

**Tugas Besar 1 IF3070 Dasar Intelegensi Artifisial**  
**Pencarian Solusi Diagonal Magic Cube dengan Local Search**



**Disusun Oleh:**

**Kelompok 18**

Arvyno Pranata Limahardja	18222007
Bastian Natanael Sibarani	18222053
Naomi Pricilla Agustine	18222065
Micky Valentino	18222093

**Program Studi Sistem dan Teknologi Informasi**  
**Sekolah Teknik Elektro dan Informatika**  
**Institut Teknologi Bandung**

## DAFTAR ISI

<b>BAB I</b>	<b>3</b>
<b>DESKRIPSI PERSOALAN</b>	<b>3</b>
<b>BAB II</b>	<b>5</b>
<b>PEMBAHASAN</b>	<b>5</b>
2.1 Pemilihan Objective Function	5
2.2 Penjelasan Implementasi Algoritma Local Search	20
2.2.1 Steepest Ascent Hill-climbing	20
2.2.2 Hill-climbing with Sideways Move	24
2.2.3 Random Restart Hill-climbing	29
2.2.4 Stochastic Hill-climbing	32
2.2.5 Simulated Annealing	35
2.1.6 Genetic Algorithm	40
2.3 Hasil Eksperimen dan Analisis	48
2.2.1 Steepest Ascent Hill-climbing	48
2.2.2 Hill-climbing with Sideways Move	52
2.2.3 Random Restart Hill-climbing	57
2.2.4 Stochastic Hill-climbing	62
2.2.5 Simulated Annealing	67
2.1.6 Genetic Algorithm	73
<b>BAB III</b>	<b>109</b>
<b>KESIMPULAN DAN SARAN</b>	<b>109</b>
<b>BAB IV</b>	<b>110</b>
<b>PEMBAGIAN TUGAS</b>	<b>110</b>
<b>REFERENSI</b>	<b>111</b>

## BAB I

### DESKRIPSI PERSOALAN

*Diagonal magic cube* merupakan sebuah kubus yang memiliki ukuran  $n \times n \times n$  dengan angka 1 hingga  $n^3$  yang tersusun secara acak tanpa pengulangan dengan  $n$  adalah panjang dari setiap sisi yang terdapat pada kubus tersebut. Pada *diagonal magic cube*, terdapat satu angka yang merupakan *magic number* dari kubus yang dapat dicari melalui rumus sebagai berikut.

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

Keterangan :

$n$  : Panjang sisi kubus

*Magic number* merupakan angka yang tidak harus termasuk dalam rentang 1 hingga  $n^3$  dan tidak harus termasuk ke dalam angka yang dimasukkan ke dalam kubus. Untuk membentuk sebuah *diagonal magic cube* dalam keadaan yang ideal, terdapat beberapa syarat yang harus dipenuhi yaitu sebagai berikut.

- Jumlah angka untuk setiap baris sama dengan *magic number*
- Jumlah angka untuk setiap kolom sama dengan *magic number*
- Jumlah angka untuk setiap tiang sama dengan *magic number*
- Jumlah angka untuk setiap diagonal pada kubus sama dengan *magic number*
- Jumlah angka untuk setiap diagonal ruang dalam kubus sama dengan *magic number*

Permasalahan yang dihadapi pada tugas ini adalah mencari solusi *diagonal magic cube* agar dapat memenuhi kondisi ideal sesuai dengan syarat di atas dengan memanfaatkan algoritma *local search*. *Diagonal magic cube* yang akan digunakan adalah yang berukuran  $5 \times 5 \times 5$  atau dengan panjang dari setiap sisi yang terdapat pada kubus, yaitu 5. Pada setiap iterasi yang dilakukan dengan algoritma *local search*, langkah yang boleh dilakukan hanya dengan menukar posisi 2 angka pada kubus tersebut yang tidak harus bersebelahan, tetapi khusus untuk *genetic algorithm* diperbolehkan menukar posisi lebih dari 2 angka dalam sekali iterasi. Sebelum menentukan algoritma local search yang tepat untuk menyelesaikan permasalahan *diagonal magic cube*, penting untuk menentukan terlebih dahulu *objective function* yang akan digunakan

sebagai alat ukur untuk mengetahui seberapa jauh kondisi saat ini dengan kondisi ideal yang diinginkan yaitu *magic number*. Setelah itu, akan digunakan algoritma *local search* yang paling sesuai dalam memecahkan permasalahan ini.

## BAB II

### PEMBAHASAN

#### 2.1 Pemilihan Objective Function

*Objective function* digunakan untuk mengukur seberapa jauh kondisi saat ini dengan kondisi diagonal magic cube yang ideal yaitu jumlah angka pada setiap baris, kolom, tiang, diagonal ruang, dan seluruh diagonal pada suatu potongan dari kubus memiliki jumlah yang sama dengan *magic number*. Untuk menghitung *magic number* atau  $M$  dari sebuah *diagonal magic cube*, rumus yang akan digunakan untuk kubus dengan ukuran  $n \times n \times n$  dengan  $n$  adalah panjang setiap sisi kubus adalah sebagai berikut.

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

Keterangan :

$n$  : Panjang sisi kubus

Sesuai spesifikasi, permasalahan yang harus diselesaikan menggunakan *diagonal magic cube* berukuran  $5 \times 5 \times 5$  atau dengan panjang sisi kubus 5. Untuk kubus dengan panjang sisi 5 atau  $n = 5$ , berikut adalah *magic number* yang dimiliki oleh kubus tersebut.

$$M = \frac{n \times (n^3 + 1)}{2} = \frac{5 \times (5^3 + 1)}{2} = \frac{5 \times 126}{2} = \frac{630}{2} = 315$$

*Objective function* digunakan sebagai alat ukur evaluasi dengan menghitung nilai suatu keadaan kubus untuk mencari kombinasi angka dalam kubus yang memiliki akurasi terbaik. Dalam hal ini, pemilihan *objective function* kami lakukan melalui tiga tahap, yaitu penentuan metode, penentuan bobot, dan penentuan *objective function*.

#### a. Penentuan Metode

Hasil pengujian ditentukan dengan menghitung rata-rata dari hasil percobaan setiap kombinasi *objective function* dalam mengarahkan suatu algoritma *local search* yang dipilih, dalam hal ini adalah *stochastic hill-climbing*

dalam menghasilkan *diagonal magic cube* yang paling banyak memiliki jumlah elemen yang sesuai dengan *magic number*. Adapun terdapat

Rata-rata merupakan salah satu metode yang dipilih sebagai dasar dari *objective function*. Penentuan hasil terbaik adalah ketika setiap elemen dari kubus (baris, kolom, tiang, diagonal, diagonal ruang) memiliki jumlah yang sama dengan *magic number*-nya yaitu sebesar 315 atau bisa dibilang bahwa selisih jumlah setiap elemen dengan *magic number* yang dimiliki kubus harus sama dengan nol. Rata-rata dapat digunakan untuk mengukur seberapa baik solusi dalam memenuhi kriteria atau tujuan yang diharapkan dengan mencari tahu rata-rata selisih dari setiap elemen yang ada di dalam kubus tersebut. Rata-rata selisih yang dicari adalah rata-rata yang sama dengan nol atau mendekati nol.

Standar deviasi menunjukkan seberapa jauh nilai-nilai selisih dari jumlah tiap elemen dari suatu kubus dengan *magic number*-nya. Penentuan hasil terbaik adalah dengan mencari selisih standar deviasi terkecil, yang menandakan tingkat keseragaman yang tinggi pada kubus dan dinilai lebih dekat menuju konfigurasi yang ideal karena elemen-elemen lebih terpusat di sekitar *magic number*.

```
def cek_spesifikasi_tes_algo(n, kubus, w1, w2, jenis_objektif):
    magic_number = hitung_magic_number(n)
    jumlah_315 = 0 # Jumlah elemen yang sama dengan 315
    frekuensi_pemeriksaan = 0 # Banyak iterasi dalam proses
    pemeriksaan selisih dan skor pada baris, kolom, tiang, diagonal sisi,
    dan diagonal ruang

    arr_selisih = [] # Menyimpan semua jumlah untuk perhitungan
    standar deviasi

    # Memeriksa skor pada baris di setiap layer pada kubus
    for i in range(n):
        for j in range(n):
            jumlah_baris = 0
            for k in range(n):
                jumlah_baris += kubus[i][j][k]
            if jumlah_baris == magic_number:
                jumlah_315 += 1
```

```

        selisih_baris = abs(jumlah_baris - magic_number)
        arr_sелих.append(sелих_baris) # Menyimpan selisih
baris
        frekuensi_pemeriksaan += 1

# Memeriksa skor pada kolom di setiap layer pada kubus
for i in range(n):
    for k in range(n):
        jumlah_kolom = 0
        for j in range(n):
            jumlah_kolom += kubus[i][j][k]
        if jumlah_kolom == magic_number:
            jumlah_315 += 1
        selisih_kolom = abs(jumlah_kolom - magic_number)
        arr_sелих.append(sелих_kolom) # Menyimpan selisih
kolom
        frekuensi_pemeriksaan += 1

# Memeriksa skor pada tiang di setiap layer pada kubus
for j in range(n):
    for k in range(n):
        jumlah_tiang = 0
        for i in range(n):
            jumlah_tiang += kubus[i][j][k]
        if jumlah_tiang == magic_number:
            jumlah_315 += 1
        selisih_tiang = abs(jumlah_tiang - magic_number)
        arr_sелих.append(sелих_tiang) # Menyimpan selisih
kolom
        frekuensi_pemeriksaan += 1

# Memeriksa diagonal sisi pada layer di sumbu-x dan sumbu-y
(diagonal sisi 1)
for i in range(n):
    # Diagonal sisi 1a: dari (i = [0 - n-1], j = 0, k = 0) ke (i
= [0 - n-1], j = n-1, k = n-1)
    jumlah_diagonal_sisi = 0
    for j in range(n):
        jumlah_diagonal_sisi += kubus[i][j][j]

```

```

        if jumlah_diagonal_sisi == magic_number:
            jumlah_315 += 1
            selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -
magic_number)
            arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
            frekuensi_pemeriksaan += 1

            # Diagonal sisi 1b: dari (i = [0 - n-1], j = 0, k = n-1) ke
(i = [0 - n-1], j = n-1, k = 0)
            jumlah_diagonal_sisi = 0
            for j in range(n):
                jumlah_diagonal_sisi += kubus[i][j][n-1-j]
            if jumlah_diagonal_sisi == magic_number:
                jumlah_315 += 1
                selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -
magic_number)
                arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
                frekuensi_pemeriksaan += 1

            # Memeriksa diagonal sisi pada layer di sumbu-x dan sumbu-z
(diagonal sisi 2)
            for i in range(n):
                # Diagonal sisi 2a: dari (i = 0, j = [0 - n-1], k = 0) ke (i
= n-1, j = [0 - n-1], k = n-1)
                jumlah_diagonal_sisi = 0
                for j in range(n):
                    jumlah_diagonal_sisi += kubus[j][i][j]
                if jumlah_diagonal_sisi == magic_number:
                    jumlah_315 += 1
                    selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -
magic_number)
                    arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
                    frekuensi_pemeriksaan += 1

                # Diagonal sisi 2b: dari (i = 0, j = [0 - n-1], k = n-1) ke
(i = n-1, j = [0 - n-1], k = 0)

```

```

jumlah_diagonal_sisi = 0
for j in range(n):
    jumlah_diagonal_sisi += kubus[j][i][n-1-j]
if jumlah_diagonal_sisi == magic_number:
    jumlah_315 += 1
selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -
magic_number)
arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
frekuensi_pemeriksaan += 1

# Memeriksa diagonal sisi pada layer di sumbu-y dan sumbu-z
(diagonal sisi 3)
for i in range(n):
    # Diagonal sisi 3a: dari (i = 0, j = 0, k = [0 - n-1]) ke (i
= n-1, j = n-1, k = [0 - n-1])
    jumlah_diagonal_sisi = 0
    for j in range(n):
        jumlah_diagonal_sisi += kubus[j][j][i]
    if jumlah_diagonal_sisi == magic_number:
        jumlah_315 += 1
    selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -
magic_number)
    arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
    frekuensi_pemeriksaan += 1

    # Diagonal sisi 3b: dari (i = 0, j = n-1, k = [0 - n-1]) ke
(i = n-1, j = 0, k = [0 - n-1])
    jumlah_diagonal_sisi = 0
    for j in range(n):
        jumlah_diagonal_sisi += kubus[j][n-1-j][i]
    if jumlah_diagonal_sisi == magic_number:
        jumlah_315 += 1
    selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -
magic_number)
    arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
    frekuensi_pemeriksaan += 1

