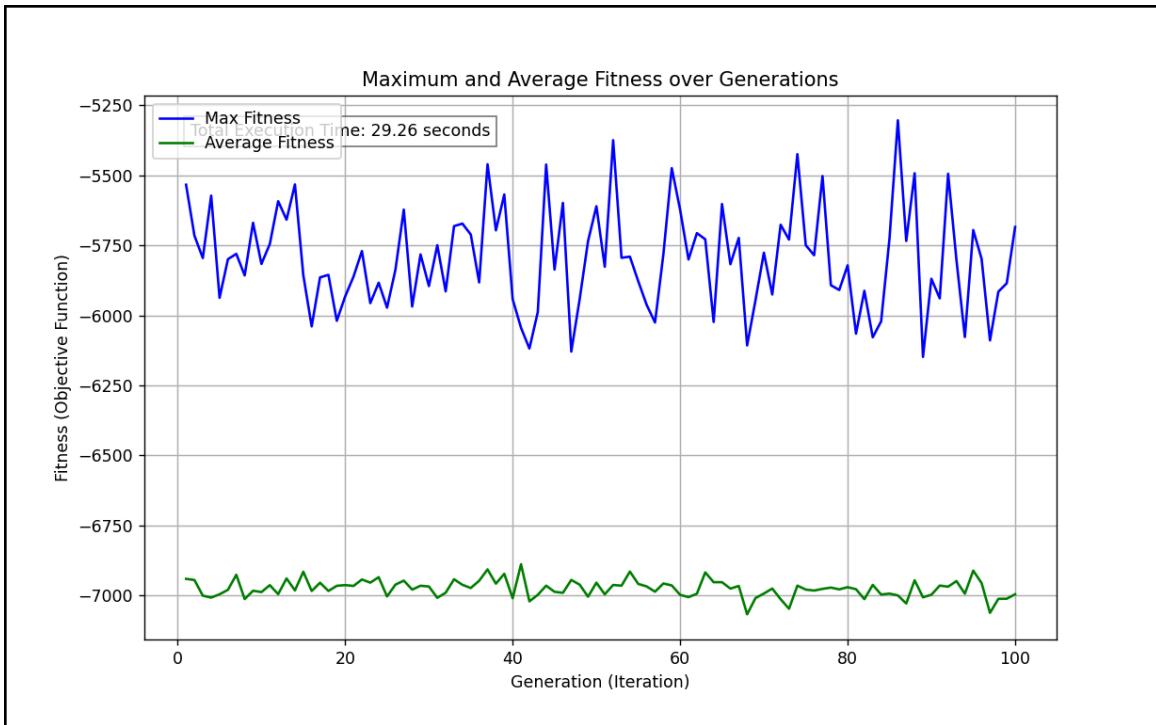


Nilai objective function akhir yang dicapai	5390.0
Jumlah populasi	150
Banyak iterasi	100
Durasi proses pencarian	28.21 detik

iii. Percobaan 3



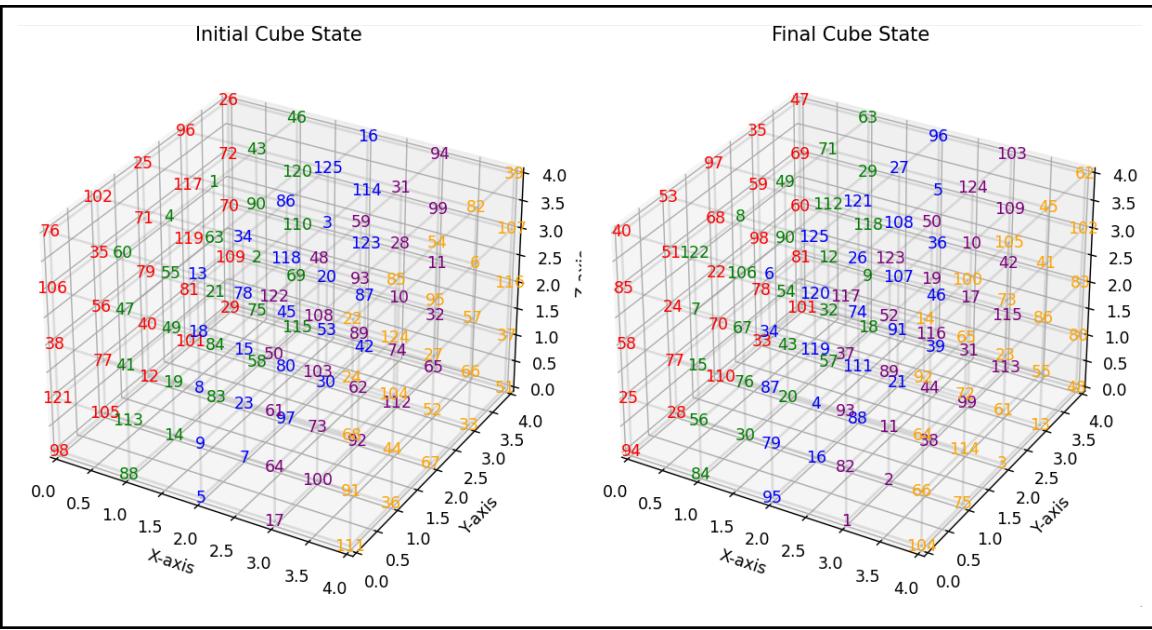
Plot nilai objective function terhadap banyak iterasi yang telah dilewati