```

```
# Memeriksa diagonal ruang
# Diagonal ruang 1: dari (0,0,0) ke (n-1,n-1,n-1)
jumlah_diagonal_ruang = 0
for i in range(n):
    jumlah_diagonal_ruang += kubus[i][i][i]
if jumlah_diagonal_ruang == magic_number:
    jumlah_315 += 1
selisih_diagonal_ruang = abs(jumlah_diagonal_ruang -
magic_number)
arr_sелих.append(sелих_diagonal_ruang) # Menyimpan selisih
diagonal ruang
frekuensi_pemeriksaan += 1

# Diagonal ruang 2: dari (0,0,n-1) ke (n-1,n-1,0)
jumlah_diagonal_ruang = 0
for i in range(n):
    jumlah_diagonal_ruang += kubus[i][i][n-1-i]
if jumlah_diagonal_ruang == magic_number:
    jumlah_315 += 1
selisih_diagonal_ruang = abs(jumlah_diagonal_ruang -
magic_number)
arr_sелих.append(sелих_diagonal_ruang) # Menyimpan selisih
diagonal ruang
frekuensi_pemeriksaan += 1

# Diagonal ruang 3: dari (0,n-1,0) ke (n-1,0,n-1)
jumlah_diagonal_ruang = 0
for i in range(n):
    jumlah_diagonal_ruang += kubus[i][n-1-i][i]
if jumlah_diagonal_ruang == magic_number:
    jumlah_315 += 1
selisih_diagonal_ruang = abs(jumlah_diagonal_ruang -
magic_number)
arr_sелих.append(sелих_diagonal_ruang) # Menyimpan selisih
diagonal ruang
frekuensi_pemeriksaan += 1

# Diagonal ruang 4: dari (0,n-1,n-1) ke (n-1,0,0)
```

```

jumlah_diagonal_ruang = 0
for i in range(n):
    jumlah_diagonal_ruang += kubus[i][n-1-i][n-1-i]
if jumlah_diagonal_ruang == magic_number:
    jumlah_315 += 1
selisih_diagonal_ruang = abs(jumlah_diagonal_ruang -
magic_number)
arr_selisih.append(selisih_diagonal_ruang) # Menyimpan selisih
diagonal ruang
frekuensi_pemeriksaan += 1

# Menghitung skor evaluasi fungsi objektif berdasarkan
jenis_objektif
if (str.lower(jenis_objektif) == "avg"):
    # Target primer (w1%): Persentase jumlah garis yang tepat sama
dengan magic number
    skor_target_primer = (jumlah_315 / frekuensi_pemeriksaan) * w1

    # Target sekunder (w2%): Menggunakan konsep mean untuk menilai
derajat deviasi jumlah terhadap magic number
    avg_selisih = sum(arr_selisih) / len(arr_selisih)
    maks_kemungkinan_selisih = magic_number - 15 # Selisih
maksimum yang mungkin terjadi
    rasio_kedekatan = 1 - (avg_selisih / maks_kemungkinan_selisih)
    skor_target_sekunder = rasio_kedekatan * w2

    # Total skor
persentase_sukses = skor_target_primer + skor_target_sekunder

    # Jenis_objektif adalah standar deviasi (std) untuk target
sekunder
elif (str.lower(jenis_objektif) == "std"):
    # Target primer (w1%): Persentase jumlah garis yang tepat sama
dengan magic number
    skor_target_primer = (jumlah_315 / frekuensi_pemeriksaan) * w1

    # Target sekunder (w2%): Menggunakan konsep standar deviasi
untuk menilai derajat deviasi jumlah terhadap magic number

```

```

# Menghitung mean dari daftar arr_selisih
avg_selisih = sum(arr_selisih) / len(arr_selisih)

# Menghitung variansi
variansi = sum((x - avg_selisih) ** 2 for x in arr_selisih) /
frekuensi_pemeriksaan

# Menghitung standar deviasi sebagai akar dari variansi
std_dev = variansi ** 0.5

# Normalisasi dengan magic_number sebagai batas deviasi
maksimum
maks_std_dev = magic_number - 15
rasio_kedekatan = 1 - (std_dev / maks_std_dev)
skor_target_sekunder = rasio_kedekatan * w2

# Total skor
persentase_sukses = skor_target_primer + skor_target_sekunder

return round(persentase_sukses, 3), jumlah_315

```

```

def algo_tes_objective_function(n, maks_iterasi, threshold_akurasi,
kubus, jenis_objektif, w1, w2): # Pilihan algoritma adalah stochastic
hill-climbing
    # Menginisialisasi kubus_terbaik dengan kubus yang menjadi input
fungsi
    kubus_terbaik = kubus

    # Menginisialisasi skor_kubus_terbaik dengan skor kubus yang
menjadi input fungsi
    skor_kubus_terbaik, jumlah_315_terbaik =
cek_spesifikasi_tes_algo(n, kubus_terbaik, w1, w2, jenis_objektif)

    # Menginisialisasi jumlah iterasi dengan nilai 0
    iterasi = 0

    # Melakukan iterasi hingga mencapai batas maksimum atau threshold
akurasi

```

```

while (iterasi < maks_iterasi):
    # Membuat konfigurasi kubus baru dengan menukar angka secara
    acak
    kubus_baru = swap_angka(n, np.copy(kubus_terbaik))
    skor_kubus_baru, jumlah_315_baru =
    cek_spesifikasi_tes_algo(n, kubus_baru, w1, w2, jenis_objektif)
    iterasi += 1

    # Melakukan pembaruan jika ditemukan konfigurasi yang lebih
    baik
    if skor_kubus_baru > skor_kubus_terbaik:
        skor_kubus_terbaik = skor_kubus_baru
        jumlah_315_terbaik = jumlah_315_baru
        kubus_terbaik = np.copy(kubus_baru)

    # Menghentikan iterasi jika sudah mencapai threshold
    if (skor_kubus_terbaik >= threshold_akurasi):
        break

return jumlah_315_terbaik

```

### b. Penentuan Bobot

Tujuan utama dari *objective function* adalah untuk menentukan solusi terbaik di mana definisi dari solusi terbaik itu adalah kubus yang setiap elemennya ketika dijumlahkan akan setara dengan *magic number*. Kedua metode di atas digunakan untuk membantu menemukan solusi terbaik, menemukan kubus yang diharapkan mendekati keadaan solusi terbaik namun bukan sebagai penentu penilaian utama. Jadi, dibutuhkan pembobotan agar *objective function* lebih mendahulukan menemukan kubus dengan jumlah setiap elemen sama dengan *magic number* daripada menemukan kubus yang mendekati solusi terbaik. Ditentukan pembobotan untuk penemuan kubus dengan jumlah elemen yang sama dengan *magic number* sebagai penentu penilaian utama dengan pembobotan minimal 70% dan penemuan kubus yang mendekati solusi terbaik sebagai penilaian sekunder dari *objective function* dengan pembobotan maksimal 30%. Variasi dari pembobotan yang dipakai adalah {70%, 30%}, {80%, 20%}, dan {90%, 10%}.

```

def tes_objective_function(n, kubus):
    # Inisialisasi array untuk menyimpan hasil pengujian
    arr_jumlah_315 = []
    # Inisialisasi matriks untuk menyimpan riwayat pengujian
    histori_tes = [[None, None, None, None] for i in range (6)]

    # Mendefinisikan parameter-parameter yang akan diuji
    w1 = [70, 80, 90] # Bobot target primer
    w2 = [30, 20, 10] # Bobot target sekunder
    jenis_objektif = ["avg", "std"] # Jenis fungsi objektif

    # Melakukan pengujian untuk setiap kombinasi parameter
    for i in range (2):
        for j in range (3):
            arr_jumlah_315 = []
            avg_jumlah_315 = 0
            # Melakukan 100 kali percobaan untuk setiap kombinasi
            for k in range (100):
                jumlah_315_tes = algo_tes_objective_function(n, 100, 100,
kubus, jenis_objektif[i], w1[j], w2[j])
                arr_jumlah_315.append(jumlah_315_tes)
            # Menghitung rata-rata hasil percobaan
            avg_jumlah_315 = sum(arr_jumlah_315) / len(arr_jumlah_315)
            histori_tes.append([jenis_objektif[i], w1[j], w2[j],
avg_jumlah_315])

    # Mencari kombinasi parameter terbaik
    maks_avg_jumlah_315 = -1
    maks_elemen = None

    # Mencari nilai rata-rata tertinggi dari seluruh kombinasi
    for elemen in histori_tes:
        if (elemen[3] is not None) and (elemen[3] >
maks_avg_jumlah_315):
            maks_avg_jumlah_315 = elemen[3]
            maks_elemen = elemen

    return maks_avg_jumlah_315, maks_elemen

```

### c. Penentuan *Objective Function*

Tahap terakhir adalah untuk menentukan kombinasi pembobotan serta metode yang paling optimal untuk digunakan dengan memaksimalkan nilai rata-rata hasil pengujian dari beberapa percobaan. Percobaan akan dilakukan dengan algoritma *stochastic hill-climbing* yang akan diiterasi dalam jumlah tertentu, dengan setiap hasil dari kombinasi *objective function* akan disimpan dan dirata-rata lalu dibandingkan untuk menentukan kombinasi metode dan pembobotan terbaik.

```
▶ n = 5
kubus = inisialisasi_kubus(n)
maks_avg_jumlah_315, maks_elelen = tes_objective_function(n, kubus)

# Menampilkan hasil pengujian terbaik
print(f"Nilai maksimum mean jumlah magic number: {maks_avg_jumlah_315}")
print(f"Objective function terbaik:\nJenis Objektif = avg\nBobot Target Primer = {maks_elelen[0]}\nBobot Target Sekunder = {maks_elelen[2]}")

⇒ Nilai maksimum mean jumlah magic number: 4.18
Objective function terbaik:
Jenis Objektif = avg
Bobot Target Primer = 80
Bobot Target Sekunder = 20
```

```
def cek_spesifikasi(n, kubus):
    magic_number = hitung_magic_number(n)
    jumlah_315 = 0 # Jumlah elemen yang sama dengan 315
    frekuensi_pemeriksaan = 0 # Banyak iterasi dalam proses
    pemeriksaan selisih dan skor pada baris, kolom, tiang, diagonal sisi,
    dan diagonal ruang

    arr_selisih = [] # Menyimpan semua jumlah untuk perhitungan
    standar deviasi

    # Memeriksa skor pada baris di setiap layer pada kubus
    for i in range(n):
        for j in range(n):
            jumlah_baris = 0
            for k in range(n):
                jumlah_baris += kubus[i][j][k]
            if jumlah_baris == magic_number:
                jumlah_315 += 1
            selisih_baris = abs(jumlah_baris - magic_number)
            arr_selisih.append(selisih_baris) # Menyimpan selisih
    baris
```

```

frekuensi_pemeriksaan += 1

# Memeriksa skor pada kolom di setiap layer pada kubus
for i in range(n):
    for k in range(n):
        jumlah_kolom = 0
        for j in range(n):
            jumlah_kolom += kubus[i][j][k]
        if jumlah_kolom == magic_number:
            jumlah_315 += 1
        selisih_kolom = abs(jumlah_kolom - magic_number)
        arr_sелих.append(sелих_kolom) # Menyimpan selisih
kolom
        frekuensi_pemeriksaan += 1

# Memeriksa skor pada tiang di setiap layer pada kubus
for j in range(n):
    for k in range(n):
        jumlah_tiang = 0
        for i in range(n):
            jumlah_tiang += kubus[i][j][k]
        if jumlah_tiang == magic_number:
            jumlah_315 += 1
        selisih_tiang = abs(jumlah_tiang - magic_number)
        arr_sелих.append(sелих_tiang) # Menyimpan selisih
kolom
        frekuensi_pemeriksaan += 1

# Memeriksa diagonal sisi pada layer di sumbu-x dan sumbu-y
(diagonal sisi 1)
for i in range(n):
    # Diagonal sisi 1a: dari (i = [0 - n-1], j = 0, k = 0) ke (i
    = [0 - n-1], j = n-1, k = n-1)
    jumlah_diagonal_sisi = 0
    for j in range(n):
        jumlah_diagonal_sisi += kubus[i][j][j]
    if jumlah_diagonal_sisi == magic_number:
        jumlah_315 += 1
    selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -

```

```

magic_number)
    arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
    frekuensi_pemeriksaan += 1

        # Diagonal sisi 1b: dari (i = [0 - n-1], j = 0, k = n-1) ke
(i = [0 - n-1], j = n-1, k = 0)
        jumlah_diagonal_sisi = 0
        for j in range(n):
            jumlah_diagonal_sisi += kubus[i][j][n-1-j]
        if jumlah_diagonal_sisi == magic_number:
            jumlah_315 += 1
        selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -
magic_number)
        arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
        frekuensi_pemeriksaan += 1

        # Memeriksa diagonal sisi pada layer di sumbu-x dan sumbu-z
(diagonal sisi 2)
        for i in range(n):
            # Diagonal sisi 2a: dari (i = 0, j = [0 - n-1], k = 0) ke (i
= n-1, j = [0 - n-1], k = n-1)
            jumlah_diagonal_sisi = 0
            for j in range(n):
                jumlah_diagonal_sisi += kubus[j][i][j]
            if jumlah_diagonal_sisi == magic_number:
                jumlah_315 += 1
            selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -
magic_number)
            arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
            frekuensi_pemeriksaan += 1

            # Diagonal sisi 2b: dari (i = 0, j = [0 - n-1], k = n-1) ke
(i = n-1, j = [0 - n-1], k = 0)
            jumlah_diagonal_sisi = 0
            for j in range(n):
                jumlah_diagonal_sisi += kubus[j][i][n-1-j]

```

```

        if jumlah_diagonal_sisi == magic_number:
            jumlah_315 += 1
            selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -
magic_number)
            arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
            frekuensi_pemeriksaan += 1

        # Memeriksa diagonal sisi pada layer di sumbu-y dan sumbu-z
(diagonal sisi 3)
        for i in range(n):
            # Diagonal sisi 3a: dari (i = 0, j = 0, k = [0 - n-1]) ke (i
= n-1, j = n-1, k = [0 - n-1])
            jumlah_diagonal_sisi = 0
            for j in range(n):
                jumlah_diagonal_sisi += kubus[j][j][i]
            if jumlah_diagonal_sisi == magic_number:
                jumlah_315 += 1
                selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -
magic_number)
                arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
                frekuensi_pemeriksaan += 1

            # Diagonal sisi 3b: dari (i = 0, j = n-1, k = [0 - n-1]) ke
(i = n-1, j = 0, k = [0 - n-1])
            jumlah_diagonal_sisi = 0
            for j in range(n):
                jumlah_diagonal_sisi += kubus[j][n-1-j][i]
            if jumlah_diagonal_sisi == magic_number:
                jumlah_315 += 1
                selisih_diagonal_sisi = abs(jumlah_diagonal_sisi -
magic_number)
                arr_selisih.append(selisih_diagonal_sisi) # Menyimpan selisih
diagonal sisi
                frekuensi_pemeriksaan += 1

        # Memeriksa diagonal ruang
        # Diagonal ruang 1: dari (0,0,0) ke (n-1,n-1,n-1)

```

```

jumlah_diagonal_ruang = 0
for i in range(n):
    jumlah_diagonal_ruang += kubus[i][i][i]
if jumlah_diagonal_ruang == magic_number:
    jumlah_315 += 1
selisih_diagonal_ruang = abs(jumlah_diagonal_ruang -
magic_number)
arr_sелих.append(sелих_diagonal_ruang) # Menyimpan selisih
diagonal ruang
frekuensi_pemeriksaan += 1

# Diagonal ruang 2: dari (0,0,n-1) ke (n-1,n-1,0)
jumlah_diagonal_ruang = 0
for i in range(n):
    jumlah_diagonal_ruang += kubus[i][i][n-1-i]
if jumlah_diagonal_ruang == magic_number:
    jumlah_315 += 1
selisih_diagonal_ruang = abs(jumlah_diagonal_ruang -
magic_number)
arr_sелих.append(sелих_diagonal_ruang) # Menyimpan selisih
diagonal ruang
frekuensi_pemeriksaan += 1

# Diagonal ruang 3: dari (0,n-1,0) ke (n-1,0,n-1)
jumlah_diagonal_ruang = 0
for i in range(n):
    jumlah_diagonal_ruang += kubus[i][n-1-i][i]
if jumlah_diagonal_ruang == magic_number:
    jumlah_315 += 1
selisih_diagonal_ruang = abs(jumlah_diagonal_ruang -
magic_number)
arr_sелих.append(sелих_diagonal_ruang) # Menyimpan selisih
diagonal ruang
frekuensi_pemeriksaan += 1

# Diagonal ruang 4: dari (0,n-1,n-1) ke (n-1,0,0)
jumlah_diagonal_ruang = 0
for i in range(n):
    jumlah_diagonal_ruang += kubus[i][n-1-i][n-1-i]

```

```

if jumlah_diagonal_ruang == magic_number:
    jumlah_315 += 1
    selisih_diagonal_ruang = abs(jumlah_diagonal_ruang -
magic_number)
    arr_sелих.append(sелих_diagonal_ruang) # Menyimpan selisih
diagonal ruang
    frekuensi_pemeriksaan += 1

# Target primer (w1%): Persentase jumlah garis yang tepat sama
dengan magic number
skor_target_primer = (jumlah_315 / frekuensi_pemeriksaan) * 80

# Target sekunder (w2%): Menggunakan konsep mean untuk menilai
derajat deviasi jumlah terhadap magic number
avg_sелих = sum(arr_sелих) / len(arr_sелих)
maks_kemungkinan_sелих = magic_number - 15 # Selisih maksimum
yang mungkin terjadi
rasio_kedekatan = 1 - (avg_sелих / maks_kemungkinan_sелих)
skor_target_sekunder = rasio_kedekatan * 20

# Total skor
persentase_sukses = skor_target_primer + skor_target_sekunder

return round(persentase_sukses, 3), jumlah_315

```

## 2.2 Penjelasan Implementasi Algoritma *Local Search*

### 2.2.1 *Steepest Ascent Hill-climbing*

Implementasi dari algoritma *Steepest Ascent Hill-climbing* dilakukan dengan memeriksa setiap tetangga dari konfigurasi kubus yang saat ini diperiksa dan mencari solusi terbaik yaitu tetangga dengan skor tertinggi yang akan dipilih sebagai kondisi baru. Proses ini akan diulang hingga tidak ada tetangga yang memiliki skor yang lebih baik dari kondisi kubus terbaik saat ini.

Sebelum mulai proses iterasi, terdapat satu fungsi yang dibutuhkan dalam pengaplikasian algoritma *Steepest Ascent Hill-climbing* yaitu fungsi

generate\_best\_neighbors\_steepest yang bertujuan untuk mencari tetangga terbaik dari kubus dengan cara menukar dua posisi elemen pada kubus untuk mendapatkan tetangga baru. Tahap awal dari fungsi ini adalah menyalin kubus awal sebagai kubus terbaik sementara dan menghitung skor terbaik sementara. Selanjutnya, iterasi akan dilakukan untuk memeriksa semua kemungkinan tetangga dengan menukar posisi dua elemen di kubus untuk menghasilkan kombinasi tetangga yang baru dan akan dihitung skornya dengan menggunakan fungsi cek\_spesifikasi. Jika tetangga baru memiliki skor yang lebih baik, kubus terbaik dan skornya akan diperbarui dengan tetangga yang terbaik serta nilai improvisasi\_skor akan menjadi *true*.

```
def generate_best_neighbors_steepest(n, kubus):
    # Menginisialisasi kubus terbaik dengan kubus yang menjadi input
    fungsi:
        kubus_terbaik = np.copy(kubus)
        skor_kubus_terbaik, jumlah_315_terbaik = cek_spesifikasi(n,
        kubus_terbaik)
        improvisasi_skor = False

    # Memeriksa semua kemungkinan neighbor
    for i in range (n**3):
        for j in range (i + 1, n**3):
            # Menginisialisasi neighbor dari kubus yang menjadi input
            fungsi dengan melakukan pertukaran indeks i dan j
            kubus_baru = np.copy(kubus)
            posisi1 = np.unravel_index(i, (n, n, n))
            posisi2 = np.unravel_index(j, (n, n, n))
            kubus_baru[posisi1], kubus_baru[posisi2] =
            kubus_baru[posisi2], kubus_baru[posisi1]

            # Menghitung skor kubus neighbor
            skor_kubus_baru, jumlah_315_baru = cek_spesifikasi(n,
            kubus_baru)

            # Jika skor baru lebih baik, update kubus terbaik
            if skor_kubus_baru > skor_kubus_terbaik:
                kubus_terbaik = np.copy(kubus_baru)
```

```

skor_kubus_terbaik = skor_kubus_baru
jumlah_315_terbaik = jumlah_315_baru
improvisasi_skor = True

return kubus_terbaik, improvisasi_skor

```

Fungsi tersebut kemudian digunakan dalam fungsi utama, yaitu `steepest_ascent_hill_climbing`. Dalam proses ini, setiap riwayat skor kubus dan jumlah *magic number* akan disimpan dalam sebuah *array*. Pada tahap awal, inisialisasi kubus terbaik akan dilakukan dengan menyalin kubus awal dan menghitung skor awal sebagai skor terbaik sementara dengan fungsi `cek_spesifikasi`. Selanjutnya, proses iterasi akan dilakukan dengan menjalankan fungsi `generate_best_neighbors_steepest` untuk mencari tetangga terbaik. Jika ditemukan perbaikan, kubus terbaik akan diubah menjadi kubus yang baru, tetapi jika tidak ada lagi perbaikan skor yang dapat dilakukan atau jika sudah mencapai skor sempurna, iterasi akan dihentikan. Selain itu, algoritma ini juga akan mencatat waktu eksekusi dan riwayat setiap perbaikan agar hasil akhirnya mencakup informasi iterasi yang terjadi dan waktu yang diperlukan kubus.

```

def steepest_ascent_hill_climbing(n, kubus):
    # Memulai perhitungan waktu
    start_time = time.perf_counter()

    # Menginisialisasi array history untuk menyimpan informasi
    histori = {
        'skor_kubus_terbaik': None,
        'jumlah_315_terbaik': None,
        'kubus_terbaik': kubus,
        'iterasi': None,
        'skor_kubus': [],
        'jumlah_315': [],
        'elapsed_time': None
    }

    # Menginisialisasi kubus_terbaik dengan kubus yang menjadi input
    fungsi

```

```
kubus_terbaik = np.copy(kubus)

# Menginisialisasi skor_kubus_terbaik dengan skor kubus yang
menjadi input fungsi
skor_kubus_terbaik, jumlah_315_terbaik = cek_spesifikasi(n,
kubus_terbaik)

# Menghitung jumlah iterasi
iterasi = 0

# Melakukan iterasi sampai tidak ada perbaikan skor
while (True):
    # Memilih neighbor terbaik dari setiap kemungkinan neighbor
    # dari current state kubus_terbaik
    kubus_baru, improvisasi_skor =
generate_best_neighbors_steepest(n, kubus_terbaik)

    # Menghitung skor kubus baru
    skor_kubus_baru, jumlah_315_baru = cek_spesifikasi(n,
kubus_baru)
    iterasi += 1

    # Menyimpan informasi kubus baru
    histori['skor_kubus'].append(skor_kubus_baru)
    histori['jumlah_315'].append(jumlah_315_baru)

    if not improvisasi_skor:
        break

    kubus_terbaik = np.copy(kubus_baru)
    skor_kubus_terbaik = skor_kubus_baru
    jumlah_315_terbaik = jumlah_315_baru

    # Jika sudah mencapai skor sempurna (100%), hentikan iterasi
    if skor_kubus_terbaik == 100:
        break

    # Update array histori
    histori['skor_kubus_terbaik'] = skor_kubus_terbaik
```

```

histori['jumlah_315_terbaik'] = jumlah_315_terbaik
histori['kubus_terbaik'] = kubus_terbaik
histori['iterasi'] = iterasi

# Menghitung hasil timing
end_time = time.perf_counter()
elapsed_time = end_time - start_time
histori['elapsed_time'] = round(elapsed_time, 3)

return histori

```

### 2.2.2 *Hill-climbing with Sideways Move*

Algoritma *local search* ini merupakan variasi dari algoritma *Steepest Ascent Hill-climbing* yang akan mencari seluruh kemungkinan *neighbor* dari *current state* dan akan berpindah ke *neighbor* yang memiliki akurasi yang lebih baik dari sebelumnya. Namun, kemungkinan untuk terjebak di local optima tidak dapat ditangani bila menggunakan metode *Steepest Ascent Hill-climbing*. Oleh karena itu, algoritma *Hill-climbing with Sideways Move* ini memperbaiki kembali metode dari *Steepest Ascent Hill-climbing* dengan algoritma yang dapat mempertimbangkan *neighbor* yang memiliki akurasi yang sama dengan *current state*. Dengan perubahan ini, kemungkinan terjebak di dalam *local optima* akan berkurang.

Dalam pengimplementasian algoritma ini, dibutuhkan beberapa tahap yang digunakan untuk membantu pengoptimalan algoritma ini. Tahap-tahap tersebut adalah sebagai berikut.

- Pembuatan Fungsi Pencarian *Neighbor*

Sebelum mulai proses iterasi, terdapat satu fungsi yang dibutuhkan dalam pengaplikasian algoritma *Hill-climbing with Sideways Move* yaitu fungsi *generate\_best\_neighbors\_sideways* yang bertujuan untuk mencari tetangga terbaik dari kubus dengan cara menukar dua posisi elemen pada kubus untuk mendapatkan tetangga baru. Tahap awal dari fungsi ini adalah

menyalin kubus awal sebagai kubus terbaik sementara dan menghitung skor terbaik sementara. Selanjutnya, iterasi akan dilakukan untuk memeriksa semua kemungkinan tetangga dengan menukar posisi dua elemen di kubus untuk menghasilkan kombinasi tetangga yang baru dan akan dihitung skornya dengan menggunakan fungsi `cek_spesifikasi`. Jika tetangga baru memiliki skor yang lebih baik atau setidaknya sama dengan `jumlah_315` yang lebih baik, kubus terbaik dan skornya akan diperbarui dengan tetangga yang terbaik serta nilai `improvisasi_skor` akan menjadi *true*.

```
def random_restart_hill_climbing(n, maks_restart, threshold_akurasi, kubus):
    # Memulai perhitungan waktu
    start_time = time.perf_counter()

    # Menginisialisasi array history untuk menyimpan informasi
    histori = {
        'skor_kubus_terbaik': None,
        'jumlah_315_terbaik': None,
        'kubus_terbaik': kubus,
        'restart': None,
        'iterasi': [],
        'skor_kubus': [],
        'jumlah_315': [],
        'elapsed_time': None
    }

    # Menginisialisasi kubus_terbaik dengan kubus yang menjadi input
    # fungsi
    kubus_terbaik = np.copy(kubus)

    # Menginisialisasi skor_kubus_terbaik dengan skor kubus yang
    # menjadi input fungsi
    skor_kubus_terbaik, jumlah_315_terbaik = cek_spesifikasi(n,
        kubus_terbaik)

    # Menghitung jumlah iterasi
```

```

restart = 0

while (restart < maks_restart):
    # Membuat konfigurasi kubus baru secara acak untuk restart
    kubus_random = inisialisasi_kubus(n)

        # Melakukan hill climbing dari konfigurasi acak ini sampai
        mencapai local optimum
        histori_kubus_climbing = steepest_ascent_hill_climbing(n,
        kubus_random)
        restart += 1
        iterasi = histori_kubus_climbing['iterasi']
        skor_kubus_climbing =
        histori_kubus_climbing['skor_kubus_terbaik']
        jumlah_315_climbing =
        histori_kubus_climbing['jumlah_315_terbaik']
        kubus_climbing = histori_kubus_climbing['kubus_terbaik']

        # Menyimpan informasi kubus baru
        histori['iterasi'].append(iterasi)
        histori['skor_kubus'].append(skor_kubus_climbing)
        histori['jumlah_315'].append(jumlah_315_climbing)

        # Jika skor baru lebih baik, update kubus terbaik
        if (skor_kubus_climbing > skor_kubus_terbaik):
            kubus_terbaik = np.copy(kubus_climbing)
            skor_kubus_terbaik = skor_kubus_climbing
            jumlah_315_terbaik = jumlah_315_climbing

            # Jika skor sudah mencapai atau melebihi threshold,
            hentikan iterasi
            if (skor_kubus_terbaik >= threshold_akurasi):
                break

        # Update array histori
        histori['skor_kubus_terbaik'] = skor_kubus_terbaik
        histori['jumlah_315_terbaik'] = jumlah_315_terbaik
        histori['kubus_terbaik'] = kubus_terbaik
        histori['restart'] = restart

```

```

# Menghitung hasil timing
end_time = time.perf_counter()
elapsed_time = end_time - start_time
histori['elapsed_time'] = round(elapsed_time, 3)

return histori

```

- Inisialisasi Variabel

Tahap awal dalam pengembangan algoritma ini adalah menginisialisasi variabel yang akan digunakan. Beberapa variabel yang digunakan adalah seperti kubus\_terbaik yang digunakan untuk menyimpan *state neighbor* terbaik yang dihasilkan oleh algoritma, skor\_kubus\_terbaik yang berguna untuk menyimpan skor *neighbor* terbaik hasil algoritma, variabel iterasi yang digunakan untuk menghitung jumlah iterasi yang sudah dijalankan, dan iterasi\_stagnan untuk mencatat jumlah iterasi dimana tidak ada perbaikan skor namun nilai skornya tetap sama. Terdapat juga variabel array histori yang digunakan untuk menyimpan riwayat skor dan kondisi iterasi.

- Pembaharuan Kubus

Setelah mendapatkan *neighbor* terbaik, algoritma mengevaluasi jika *neighbor* memiliki skor yang lebih baik atau sama dengan kubus *current state*. Jika ada perbaikan skor, algoritma memperbarui kubus terbaik dan skor terbaik, menyimpan informasi iterasi, dan mengulang nilai variabel iterasi\_stagnan menjadi 0. Jika skor terbaik mencapai 100, iterasi dihentikan karena sudah mencapai solusi paling optimal (*global optima*).