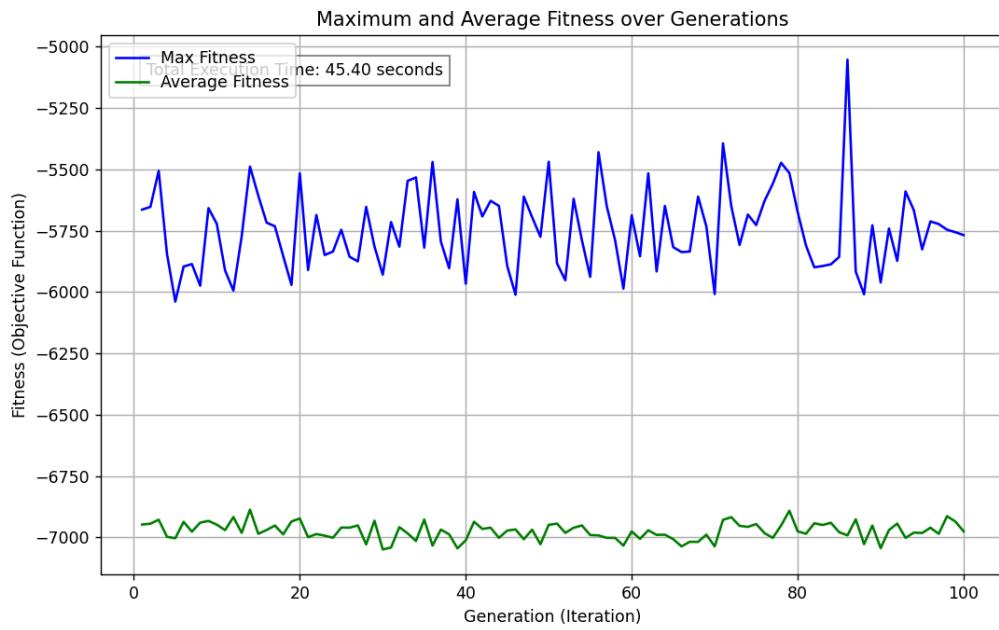
Nilai objective function akhir yang dicapai	5304.0
Jumlah populasi	150
Banyak iterasi	100
Durasi proses pencarian	29.26 detik

- c. Jumlah iterasi = 100, populasi = 200
 i. Percobaan 1

State awal vs state akhir



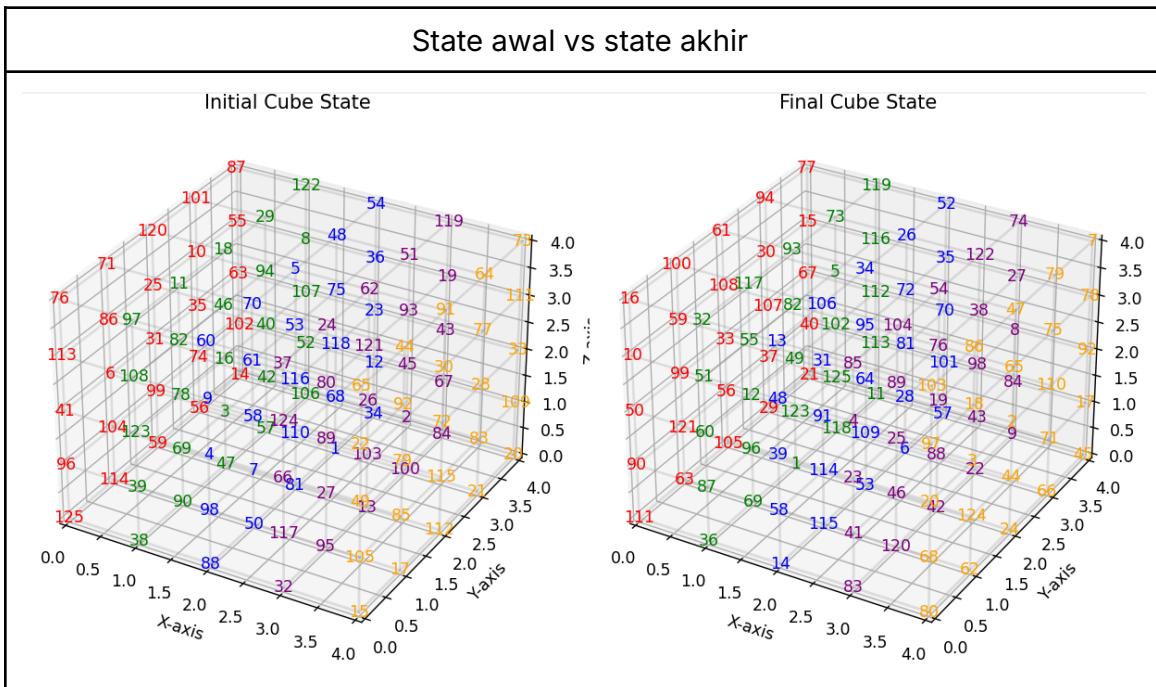
Plot nilai objective function terhadap banyak iterasi yang telah dilewati