- Sideways Move

Jika akurasi kubus *neighbor* setara dengan akurasi kubus *current state*, maka algoritma akan tetap melanjutkan iterasi. Jumlah iterasi\_stagnan akan bertambah dan jika sudah mencapai batas iterasi maksimum, iterasi akan dihentikan. Jika akurasi dari kubus *neighbor* lebih buruk dari kubus *current state* atau kubus yang disimpan di variabel kubus\_terbaik, algoritma akan langsung menghentikan iterasi.

- Perhitungan Waktu Eksekusi

Setelah iterasi selesai atau terhenti, algoritma akan menghitung waktu eksekusi menggunakan `time.perf_counter()`.

```
def random_restart_hill_climbing(n, maks_restart, threshold_akurasi, kubus):  
    # Memulai perhitungan waktu  
    start_time = time.perf_counter()  
  
    # Menginisialisasi array history untuk menyimpan informasi  
    histori = {  
        'skor_kubus_terbaik': None,  
        'jumlah_315_terbaik': None,  
        'kubus_terbaik': kubus,  
        'restart': None,  
        'iterasi': [],  
        'skor_kubus': [],  
        'jumlah_315': [],  
        'elapsed_time': None  
    }  
  
    # Menginisialisasi kubus_terbaik dengan kubus yang menjadi input  
    fungsi  
    kubus_terbaik = np.copy(kubus)  
  
    # Menginisialisasi skor_kubus_terbaik dengan skor kubus yang  
    menjadi input fungsi  
    skor_kubus_terbaik, jumlah_315_terbaik = cek_spesifikasi(n,  
    kubus_terbaik)  
  
    # Menghitung jumlah iterasi  
    restart = 0  
  
    while (restart < maks_restart):  
        # Membuat konfigurasi kubus baru secara acak untuk restart  
        kubus_random = inisialisasi_kubus(n)  
  
        # Melakukan hill climbing dari konfigurasi acak ini sampai
```

```

mencapai local optimum
    histori_kubus_climbing = steepest_ascent_hill_climbing(n,
kubus_random)
    restart += 1
    iterasi = histori_kubus_climbing['iterasi']
    skor_kubus_climbing =
histori_kubus_climbing['skor_kubus_terbaik']
    jumlah_315_climbing =
histori_kubus_climbing['jumlah_315_terbaik']
    kubus_climbing = histori_kubus_climbing['kubus_terbaik']

    # Menyimpan informasi kubus baru
    histori['iterasi'].append(iterasi)
    histori['skor_kubus'].append(skor_kubus_climbing)
    histori['jumlah_315'].append(jumlah_315_climbing)

    # Jika skor baru lebih baik, update kubus terbaik
    if (skor_kubus_climbing > skor_kubus_terbaik):
        kubus_terbaik = np.copy(kubus_climbing)
        skor_kubus_terbaik = skor_kubus_climbing
        jumlah_315_terbaik = jumlah_315_climbing

    # Jika skor sudah mencapai atau melebihi threshold,
hentikan iterasi
    if (skor_kubus_terbaik >= threshold_akurasi):
        break

    # Update array histori
    histori['skor_kubus_terbaik'] = skor_kubus_terbaik
    histori['jumlah_315_terbaik'] = jumlah_315_terbaik
    histori['kubus_terbaik'] = kubus_terbaik
    histori['restart'] = restart

    # Menghitung hasil timing
    end_time = time.perf_counter()
    elapsed_time = end_time - start_time
    histori['elapsed_time'] = round(elapsed_time, 3)

return histori

```

### 2.2.3 Random Restart Hill-climbing

*Random Restart Hill-climbing* merupakan salah satu varian *hill-climbing* yang memanfaatkan konsep algoritma *Steepest Ascent Hill-climbing* secara berulang kali dengan *initial state* yang berbeda-beda menuju tetangga yang lebih baik sehingga memiliki peluang lebih besar untuk menemukan solusi *global optima* dan menghindari terjebak dalam *local optima*. Algoritma ini menggunakan fungsi `random_restart_hill_climbing` yang diimplementasikan dalam beberapa tahapan yaitu sebagai berikut.

- Inisialisasi Variabel

Pada tahap ini, inisialisasi akan dilakukan untuk membuat *array* histori yang digunakan untuk menyimpan riwayat kondisi iterasi. Inisialisasi juga akan dilakukan pada kubus terbaik sementara dengan menyalin kubus awal dan menghitung skor awal yang disimpan dalam skor terbaik sementara dengan menggunakan fungsi `cek_spesifikasi`.

- Pemanfaatan Iteratif Algoritma *Steepest Ascent Hill-climbing*

Fungsi `random_restart_hill_climbing` akan melakukan *random restart* dengan membuat kubus baru secara acak menggunakan fungsi `inisialisasi_kubus` dan memanggil fungsi `steepest_ascent_hill_climbing` untuk menghasilkan solusi terbaik. Jika solusi tersebut memiliki skor yang lebih baik daripada kondisi kubus sekarang, kubus terbaik akan diperbarui dengan solusi tersebut. Iterasi ini akan berhenti jika skor sudah mencapai atau melebihi `threshold_akurasi` atau iterasi sudah mencapai batas maksimum untuk melakukan restart yang ditentukan.

- Solusi Akhir

Algoritma ini akan mencatat waktu eksekusi dan riwayat setiap perbaikan agar hasil akhirnya mencakup informasi iterasi yang terjadi dan waktu yang diperlukan kubus. Selanjutnya, algoritma akan menghasilkan solusi terbaik setelah iterasi telah selesai.

```
def random_restart_hill_climbing(n, maks_restart, threshold_akurasi,
kubus):
```

```
# Memulai perhitungan waktu
start_time = time.perf_counter()

# Menginisialisasi array history untuk menyimpan informasi
histori = {
    'skor_kubus_terbaik': None,
    'jumlah_315_terbaik': None,
    'kubus_terbaik': kubus,
    'restart': None,
    'iterasi': [],
    'skor_kubus': [],
    'jumlah_315': [],
    'elapsed_time': None
}

# Menginisialisasi kubus_terbaik dengan kubus yang menjadi input
fungsi
kubus_terbaik = np.copy(kubus)

# Menginisialisasi skor_kubus_terbaik dengan skor kubus yang
menjadi input fungsi
skor_kubus_terbaik, jumlah_315_terbaik = cek_spesifikasi(n,
kubus_terbaik)

# Menghitung jumlah iterasi
restart = 0

while (restart < maks_restart):
    # Membuat konfigurasi kubus baru secara acak untuk restart
    kubus_random = inisialisasi_kubus(n)

    # Melakukan hill climbing dari konfigurasi acak ini sampai
    mencapai local optimum
    histori_kubus_climbing = steepest_ascent_hill_climbing(n,
    kubus_random)
    restart += 1
    iterasi = histori_kubus_climbing['iterasi']
    skor_kubus_climbing =
    histori_kubus_climbing['skor_kubus_terbaik']
```

```

jumlah_315_climbing =
histori_kubus_climbing['jumlah_315_terbaik']
kubus_climbing = histori_kubus_climbing['kubus_terbaik']

# Menyimpan informasi kubus baru
histori['iterasi'].append(iterasi)
histori['skor_kubus'].append(skor_kubus_climbing)
histori['jumlah_315'].append(jumlah_315_climbing)

# Jika skor baru lebih baik, update kubus terbaik
if (skor_kubus_climbing > skor_kubus_terbaik):
    kubus_terbaik = np.copy(kubus_climbing)
    skor_kubus_terbaik = skor_kubus_climbing
    jumlah_315_terbaik = jumlah_315_climbing

    # Jika skor sudah mencapai atau melebihi threshold,
    hentikan iterasi
    if (skor_kubus_terbaik >= threshold_akurasi):
        break

# Update array histori
histori['skor_kubus_terbaik'] = skor_kubus_terbaik
histori['jumlah_315_terbaik'] = jumlah_315_terbaik
histori['kubus_terbaik'] = kubus_terbaik
histori['restart'] = restart

# Menghitung hasil timing
end_time = time.perf_counter()
elapsed_time = end_time - start_time
histori['elapsed_time'] = round(elapsed_time, 3)

Return histori

```

#### 2.2.4 Stochastic Hill-climbing

Varian dari algoritma hill-climbing yang terakhir akan digunakan dalam *local search* adalah *Stochastic Hill-climbing*. *Stochastic Hill-climbing* memiliki kesamaan dengan *Random Restart Hill-climbing* karena memanfaatkan konsep

logika dari Steepest Ascent Hill-climbing secara berulang kali untuk mencapai nilai optimal yang diinginkan, tetapi perbedaannya adalah Stochastic Hill-climbing tidak akan menggunakan *initial state* yang diperbarui terus-menerus secara acak, melainkan algoritma ini akan memanfaatkan tetangga yang memiliki hasil solusi yang lebih baik sebagai *initial state* yang baru. Algoritma ini menggunakan fungsi `random_restart_hill_climbing` yang diimplementasikan dalam beberapa tahapan yaitu sebagai berikut.

- Inisialisasi Variabel

Inisialisasi dilakukan untuk membuat variabel yang dibutuhkan dalam algoritma yaitu pembuatan *array* untuk menyimpan riwayat kondisi kubus pada setiap iterasi. Selain itu, kubus awal juga akan disimpan sebagai kubus terbaik yang bersifat sementara dan skor awal kubus akan dihitung dengan menggunakan fungsi `cek_spesifikasi` yang akan disimpan dalam sebagai skor kubus terbaik yang bersifat sementara.

- Iterasi Pencarian Kubus

Fungsi `stochastic_hill_climbing` akan menghasilkan satu tetangga baru secara acak dengan menukar dua elemen di dalam kubus terbaik saat ini atau kubus yang sedang menjadi *initial state* dengan fungsi `swap_angka`. Skor dari kondisi baru yang dihasilkan oleh tetangga akan dievaluasi dengan fungsi `cek_spesifikasi`. Jika skor kubus baru lebih tinggi, kubus ini akan menjadi kubus terbaik yang baru dan riwayat iterasi perbaikan akan dicapai, tetapi jika nilainya sama atau bahkan lebih kecil, iterasi akan langsung berlanjut tanpa adanya kubus yang baru. Iterasi akan terus dilakukan hingga mencapai atau melebihi *threshold* yang sudah ditentukan.

- Solusi Akhir

Waktu durasi eksekusi algoritma akan dihitung sampai kubus mencapai atau melebihi *threshold* yang diinginkan yang artinya pencarian sudah selesai. Selanjutnya, algoritma ini akan menghasilkan solusi terbaik setelah iterasi telah selesai.

```
def stochastic_hill_climbing(n, maks_iterasi, threshold_akurasi, kubus):  
    # Memulai perhitungan waktu  
    start_time = time.perf_counter()  
  
    # Menginisialisasi array history untuk menyimpan informasi  
    histori = {  
        'skor_kubus_terbaik': None,  
        'jumlah_315_terbaik': None,  
        'kubus_terbaik': kubus,  
        'iterasi': None,  
        'skor_kubus': [],  
        'jumlah_315': [],  
        'elapsed_time': None  
    }  
  
    # Menginisialisasi kubus_terbaik dengan kubus yang menjadi input  
    fungsi  
    kubus_terbaik = kubus  
  
    # Menginisialisasi skor_kubus_terbaik dengan skor kubus yang  
    menjadi input fungsi  
    skor_kubus_terbaik, jumlah_315_terbaik = cek_spesifikasi(n,  
    kubus_terbaik)  
  
    # Menginisialisasi jumlah iterasi dengan nilai 0  
    iterasi = 0  
  
    while (iterasi < maks_iterasi):  
        # Membuat konfigurasi kubus baru dengan menukar angka secara  
        acak  
        kubus_baru = swap_angka(n, np.copy(kubus_terbaik))  
        skor_kubus_baru, jumlah_315_baru = cek_spesifikasi(n,  
        kubus_baru)  
        iterasi += 1  
  
        # Menyimpan informasi kubus baru  
        histori['skor_kubus'].append(skor_kubus_baru)  
        histori['jumlah_315'].append(jumlah_315_baru)
```

```

        # Jika skor baru lebih baik atau sama dengan, update kubus
        terbaik
        if skor_kubus_baru > skor_kubus_terbaik:
            skor_kubus_terbaik = skor_kubus_baru
            jumlah_315_terbaik = jumlah_315_baru
            kubus_terbaik = np.copy(kubus_baru)

        # Jika skor sudah mencapai atau melebihi threshold,
        hentikan iterasi
        if (skor_kubus_terbaik >= threshold_akurasi):
            break

    # Update array histori
    histori['skor_kubus_terbaik'] = skor_kubus_terbaik
    histori['jumlah_315_terbaik'] = jumlah_315_terbaik
    histori['kubus_terbaik'] = kubus_terbaik
    histori['iterasi'] = iterasi

    # Menghitung hasil timing
    end_time = time.perf_counter()
    elapsed_time = end_time - start_time
    histori['elapsed_time'] = round(elapsed_time, 3)

return histori

```

### 2.2.5 Simulated Annealing

*Simulated Annealing* merupakan sebuah algoritma yang mengikuti konsep dari proses pendinginan logam dalam ilmu material. Dalam konsep tersebut, material dipanaskan hingga mencapai suhu tinggi lalu didinginkan secara perlahan untuk mencapai struktur paling stabil atau energi terendah. Konsep ini dimanfaatkan dalam algoritma *simulated annealing* dengan tujuan untuk mencari solusi optimal dengan mencoba berbagai solusi, bahkan yang buruk dengan memanfaatkan suatu probabilitas dengan rumus  $e^{\frac{\Delta E}{T}}$ , dimana temperatur (T) akan

berkurang secara bertahap. *Simulated Annealing* berguna untuk menghindari perangkap dalam *local optima* karena konsep ini.

Beberapa tahap dalam pengembangan algoritma simulated annealing adalah sebagai berikut.

- Inisialisasi Variabel

Beberapa variabel yang digunakan dalam algoritma ini adalah temperatur yang menjadi parameter yang dimasukkan oleh pengguna sebagai temperatur untuk rumus probabilitas penerimaan kegagalan, kubus\_terbaik untuk menyimpan kubus *neighbor* terbaik hasil algoritma, dan skor\_kubus\_terbaik yang menyimpan akurasi kubus *neighbor* terbaik hasil algoritma. Terdapat juga variabel *array* histori yang digunakan untuk menyimpan riwayat skor dan kondisi iterasi.

- Pencarian *Neighbor*

Algoritma akan mencari *neighbor* baru secara *random* dengan menggunakan fungsi *swap\_angka* dari kubus *current state*. Kubus yang telah dimodifikasi akan dihitung akurasinya serta jumlah\_315 (menghitung jumlah elemen kubus yang memiliki jumlah setara dengan magic number kubus tersebut) dengan menggunakan *cek\_spesifikasi*. Perbedaan skor (skor\_delta) dihitung untuk menentukan aksi apa yang akan diambil kemudian oleh algoritma.

- Perhitungan Probabilitas Penerimaan *Neighbor* yang Lebih Buruk

Setelah menghitung skor\_delta, jika skor\_delta positif maka secara otomatis kubus *neighbor* akan diterima sebagai kubus terbaik saat ini. Namun, bila skor\_delta negatif maka algoritma harus mengetahui probabilitas penerimaan kubus *neighbor* tersebut dengan rumus  $e^{\frac{\Delta E}{T}}$ . Bila nilai probabilitas masih bisa diterima maka algoritma akan melanjutkan proses menggunakan kubus *neighbor* tersebut dan temperatur hasil masukan dari pengguna akan dikurangi. Namun, bila nilai probabilitas tidak dapat diterima maka algoritma akan melanjutkan prosesnya dengan kubus terakhir sebelum pencarian *neighbor* yang buruk.

- Pembaharuan Kubus Terbaik

Jika kubus baru (kubus\_temp) menghasilkan akurasi yang lebih baik dari kubus\_baru, maka variabel kubus\_baru akan diperbarui dengan kubus yang ada di kubus\_temp.

- Solusi Akhir

Setelah iterasi selesai, baik karena mencapai batas iterasi atau karena tidak ada perbaikan lebih lanjut, fungsi menghitung waktu eksekusi dan mengembalikan berbagai informasi penting seperti kubus terbaik beserta informasi penting lainnya yang disimpan dalam *array* histori. Informasi ini berguna untuk memahami efektivitas algoritma dan memvisualisasikan hasil akhir pencarian algoritma untuk menemukan solusi paling optimal.

```
def simulated_annealing(n, maks_iterasi, threshold_akurasi,
temperatur_awal, minimum_temperatur, laju_penurunan, kubus):
    # Memulai perhitungan waktu
    start_time = time.perf_counter()

    # Menginisialisasi array history untuk menyimpan informasi
    histori = {
        'skor_kubus_terbaik': None,
        'jumlah_315_terbaik': None,
        'kubus_terbaik': kubus,
        'iterasi': None,
        'iterasi_local_optima': 0,
        'skor_kubus': [],
        'jumlah_315': [],
        'probabilitas': [],
        'temperatur': [],
        'elapsed_time': None
    }

    # Menginisialisasi kubus terbaik dan kubus sekarang dengan kubus
    # yang menjadi input fungsi
    kubus_terbaik = np.copy(kubus)
    kubus_sekarang = np.copy(kubus)
```

```
# Menginisialisasi skor awal
skor_kubus_terbaik, jumlah_315_terbaik = cek_spesifikasi(n,
kubus_terbaik)
skor_sekarang = skor_kubus_terbaik
jumlah_315_sekarang = jumlah_315_terbaik

# Menginisialisasi parameter temperatur dan iterasi
temperatur = temperatur_awal
iterasi = 0
rejected_streak = 0

while (iterasi < maks_iterasi) and (temperatur >
minimum_temperatur):
    # Membuat kubus_temp yang dihasilkan dengan fungsi swap_angka
    # terhadap kubus_sekarang
    kubus_temp = swap_angka(n, np.copy(kubus_sekarang))
    skor_kubus_temp, jumlah_315_temp = cek_spesifikasi(n,
kubus_temp)
    iterasi += 1

    # Menghitung perbedaan skor dan probabilitas penerimaan
    skor_delta = skor_kubus_temp - skor_sekarang
    if skor_delta > 0:
        probabilitas = 1.0
    else:
        # Menghitung probabilitas penerimaan solusi yang lebih
        # buruk
        probabilitas = np.exp(skor_delta / temperatur)

    # Menyimpan informasi untuk history
    histori['skor_kubus'].append(skor_sekarang)
    histori['jumlah_315'].append(jumlah_315_sekarang)
    histori['probabilitas'].append(probabilitas)
    histori['temperatur'].append(temperatur)

    # Menentukan apakah menerima solusi baru
    if random.random() < probabilitas:
        # Menerima solusi baru
        kubus_sekarang = np.copy(kubus_temp)
```

```

skor_sekarang = skor_kubus_temp
jumlah_315_sekarang = jumlah_315_temp
rejected_streak = 0

# Update solusi terbaik jika ditemukan yang lebih baik
if skor_sekarang > skor_kubus_terbaik:
    kubus_terbaik = np.copy(kubus_sekarang)
    skor_kubus_terbaik = skor_sekarang
    jumlah_315_terbaik = jumlah_315_sekarang

# Berhenti jika sudah mencapai threshold akurasi
if skor_kubus_terbaik >= threshold_akurasi:
    break

else:
    # Solusi ditolak, tambah penghitung penolakan
    berturut-turut
    rejected_streak += 1

    # Memeriksa apakah terjebak di local optima
    if rejected_streak >= 10: # Threshold untuk mendeteksi local
optima
        histori['iterasi_local_optima'] += 1
        rejected_streak = 0
        # Menaikkan temperatur sementara untuk keluar dari local
optima
        temperatur = min(temperatur_awal, temperatur * 2)
    else:
        # Menurunkan temperatur sesuai jadwal pendinginan
        if temperatur > minimum_temperatur:
            temperatur = max(minimum_temperatur, temperatur *
laju_penurunan)

    # Update array histori dengan hasil akhir
histori['skor_kubus_terbaik'] = skor_kubus_terbaik
histori['jumlah_315_terbaik'] = jumlah_315_terbaik
histori['kubus_terbaik'] = kubus_terbaik
histori['iterasi'] = iterasi

# Menghitung waktu eksekusi

```

```
end_time = time.perf_counter()
histori['elapsed_time'] = round(end_time - start_time, 3)

return histori
```

#### 2.1.6 Genetic Algorithm

Pengaplikasian algoritma genetika dibagi ke dalam tiga tahap, yaitu tahap seleksi, tahap reproduksi, dan tahap mutasi. Namun, sebelum memasuki tahap-tahap yang sudah disebutkan diperlukan adanya persiapan.

Beberapa tahap persiapan yang dilakukan dalam pengaplikasian algoritma genetika adalah sebagai berikut.

- Pembuatan Fungsi *Crossover* untuk Proses Reproduksi

Tujuan pembuatan fungsi ini adalah sebagai alat pertukaran gen yang akan dilakukan di proses reproduksi. Pertukaran gen antara *parent 1* dan *parent 2* akan menghasilkan *child 1* dan *child 2*. Tahap awal dari algoritma ini adalah dengan mengubah struktur kubus *parent* yang awalnya berbentuk tiga dimensi menjadi satu dimensi. Hal ini dilakukan agar tidak terbentuk angka yang sama di dalam kubus hasil pertukaran gen. Setelah kedua *parent* diubah bentuknya dari tiga dimensi menjadi satu dimensi, akan ditentukan *crossover point (c point)* yang akan menentukan dari mana gen dari setiap *parent* akan diturunkan ke anak. Pembentukan gen di kubus *child 1* adalah dengan mengambil bagian pertama dari *parent 1* hingga *c point* serta mengambil bagian dari *parent 2*, mulai dari *c point* hingga akhir. Pembentukan *child 2* juga dilakukan seperti itu namun sebaliknya. Gen yang merupakan angka yang dimasukkan ke dalam kubus *child* harus dipastikan belum ada di kubus *child*. Jadi, angka yang dimasukkan hanyalah angka yang belum ada di kubus *child* untuk memastikan tidak adanya angka yang sama di dalam kubus. Dikarenakan *parent* yang digunakan untuk membentuk kubus *child* masih berbentuk satu dimensi maka kubus *child* juga berbentuk satu dimensi, oleh karena itu struktur dari kubus *child* harus diatur ulang untuk menjadi tiga dimensi. Setelah

menjadi kubus tiga dimensi, kedua kubus *child* akan diukur akurasinya dengan menggunakan fungsi *cek\_spesifikasi*.

```
def crossover(parent1, parent2, n, c_point = None):

    if c_point is not None:
        if not (1 <= c_point <= n**3 -1):
            print ("c_point harus antara 1 dan " + str(n**3 -1))
            return
    else:
        c_point = random.randint(1, n**3 -1)

    parent1_1d = []
    parent2_1d = []

    for i in range(n):
        for j in range(n):
            for k in range(n):
                parent1_1d.append(parent1[i][j][k])
                parent2_1d.append(parent2[i][j][k])

    # Ambil bagian pertama dari parent1 untuk child1
    child1_part1 = parent1_1d[:c_point]

    # Ambil bagian kedua dari parent2, tapi hanya angka yang belum
    ada di child1_part1
    child1_part2 = []
    used_numbers = set(child1_part1)
    for num in parent2_1d[c_point:]:
        if num not in used_numbers:
            child1_part2.append(num)
            used_numbers.add(num)

    # Jika masih ada angka yang kurang, tambahkan dari angka yang
    belum digunakan
    all_numbers = set(range(1, n**3 + 1))
    remaining_numbers = list(all_numbers - used_numbers)
    child1_part2.extend(remaining_numbers)
```

```

# Ulangi proses yang sama untuk child2
child2_part1 = parent2_1d[:c_point]
child2_part2 = []
used_numbers = set(child2_part1)
for num in parent1_1d[c_point:]:
    if num not in used_numbers:
        child2_part2.append(num)
        used_numbers.add(num)

# Tambahkan angka yang kurang untuk child2
all_numbers = set(range(1, n**3 + 1))
remaining_numbers = list(all_numbers - used_numbers)
child2_part2.extend(remaining_numbers)

# Gabungkan bagian-bagian
child1_1d = child1_part1 + child1_part2
child2_1d = child2_part1 + child2_part2

# Verifikasi bahwa setiap child memiliki angka unik
assert len(set(child1_1d)) == n**3, "Child1 memiliki angka yang berulang"
assert len(set(child2_1d)) == n**3, "Child2 memiliki angka yang berulang"
assert all(1 <= x <= n**3 for x in child1_1d), "Child1 memiliki angka di luar range"
assert all(1 <= x <= n**3 for x in child2_1d), "Child2 memiliki angka di luar range"

# Copy parents dan isi dengan nilai baru
child1 = np.array(child1_1d).reshape((n, n, n))
child2 = np.array(child2_1d).reshape((n, n, n))

child1_akurasi, child1_jumlah_315 = cek_spesifikasi(n, child1)
child2_akurasi, child2_jumlah_315 = cek_spesifikasi(n, child2)

return (
    child1, {'akurasi': child1_akurasi, 'jumlah_315':
child1_jumlah_315},
    child2, {'akurasi': child2_akurasi, 'jumlah_315':

```

```
child2_jumlah_315}  
)
```

- Pembuatan Fungsi Inisialisasi Populasi

Fungsi ini berfungsi untuk membuat populasi awal dalam algoritma genetika, dimana setiap individu adalah setiap kubus yang telah diinisialisasi dan diberi nilai akurasi. Parameter yang ada di dalam fungsi ini adalah `population_size` yang menentukan jumlah individu dalam suatu populasi. Tahap awal dalam fungsi ini adalah membuat *list* kosong sebagai tempat untuk menambahkan individu-individu di dalam suatu populasi. Individu dibuat dengan menggunakan fungsi `inisialisasi_kubus` yang diiterasi sebanyak `population_size`. Setiap individu yang dibuat lewat iterasi akan diukur akurasinya dengan fungsi `cek_spesifikasi`. Individu dan akurasinya akan disimpan dalam bentuk *dictionary* dengan dua entri yaitu kubus dan akurasi, yang kemudian ditambahkan ke dalam *list population*.

```
def inisialisasi_populasi(population_size):  
    population = []  
    for _ in range (population_size):  
        kubus_baru = inisialisasi_kubus(n)  
        akurasi, jumlah_315 = cek_spesifikasi(n, kubus_baru)  
        population.append({  
            'kubus': kubus_baru,  
            'akurasi': akurasi,  
            'jumlah_315': jumlah_315  
        })  
    return population
```

Setelah itu, *genetic algorithm* siap dikembangkan dalam 3 tahapan berikut.

- Seleksi

Dalam proses ini, individu yang berada di dalam populasi akan diurutkan sesuai dengan akurasi yang dimiliki, semakin tinggi akurasi maka akan semakin baik juga peringkatnya. Setelah tahap pemeringkatan, setiap individu akan diberikan probabilitas seleksi sesuai dengan peringkatnya masing-masing, semakin tinggi peringkatnya maka akan semakin rendah

kemungkinan terpilih menjadi sebuah *parent*, hal ini dilakukan agar memungkinkan variasi yang lebih besar dalam eksplorasi solusi. Dua individu/kubus terpilih sebagai *parent* dengan memastikan kedua *parent* tersebut merupakan dari dua individu yang berbeda agar mempertahankan keragaman genetika.

- Reproduksi

Dalam proses ini, diciptakan dua kubus *child* hasil reproduksi dua kubus *parent*. Dengan menggunakan fungsi *crossover* yang sudah dijelaskan di atas, *parent1* dan *parent2* akan menghasilkan kubus *child* yang diharapkan dapat meningkatkan akurasi keseluruhan dengan menggabungkan “gen” (angka elemen kubus *parent*).