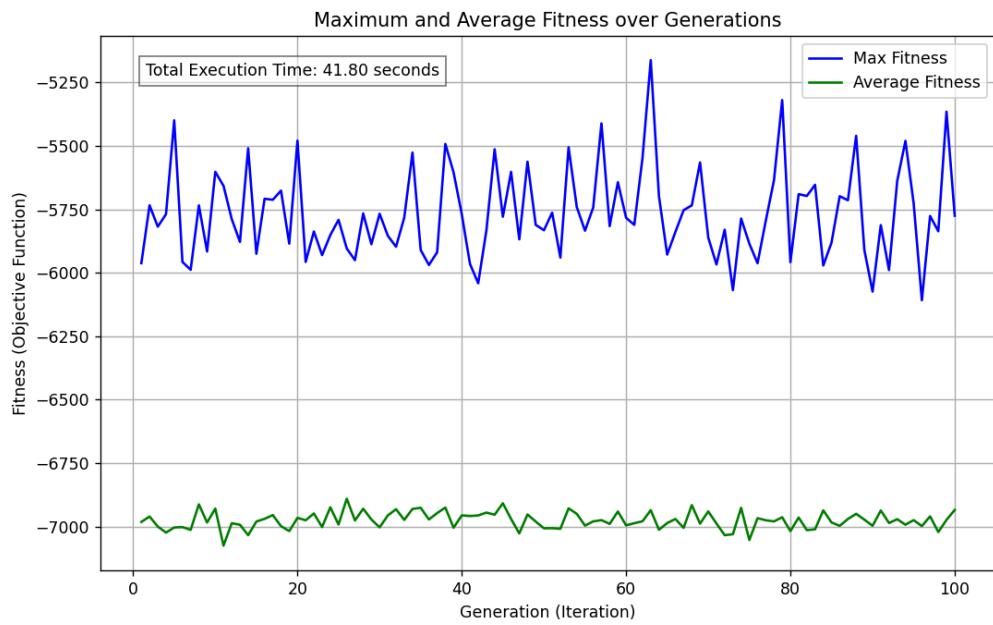
Nilai objective function akhir yang dicapai	-5091.0
Jumlah populasi	200
Banyak iterasi	100