- Mutasi

Setiap kubus *child* memiliki kemungkinan untuk mengalami perubahan pada gen sesuai dengan probabilitas mutasi yang ditentukan sebagai parameter. Mutasi ini dilakukan secara acak untuk menjaga keragaman genetik dalam populasi dan mencegah algoritma terjebak pada solusi yang kurang optimal (*local optima*). Mutasi ini dapat terjadi dengan menggunakan fungsi *swap\_angka*, akurasi kubus *child* akan dihitung kembali untuk memastikan keefektifan perubahan tersebut.

```
def genetic_algorithm(population_size, kubus, generations,
mutation_probability):
    # Memulai perhitungan waktu
    start_time = time.perf_counter()

    # Menginisialisasi array history untuk menyimpan informasi
    histori = {
        'skor_kubus_terbaik': None,
        'jumlah_315_terbaik': None,
        'kubus_terbaik': kubus,
        'evaluasi_per_generasi': [],
        'elapsed_time': None
    }
```

```

# Menginisialisasi populasi
population = inisialisasi_populasi(population_size)

for kembang_biak in range(generations):
    # Ranking populasi berdasarkan skor
    population = sorted(population, key=lambda x:
(x['jumlah_315'], x['akurasi']))
    rank_population = []

    for rank, individual in enumerate(population, 1):
        rank_population.append({
            'kubus': individual['kubus'],
            'akurasi': individual['akurasi'],
            'rank': rank
        })

    # Menghitung probabilitas dari rank
    total_fitness = 0
    for individual in rank_population:
        total_fitness += individual["rank"]

    probabilitas_pemilihan = []
    for individual in rank_population:
        probabilitas_pemilihan.append(individual["rank"] /
total_fitness)

    new_population = []
    for _ in range(0, population_size, 2):
        parent1 = np.random.choice(rank_population,
p=probabilitas_pemilihan)
        parent2 = parent1
        while parent2['kubus'] is parent1['kubus']:
            parent2 = np.random.choice(rank_population,
p=probabilitas_pemilihan)

        # Crossover populasi
        child1_kubus, child1_stats, child2_kubus, child2_stats =
crossover(parent1['kubus'], parent2['kubus'], 5)

```

```

# Proses mutasi untuk child1
if random.random() < mutation_probability:
    child1_kubus = swap_angka(n, child1_kubus)
    child1_akurasi, child1_jumlah_315 = cek_spesifikasi(n,
child1_kubus)
    child1_stats = {'akurasi': child1_akurasi,
'jumlah_315': child1_jumlah_315}

# Proses mutasi untuk child2
if random.random() < mutation_probability:
    child2_kubus = swap_angka(n, child2_kubus)
    child2_akurasi, child2_jumlah_315 = cek_spesifikasi(n,
child2_kubus)
    child2_stats = {'akurasi': child2_akurasi,
'jumlah_315': child2_jumlah_315}

new_population.extend([
    {'kubus': child1_kubus, 'akurasi':
child1_stats['akurasi'], 'jumlah_315' : child1_stats['jumlah_315']},
    {'kubus': child2_kubus, 'akurasi':
child2_stats['akurasi'], 'jumlah_315' : child2_stats['jumlah_315']}
])

# Update populasi dengan populasi baru
population = new_population

avg_jumlah_per_generasi = sum(individu['jumlah_315'] for
individu in population) / len(population)
avg_akurasi_per_generasi = sum(individu['akurasi'] for
individu in population) / len(population)
maks_jumlah_per_generasi = max(individu['jumlah_315'] for
individu in population)
maks_akurasi_per_generasi = max(individu['akurasi'] for
individu in population)
kubus_terbaik_per_generasi = max(population, key=lambda x:
(x['jumlah_315'], x['akurasi']))['kubus']

histori['evaluasi_per_generasi'].append({
    'avg_jumlah_generasi': round(avg_jumlah_per_generasi, 3),
    'avg_akurasi_generasi': round(avg_akurasi_per_generasi, 3),
    'maks_jumlah_generasi': maks_jumlah_per_generasi,
    'maks_akurasi_generasi': maks_akurasi_per_generasi,
    'kubus_terbaik_generasi': kubus_terbaik_per_generasi})

```

```
        'avg_skor_generasi': round(avg_akurasi_per_generasi, 3),
        'jumlah_terbaik_generasi': maks_jumlah_per_generasi,
        'skor_terbaik_generasi': maks_akurasi_per_generasi,
        'kubus_terbaik_generasi': kubus_terbaik_per_generasi
    })

# Cari kubus terbaik berdasarkan total_selisih
best_kubus_entry = max(population, key=lambda x:
(x['jumlah_315'], x['akurasi']))
best_kubus = best_kubus_entry[ 'kubus']

skor_kubus_terbaik, jumlah_315_terbaik = cek_spesifikasi(n,
best_kubus)
histori[ 'skor_kubus_terbaik'] = skor_kubus_terbaik
histori[ 'jumlah_315_terbaik'] = jumlah_315_terbaik
histori[ 'kubus_terbaik'] = best_kubus

# Menghitung hasil timing
end_time = time.perf_counter()
elapsed_time = end_time - start_time
histori[ 'elapsed_time'] = round(elapsed_time, 3)

return histori
```

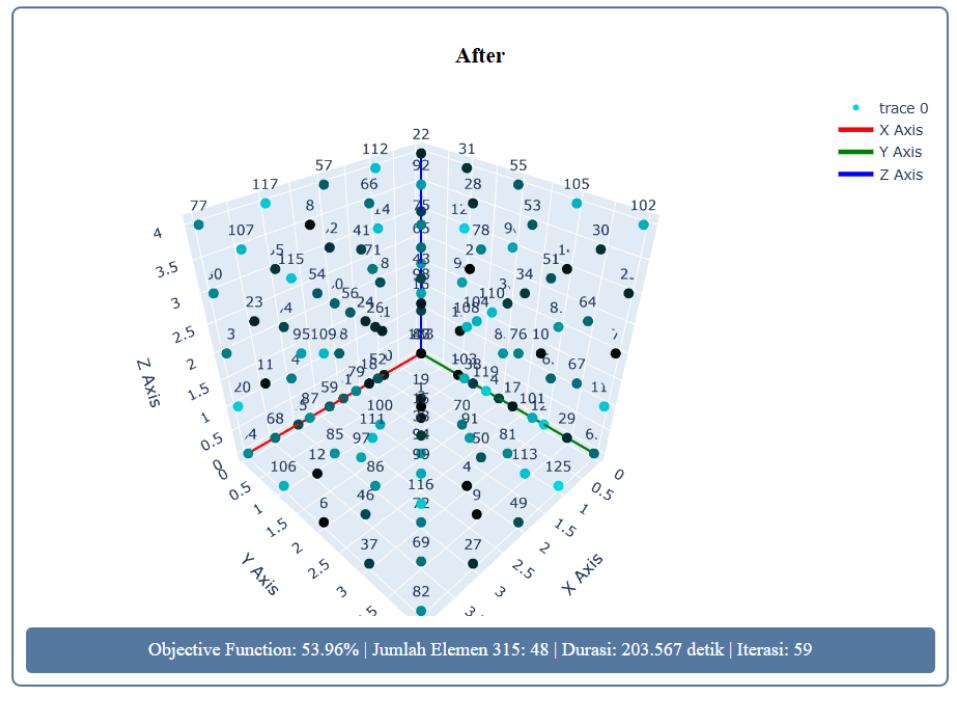
## 2.3 Hasil Eksperimen dan Analisis

### 2.2.1 Steepest Ascent Hill-climbing

Berikut adalah hasil eksperimen untuk *local search* dengan algoritma *Steepest Ascent Hill-climbing*.

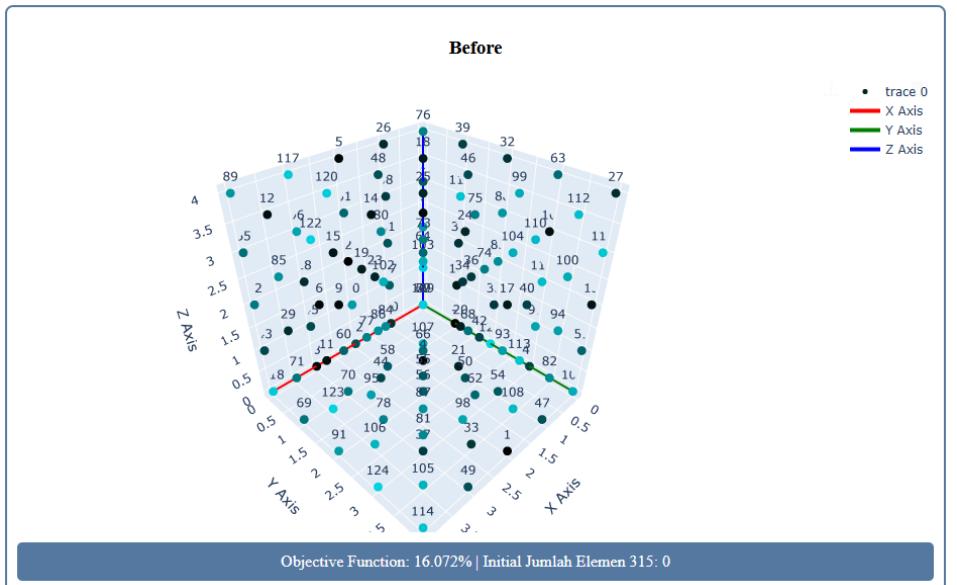
Eksperimen 1	
State awal kubus	<p>Before</p> <ul style="list-style-type: none"> <li>trace 0</li> <li>X Axis</li> <li>Y Axis</li> <li>Z Axis</li> </ul> <p>Objective Function: 16.874%   Initial Jumlah Elemen 315: 1</p>
Nilai <i>objective function</i>	53.96%
Plot <i>objective function</i>	<p>Progress Over Iterations</p> <p>Success Percentage</p> <p>Iteration</p>
Durasi proses pencarian	203.567 detik
Banyak iterasi	59

State akhir kubus

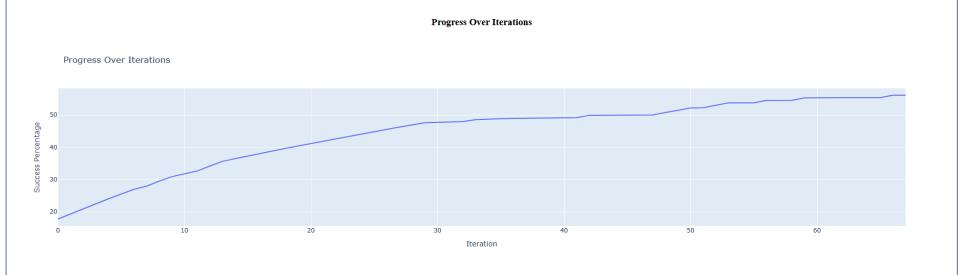
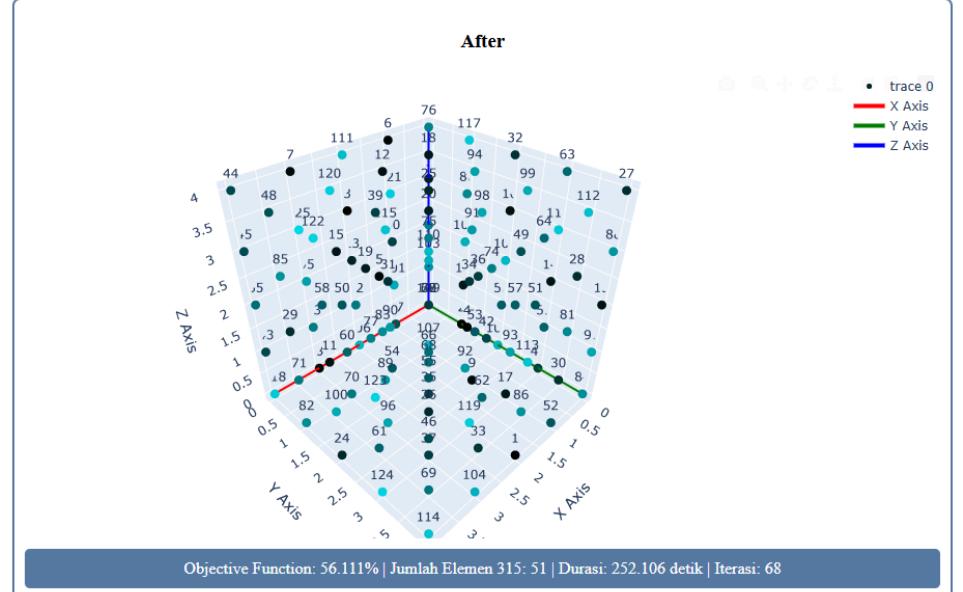


### Eksperimen 2

State awal kubus

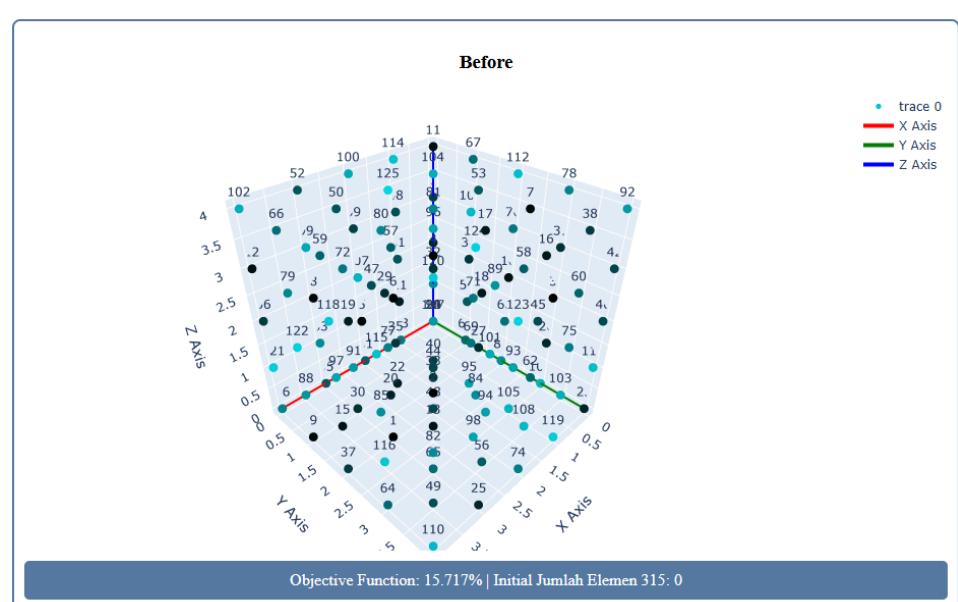


Nilai <i>objective function</i>	56.111%
---------------------------------	---------

<i>Plot objective function</i>	
Durasi proses pencarian	252.106 detik
Banyak iterasi	68
State akhir kubus	

### Eksperimen 3

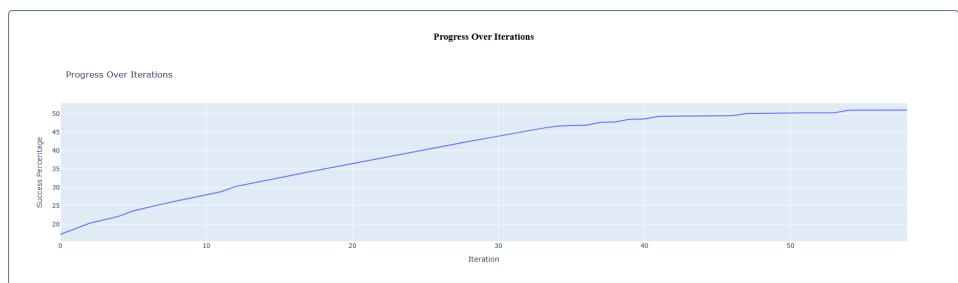
State awal kubus



Nilai *objective function*

51.037%

Plot *objective function*



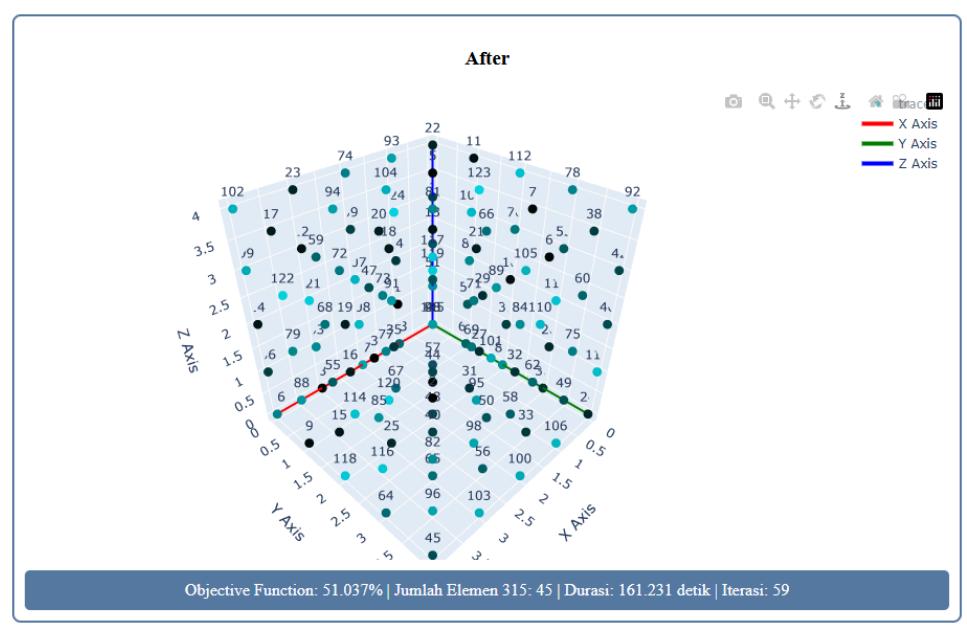
Durasi proses pencarian

161.231 detik

Banyak iterasi

59

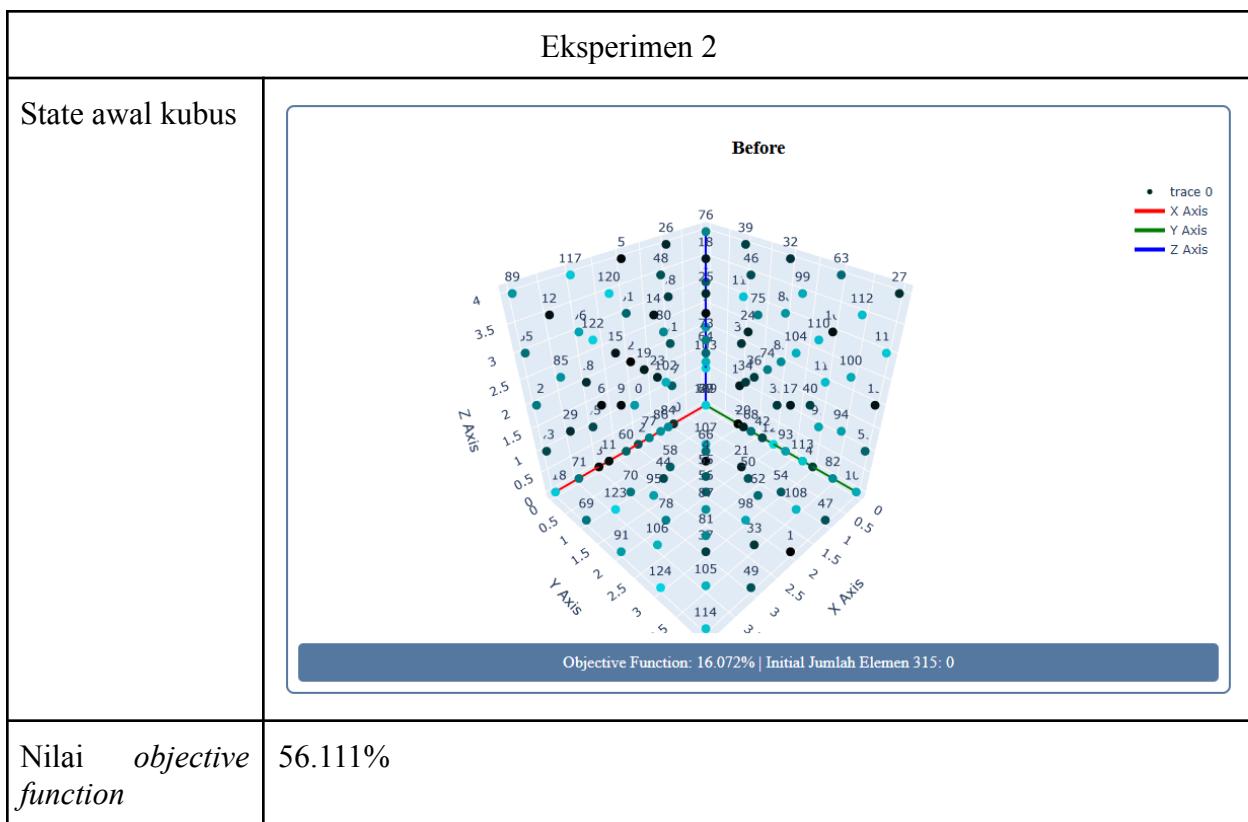
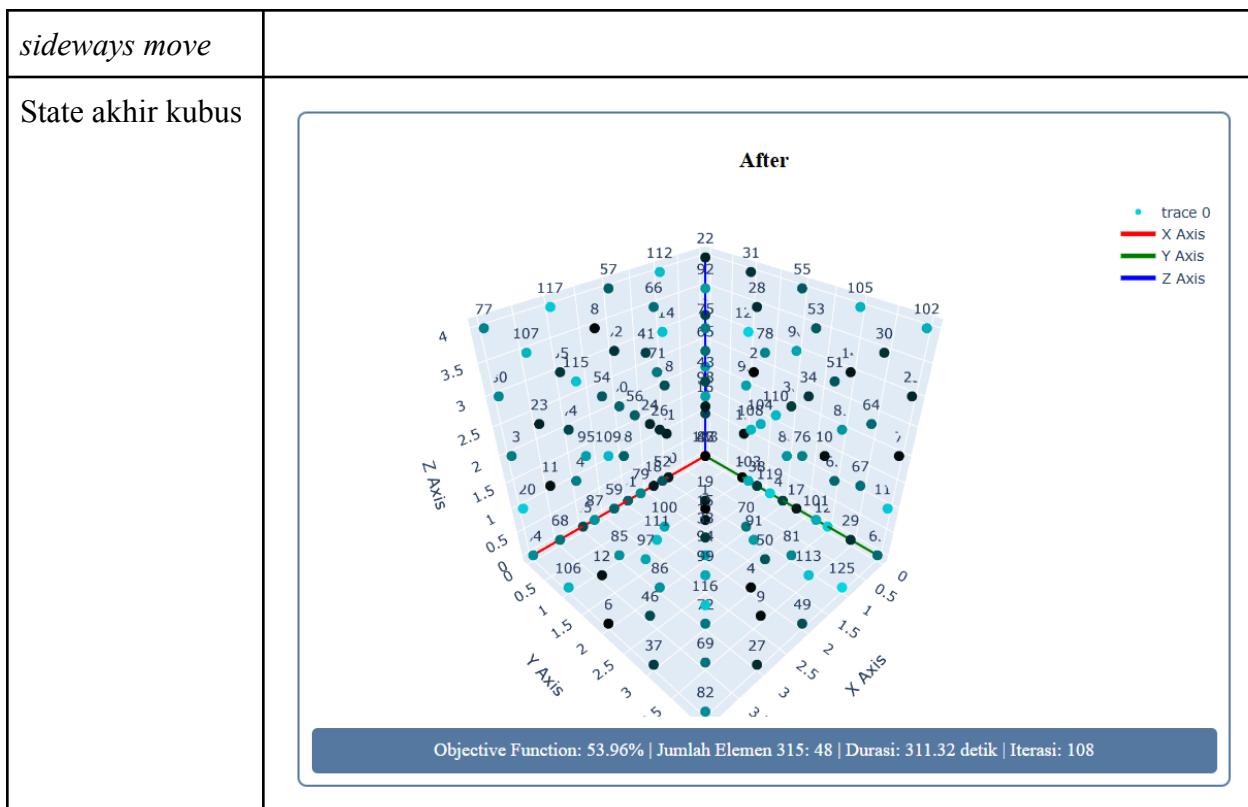
### State akhir kubus

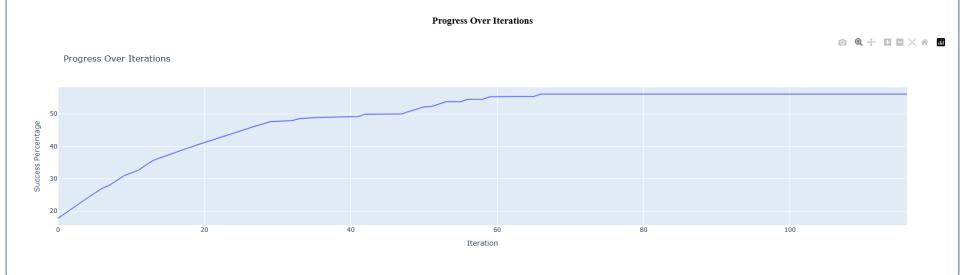
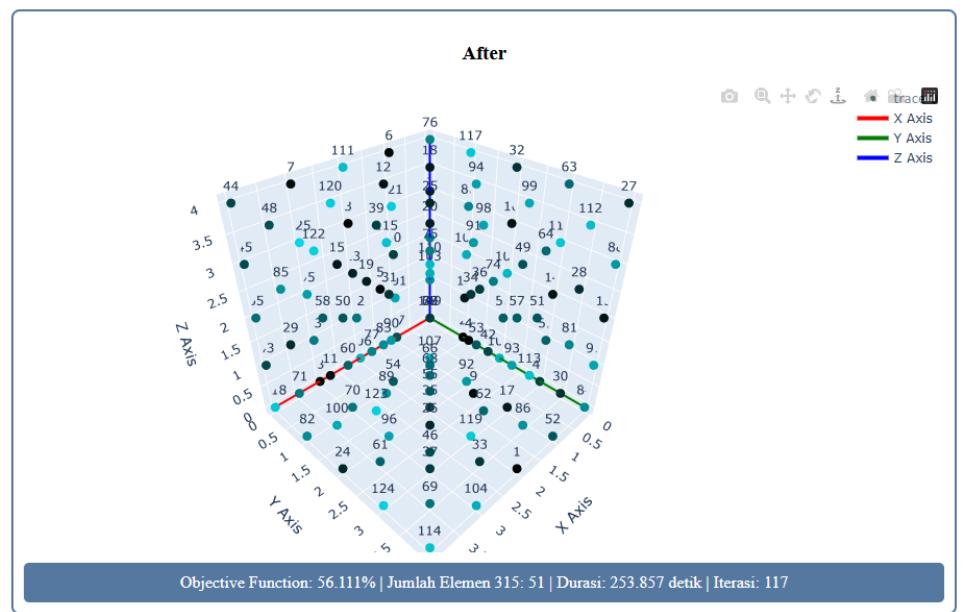


### 2.2.2 Hill-climbing with Sideways Move

Berikut adalah hasil eksperimen untuk *local search* dengan algoritma *Hill-climbing with Sideways Move*.

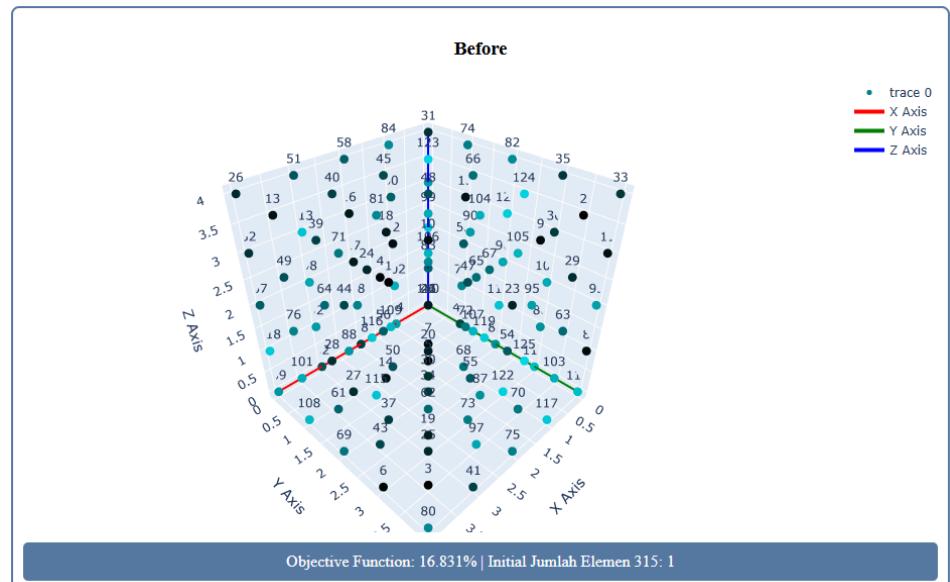
Eksperimen 1	
State awal kubus	
Nilai <i>objective function</i>	53.96%
Plot <i>objective function</i>	
Durasi proses pencarian	311.32 detik
Banyak iterasi	108
Maksimum	50



<i>Plot objective function</i>	
Durasi proses pencarian	253.857 detik
Banyak iterasi	117
Maksimum <i>sideways move</i>	50
State akhir kubus	 <p>After</p> <p>X Axis Y Axis Z Axis</p> <p>Objective Function: 56.111%   Jumlah Elemen 315: 51   Durasi: 253.857 detik   Iterasi: 117</p>