Durasi proses pencarian	45.40 detik
-------------------------	-------------

ii. Percobaan 2

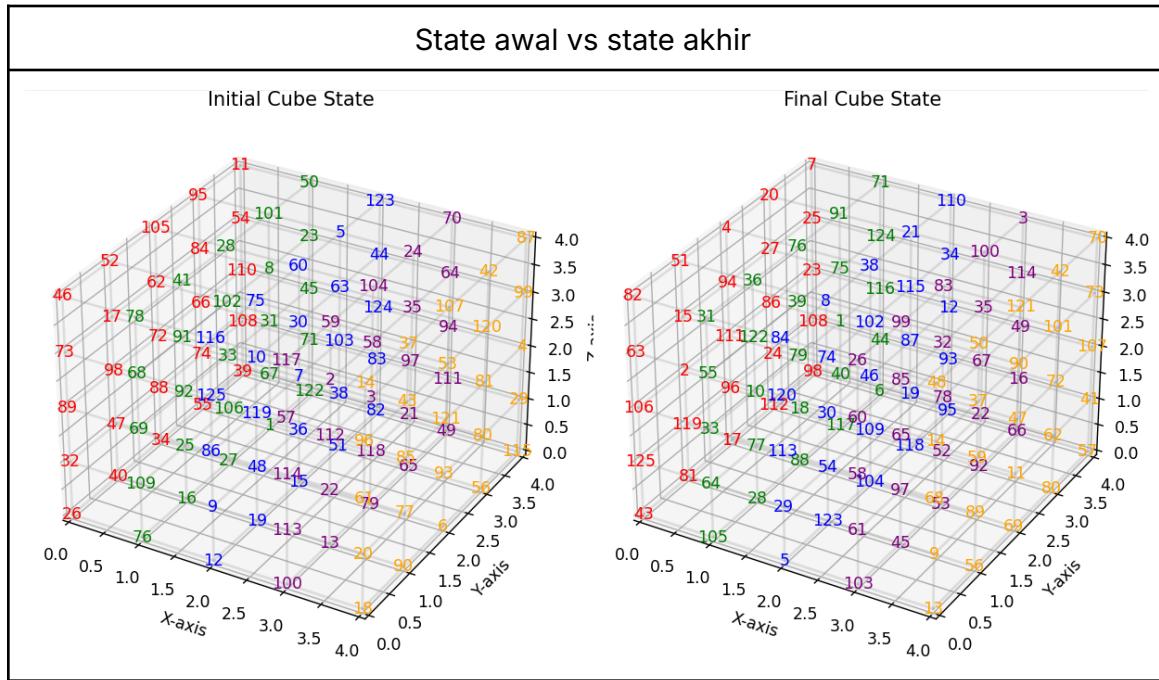


Plot nilai objective function terhadap banyak iterasi yang telah dilewati

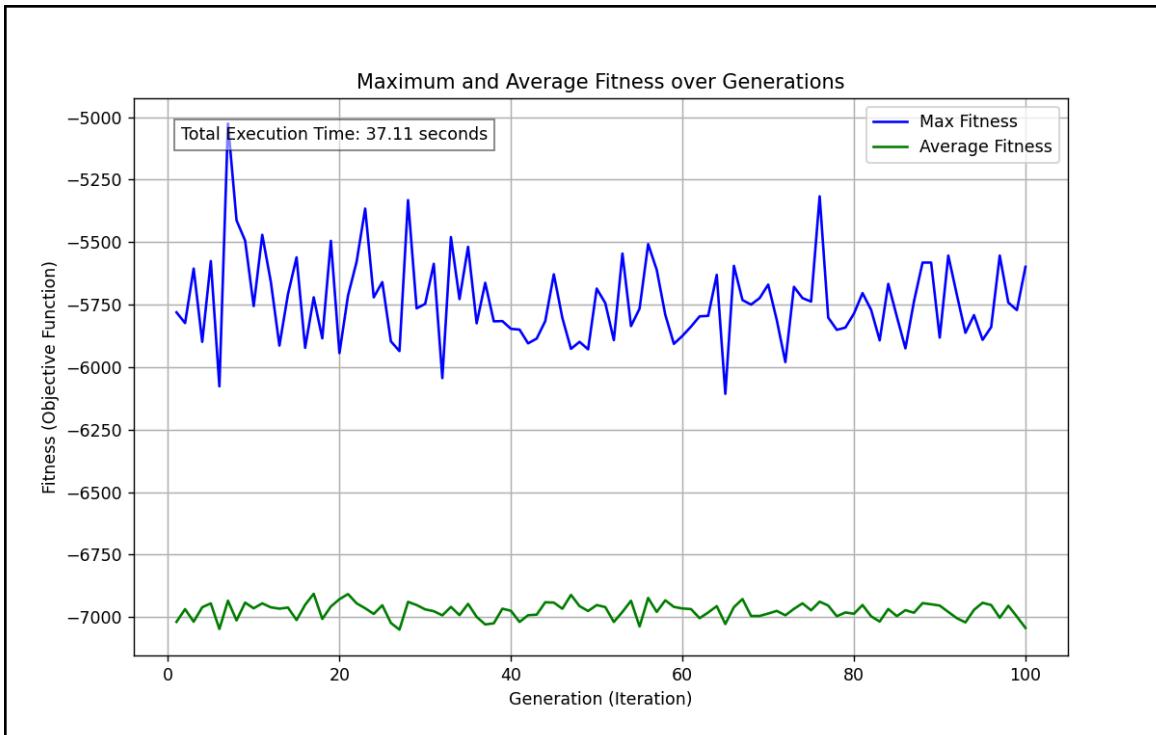


Nilai objective function akhir yang dicapai	-5124.0
Jumlah populasi	200
Banyak iterasi	100
Durasi proses pencarian	41.80 detik

iii. Percobaan 3



Plot nilai objective function terhadap banyak iterasi yang telah dilewati



Nilai objective function akhir yang dicapai	-5026.0
Jumlah populasi	200
Banyak iterasi	100
Durasi proses pencarian	37.11 detik

2.4 Analisis

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

1. Steepest Ascent

Dari 3 hasil eksperimen yang telah dilakukan, hasil terbaik untuk steepest ascent adalah -774, artinya terdapat perbedaan sebanyak -774 nilai dari hasil terbaik 0. Algoritma Steepest Ascent Hill Climbing cenderung terjebak di local maxima, hal ini terjadi karena algoritma yang hanya memilih langkah yang memberikan nilai yang lebih besar pada tiap iterasi. Jika Algoritma ini menemukan sebuah local maxima, algoritma tidak dapat melakukan pencarian lebih jauh dan akhirnya berhenti pada state tersebut tanpa mencapai global optimum

2. Hill-Climbing with Sideways Move

Dari 3 hasil eksperimen yang telah dilakukan, hasil terbaik untuk Hill Climbing with Sideways move adalah -502.0. Hasil Hill Climbing with Sideways move memiliki keunggulan dibandingkan dengan steepest ascent karena dapat tetap bergerak di posisi datar (flat). Dengan adanya strategi seperti ini, algoritma menghindari local optima. Namun, jika terdapat banyak local optima atau flat yang panjang/berulang kali, algoritma ini masih mungkin terjebak, sehingga hasilnya belum optimal.

3. Random Restart Hill Climbing

Dari 3 hasil eksperimen yang telah dilakukan, hasil terbaik untuk random restart adalah -734.0 dengan parameter max_restart sebanyak 5 kali. Algoritma Random Restart Hill Climbing menghindari terjebak dalam local optima dengan cara melakukan 'restart' secara acak. Algoritma ini memulai posisi acak untuk n kali untuk meningkatkan kemungkinan mencapai global optimum. Setiap restart akan memberi peluang untuk menemukan area pencarian yang baru dengan harapan mencapai global optimum. Namun, karena restart yang dilakukan bersifat acak, efisiensi dari algoritma tergantung dari jumlah restart yang dilakukan.

4. Stochastic Hill Climbing

Dari 3 hasil eksperimen yang telah dilakukan, hasil terbaik untuk stochastic hill climbing adalah -442.0. Algoritma Stochastic Hill Climbing cenderung memiliki nilai lebih dekat ke global optima dibandingkan dengan hill climbing yang lainnya. Stochastic Hill Climbing membangkitkan *neighbor* secara random. Algoritma ini tidak selalu memilih langkah terbaik di setiap iterasi, sehingga lebih kecil kemungkinannya untuk terjebak di local maxima. Stochastic Hill Climbing akan memilih nilai yang nilainya lebih besar dari *state* sekarang.

5. Simulated Annealing

Dari 3 hasil eksperimen yang telah dilakukan, hasil terbaik untuk simulated annealing adalah -160.0. Simulated Annealing menggunakan konsep "temperatur" yang menurun seiring waktu, sehingga memungkinkan untuk memilih langkah yang lebih buruk, jika akan mendatangkan ke solusi yang lebih baik. Pemilihan langkah lebih buruk dilakukan pada awalnya dan menjadi lebih selektif di akhir. Hal ini membantu algoritma keluar dari local maxima, terutama pada tahap awal pencarian, yang meningkatkan kemungkinan mendekati global optimum. Algoritma ini cukup efektif dalam menghindari local optima, karena algoritma ini secara bertahap memfokuskan pencarian pada solusi terbaik seiring waktu.

6. Genetic Algorithm

Dari eksperimen yang telah dilakukan, algoritma ini sama sekali tidak mendekati global optimum. Hal ini menjadi wajar sebab proses crossover serta *mutation* memang kurang cocok untuk permasalah ini. Bisa dibilang pertukaran yang dilakukan apabila harus mempertahankan 2 *parent* yang baik cukup sulit untuk mendapatkan hasil yang baik pula. Hal ini disebabkan karena hanya terdapat 1 solusi final dan terdapat banyak sekali *constraint* yang membatasi tidak baris,kolom, dan tiang dari kubus. Maka dari itu melakukan crossover berpotensi merusak konfigurasi baik yang telah didapat. Hal ini terlihat jelas dari *plot objective function* terbaik di tiap iterasinya yang terus naik turun. Tidak ada jaminan bahwa dalam permasalahan ini, apabila orangtua dari setiap cube baik, anaknya akan baik pula.

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

1. Steepest Ascent

- Hasil Terbaik : -774
- Kelebihan : Sederhana dan cepat
- Kekurangan : Mudah terjebak pada local maxima tanpa mekanisme untuk keluar

Steepest Ascent merupakan algoritma yang paling mungkin untuk berhenti di local maxima tanpa mencapai global optimum. Dibandingkan dengan algoritma seperti Stochastic Hill Climbing dan Simulated Annealing, yang lebih fleksibel, Steepest Ascent sering kali menghasilkan solusi akhir yang lebih jauh dari global optimum.

2. Hill-Climbing with Sideways Move

- Hasil Terbaik : -502
- Kelebihan : Dapat terus bergerak di area flat
- Kekurangan : Masih bisa terjebak di local optima jika terdapat flat yang terlalu panjang

Hill Climbing with Sideways Move memberikan perbaikan dibandingkan Steepest Ascent karena kemampuannya bergerak di area datar. Namun, hasilnya masih belum optimal jika dibandingkan dengan algoritma yang memiliki elemen eksplorasi lebih luas, seperti Simulated Annealing dan Genetic Algorithm.

3. Random Restart Hill Climbing

- Hasil Terbaik : -734
- Kelebihan : Dapat menghindari local optima dengan memulai ulang (restart) dari posisi acak
- Kekurangan : Efektivitas sangat bergantung pada jumlah restart yang dilakukan, dan tidak selalu menjamin mendekati global optimum

Random Restart sering kali menghasilkan hasil yang lebih baik dibandingkan Steepest Ascent atau Hill-Climbing with Sideways Move, karena dapat melompat keluar dari jebakan local maxima. Namun, hasilnya masih kurang optimal dibandingkan Simulated Annealing dan Genetic Algorithm

4. Stochastic Hill Climbing

- Hasil Terbaik : -442
- Kelebihan : Memiliki fleksibilitas dalam pemilihan langkah, sehingga lebih kecil kemungkinan untuk terjebak pada local maxima.
- Kekurangan : Pilihan langkah secara acak bisa menghasilkan ketidakpastian dan mungkin kurang efisien

Stochastic Hill Climbing memberikan hasil yang mendekati global optima dibandingkan dengan steepest ascent atau random restart.

5. Simulated Annealing

- Hasil Terbaik : -160
- Kelebihan : Efektif dalam menghindari local maxima dengan memilih langkah-langkah buruh di awal pencarian.
- Kekurangan : Memiliki parameter yang kompleks

Simulated Annealing memberikan hasil yang lebih baik dibandingkan semua algoritma hill climbing karena strategi eksplorasinya yang kuat. Ini juga biasanya mendekati global optimum lebih baik daripada Stochastic Hill Climbing.

6. Genetic Algorithm

- Hasil Terbaik : -4884

- Kelebihan: Relatif tidak ada untuk permasalahan ini, sebab memang kurang cocok untuk digunakan dalam permasalahan yang punya hasil akhir "spesifik"
- Kekurangan : Proses komputasi relatif lama dan juga dalam permasalahan ini, faktor *random* yang dimilikinya membuat terlalu tidak konsisten

Genetic Algorithm memberikan hasil yang paling buruk dari semua algoritma lainnya. Algoritma ini memang kurang cocok untuk permasalahan mencari *perfect magic cube*.

- Bagaimana perbandingan durasi proses pencarian tiap algoritma relatif terhadap algoritma lainnya?

Dari setiap algoritma yang digunakan, Stochastic Hill Climbing memberikan durasi relatif 8 iterasi proses/detik, dilanjutkan dengan Simulated Annealing dengan 17 Iterasi/detik . Lalu Algoritma Hill Climbing dan Genetic Algorithm dengan durasi proses pencarian yang lama dibandingkan dengan algoritma-algoritma lain.

Stochastic Hill Climbing memiliki waktu pencarian yang lebih singkat karena pendekatan acaknya dalam memilih langkah berikutnya di sekitar titik saat ini. Karena algoritma ini hanya memilih langkah yang memberikan sedikit peningkatan tanpa memperhatikan opsi terbaik, prosesnya berjalan cepat tanpa perlu mengevaluasi setiap langkah yang memungkinkan.

Simulated Annealing membutuhkan waktu yang lebih lama dibandingkan Stochastic Hill Climbing karena memiliki algoritma yang lebih kompleks dibandingkan dengan Stochastic Hill Climbing. Algoritma ini memperhitungkan probabilitas untuk mengambil langkah yang lebih buruk agar mendapatkan solusi yang lebih baik, proses ini membutuhkan durasi yang lebih lama dibandingkan dengan Stochastic Hill Climbing

Hill Climbing membutuhkan lebih banyak waktu per iterasi karena algoritma ini mengevaluasi setiap kemungkinan langkah untuk menemukan yang paling maksimal. Pendekatan ini secara menyeluruh mengevaluasi semua tetangga dalam ruang solusi, yang membutuhkan lebih banyak komputasi di setiap iterasi.

Genetic Algorithm memiliki durasi proses pencarian panjang karena kompleksitas langkah yang diambil dalam setiap iterasi atau generasi. Algoritma ini tidak hanya melakukan seleksi individu tetapi juga melalui proses crossover dan mutasi, yang masing-masing membutuhkan waktu pemrosesan. Dengan populasi besar atau banyaknya iterasi, Genetic Algorithm semakin lama seiring dengan meningkatnya eksplorasi ruang solusi.

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

Simulated Annealing memberikan hasil yang paling konsisten dibandingkan yang lainnya, dengan rata-rata hasil akhir fungsi objektif sekitar -100.0. Konsistensi ini dicapai karena pendekatan yang fleksibel dalam mengeksplorasi ruang solusi, menggunakan konsep "temperatur" yang secara perlahan menurun seiring waktu. Pada tahap awal, algoritma memiliki probabilitas yang lebih tinggi untuk menerima langkah yang lebih buruk (solusi yang kurang optimal) yang memungkinkan eksplorasi lebih luas dan menghindari local optima. Saat temperatur menurun, algoritma secara bertahap menjadi lebih selektif memilih solusi, mengarah pada pemilihan solusi yang lebih baik.

Algoritma-algoritma lain seperti Hill Climbing dan Genetic Algorithm kurang konsisten dengan hasil yang didapatkan. Algoritma Steepest Ascent dan Hill Climbing with Sideways Move sulit untuk menghindari local optima sehingga menyebabkan inkonsistensi hasil. Jika algoritma menemukan local maxima, ia terhenti pada solusi lokal tersebut, sehingga hasil akhirnya bisa sangat bergantung pada kondisi awal dan cenderung berakhir pada nilai objektif yang berbeda-beda setiap kali dijalankan.

Random Restart Hill Climbing juga sudah mencoba untuk menangani local optima dengan melakukan 'restart', namun hasil akhir sangat bergantung pada jumlah restart dan posisi awal suatu state. Oleh karena itu, algoritma mungkin tidak menemukan solusi terbaik di setiap percobaan, hal ini yang menyebabkan hasil yang inkonsisten.

Genetic Algorithm kurang konsisten karena sangat bergantung pada parameter populasi, jumlah generasi, dan probabilitas crossover dan probabilitas mutasi. Perubahan kecil pada nilai parameter ini dapat menghasilkan hasil yang berbeda-beda karena GA memulai pencarian dengan populasi acak. Jika parameter atau posisi awal kurang optimal, GA mungkin gagal menemukan solusi terbaik, atau membutuhkan lebih banyak generasi untuk mencapai solusi yang konsisten.

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

Banyaknya iterasi dalam Genetic Algorithm (GA) dapat meningkatkan konsistensi hasil. Hal ini terlihat dari rentang perbedaan antara nilai terbaik dan terburuk yang semakin kecil seiring bertambahnya jumlah iterasi. Dengan semakin banyak iterasi, peluang menemukan kombinasi crossover

antara individu-individu terbaik juga meningkat, sehingga kualitas solusi secara keseluruhan cenderung membaik.

Selain itu, penambahan jumlah populasi awal juga berdampak positif pada kualitas dan konsistensi hasil. Populasi yang lebih besar memberikan lebih banyak variasi state awal yang berkualitas, sehingga meningkatkan kemungkinan menghasilkan individu-individu yang lebih baik di generasi berikutnya. Kombinasi dari populasi awal yang besar dan jumlah iterasi yang tinggi memungkinkan GA untuk mengeksplorasi ruang solusi dengan lebih optimal, sehingga meningkatkan kualitas dan konsistensi hasil yang diperoleh.

BAB III

KESIMPULAN DAN SARAN

3.1 Kesimpulan

Pemecahan masalah Diagonal Magic Cube menggunakan berbagai algoritma Local Search, termasuk metode Hill Climbing, Simulated Annealing, dan Genetic Algorithm. Dengan konfigurasi cube berukuran $5 \times 5 \times 5$, tujuan utamanya adalah menemukan konfigurasi angka yang memenuhi kriteria jumlah angka yang sama di setiap baris, kolom, dan diagonal (315 untuk ukuran ini). Dari berbagai algoritma yang diuji, Simulated Annealing dipilih sebagai metode terbaik karena mampu menjelajahi ruang solusi yang kompleks dan menghindari solusi lokal yang kurang optimal. Pendekatan ini cocok untuk masalah yang memiliki banyak local maximum, seperti Diagonal Magic Cube, dan memungkinkan fleksibilitas untuk mengeksplorasi solusi yang luas di awal pencarian hingga mencapai solusi global yang lebih optimal.

3.2 Saran

Berikut beberapa saran untuk mengoptimalkan Algoritma Magic Square, perlu memperhatikan beberapa parameter dan aspek sebagai berikut.

- 1) Optimalkan Parameter, untuk Simulated Annealing, penyesuaian parameter seperti initial temperature dan cooling rate sangat penting untuk mendapatkan hasil terbaik. Menggunakan cooling rate sekitar 0.8-0.9 dapat memberikan keseimbangan antara eksplorasi dan efisiensi pencarian.
- 2) Gunakan Hybrid Approach, kombinasi antara Simulated Annealing dan Genetic Algorithm dapat dipertimbangkan, terutama untuk kasus yang lebih besar atau kompleks. Simulated Annealing dapat membantu menemukan konfigurasi awal yang baik, sementara Genetic Algorithm dapat mempercepat pencarian solusi optimal dengan mengeksplorasi generasi solusi yang berbeda.
- 3) Pertimbangkan Complete Search untuk Validasi. Meski beban komputasi Complete Search tinggi, metode ini bisa dijalankan untuk kasus dengan ukuran lebih kecil sebagai

langkah validasi. Ini akan memberikan gambaran mengenai efektivitas solusi yang dihasilkan oleh algoritma Local Search.

- 4) Skalabilitas dan Penerapan pada Ukuran Lain. Untuk pengembangan lebih lanjut, algoritma yang digunakan perlu diuji pada ukuran cube yang lebih besar, seperti $6 \times 6 \times 6$, guna menilai skalabilitas dan menyesuaikan parameter pencarian.

PEMBAGIAN TUGAS

Kegiatan	Nama (NIM)
Algoritma Steepest Ascent HC	Denise Felicia Tiowanni (13522013) Angelica Kierra Ninta Gurning (13522048) Immanuel Sebastian Girsang (13522058) Muhammad Naufal Aulia (13522074)
Algoritma HC with Sideways Move	Denise Felicia Tiowanni (13522013) Angelica Kierra Ninta Gurning (13522048) Immanuel Sebastian Girsang (13522058) Muhammad Naufal Aulia (13522074)
Algoritma Random Restart HC	Denise Felicia Tiowanni (13522013) Angelica Kierra Ninta Gurning (13522048) Immanuel Sebastian Girsang (13522058) Muhammad Naufal Aulia (13522074)
Algoritma Stochastic HC	Denise Felicia Tiowanni (13522013) Angelica Kierra Ninta Gurning (13522048) Immanuel Sebastian Girsang (13522058) Muhammad Naufal Aulia (13522074)
Algoritma Simulated Annealing	Denise Felicia Tiowanni (13522013) Angelica Kierra Ninta Gurning (13522048) Immanuel Sebastian Girsang (13522058) Muhammad Naufal Aulia (13522074)
Algoritma Genetic	Denise Felicia Tiowanni (13522013) Angelica Kierra Ninta Gurning (13522048) Immanuel Sebastian Girsang (13522058) Muhammad Naufal Aulia (13522074)
Visualisasi Plot	Denise Felicia Tiowanni (13522013)
Visualisasi State Kubus	Muhammad Naufal Aulia (13522074)
Strukturisasi Program	Immanuel Sebastian Girsang (13522058)
Bonus Video Player	Angelica Kierra Ninta Gurning (13522048)
Laporan	Denise Felicia Tiowanni (13522013) Amalia Putri (13522042) Angelica Kierra Ninta Gurning (13522048) Immanuel Sebastian Girsang (13522058) Muhammad Naufal Aulia (13522074)

README

Amalia Putri (13522042)

REFERENSI

Spesifikasi Tugas Besar 1 IF3070 Dasar Inteligensi Artifisial & IF3170 Intelligensi Ar...

- Wolfram Research. "Perfect magic cube. MathWorld" (online).
(<https://mathworld.wolfram.com/PerfectMagicCube.html>, diakses pada 28 September 2024).
- Trump, R. "Perfect magic cubes" (online).
(<https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>, diakses pada 28 September 2024).
- Leylia Khodra, Masayu. "Hill-Climbing Search" (online).
(https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804849395_IF3170_Materi3_Seg03_BeyondClassicalSearch_HillClimbing.pdf, diakses pada 28 September 2024).
- Leylia Khodra, Masayu. "Simulated Annealing" (online).
(https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804872404_IF3170_Materi03_Seg04_BeyondClassicalSearch_SimulatedAnnealing.pdf, diakses pada 28 September 2024).
- Leylia Khodra, Masayu. "Genetic Algorithm" (online).
(https://cdn-edunex.itb.ac.id/storages/files/1727405202098_IF3170_Materi03_Seg05_BeyondClassicalSearch.pdf, diakses pada 28 September 2024).