### Eksperimen 3

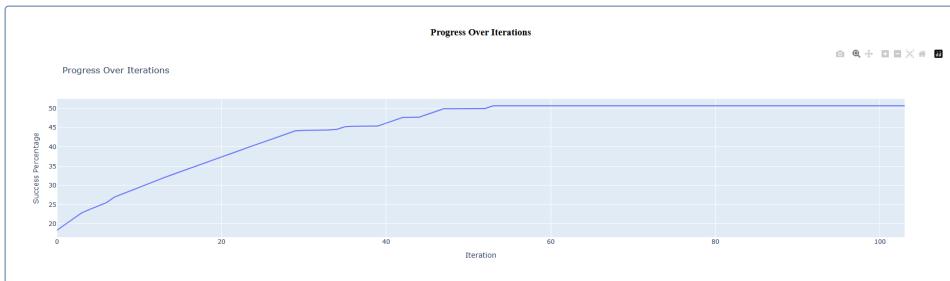
State awal kubus



Nilai *objective function*

50.685%

Plot *objective function*



Durasi proses pencarian

284.246 detik

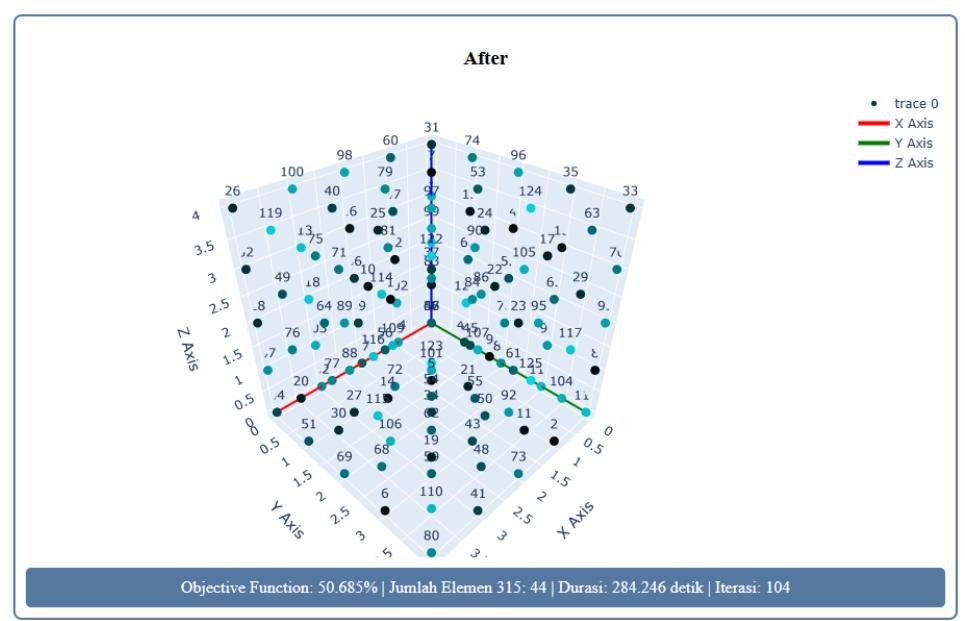
Banyak iterasi

104

Maksimum *sideways move*

50

State akhir kubus

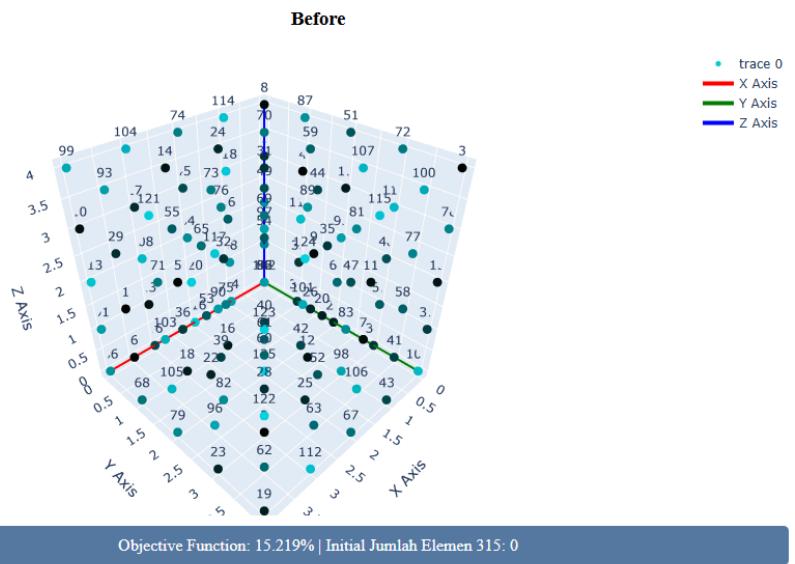


### 2.2.3 Random Restart Hill-climbing

Berikut adalah hasil eksperimen untuk *local search* dengan algoritma *Random Restart Hill-climbing*.

Eksperimen 1

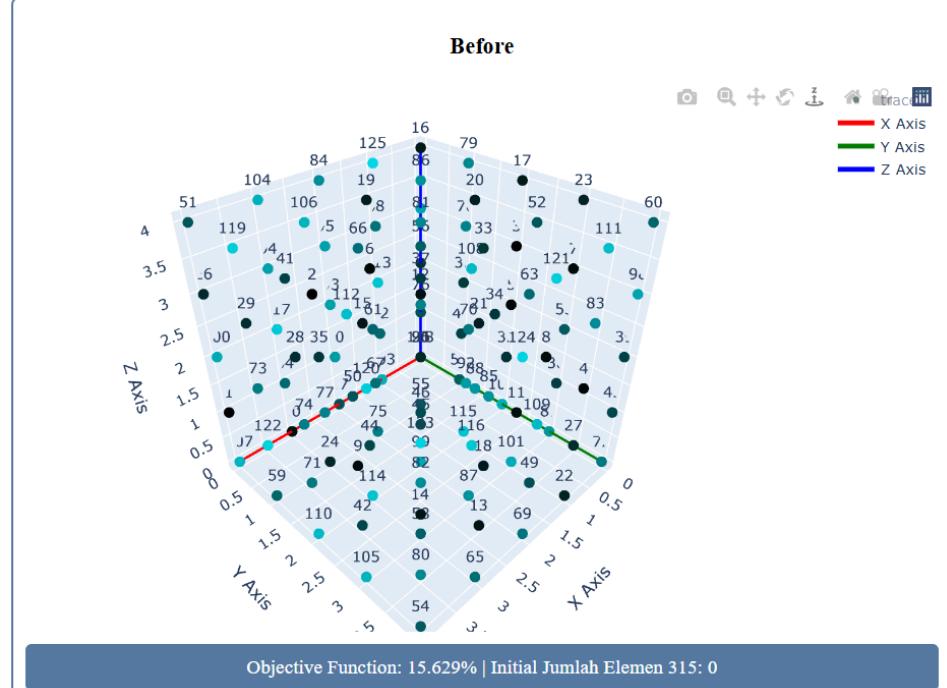
State awal kubus



Nilai <i>objective function</i>	52.705%
Plot <i>objective function</i>	
Durasi proses pencarian	718.266 detik
Banyak <i>restart</i>	3
Banyak iterasi per <i>restart</i>	<i>Restart 1:</i> 63 iterasi <i>Restart 2:</i> 61 iterasi <i>Restart 3:</i> 66 iterasi
Maksimum <i>restart</i>	3
State akhir kubus	<p>Percentase Sukses: 52.705%      Jumlah Elemen 315: 47      Durasi: 718.266 detik      Total Iterasi: 190      Restart 1: 63 iterasi, Restart 2: 61 iterasi, Restart 3: 66 iterasi</p>

## Eksperimen 2

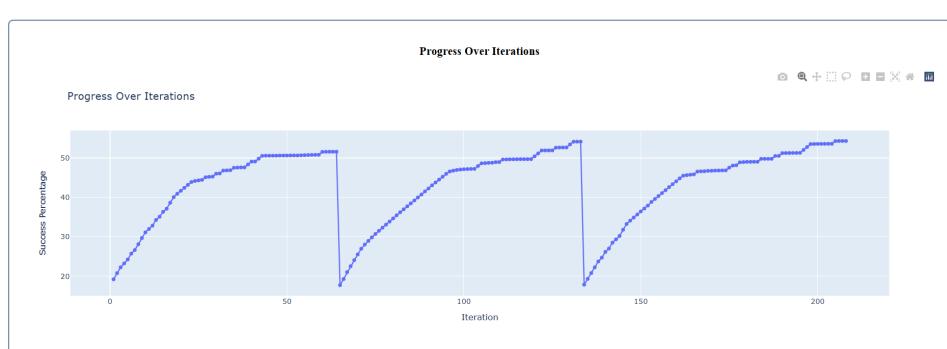
State awal kubus



Nilai *objective function*

54.316%

Plot *objective function*



Durasi proses pencarian

532.77 detik

Banyak *restart*

3

Banyak iterasi per *restart*

*Restart 1:* 64 iterasi  
*Restart 2:* 69 iterasi  
*Restart 3:* 75 iterasi

Maksimum

3

restart

State akhir kubus

**After**

X Axis  
Y Axis  
Z Axis

Z Axis

Y Axis

X Axis

Percentase Sukses: 54.316%

Jumlah Elemen 315: 49

Durasi: 532.77 detik

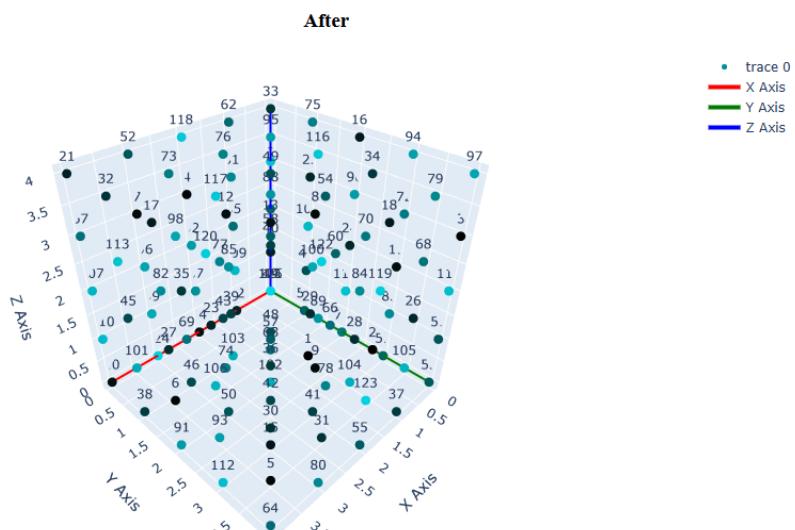
Total Iterasi: 208

Restart 1: 64 iterasi, Restart 2: 69 iterasi, Restart 3: 75 iterasi

### Eksperimen 3

State awal kubus	<p>Objective Function: 15.219%   Initial Jumlah Elemen 315: 0</p>
Nilai <i>objective function</i>	54.286%
Plot <i>objective function</i>	
Durasi proses pencarian	682.264 detik
Banyak <i>restart</i>	3
Banyak iterasi per <i>restart</i>	<p><i>Restart 1:</i> 66 iterasi  <i>Restart 2:</i> 59 iterasi  <i>Restart 3:</i> 51 iterasi</p>
Maksimum <i>restart</i>	3

### State akhir kubus



Persentase Sukses: 54.286%

Jumlah Elemen 315: 49

Durasi: 682.264 detik

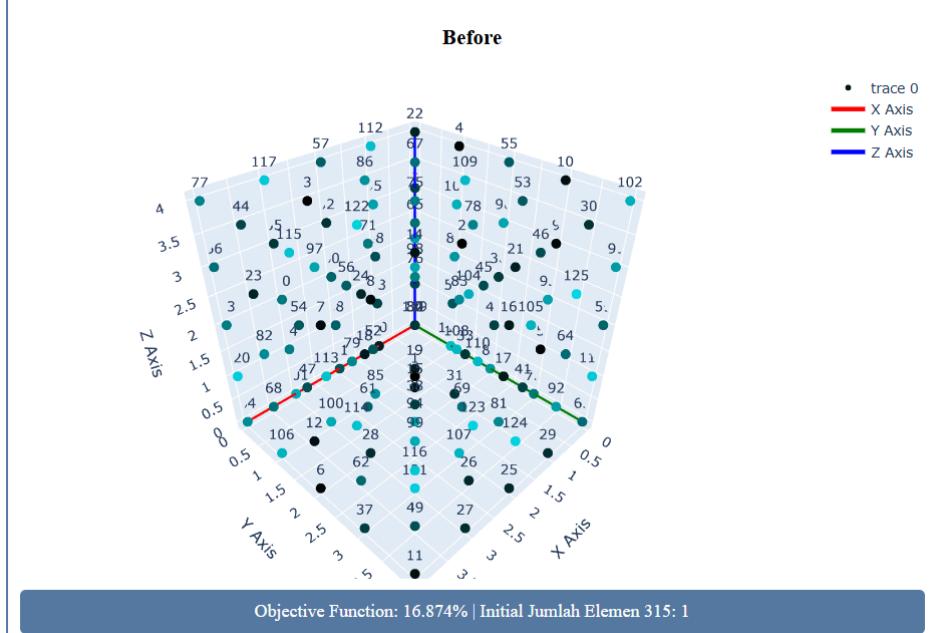
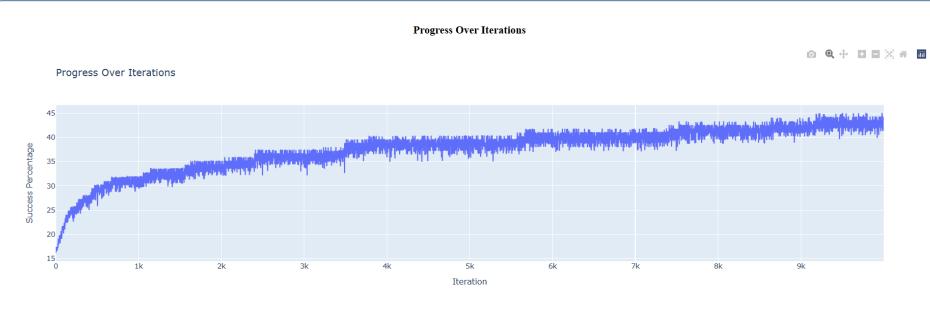
Total Iterasi: 176

Restart 1: 66 iterasi, Restart 2: 59 iterasi, Restart 3: 51 iterasi

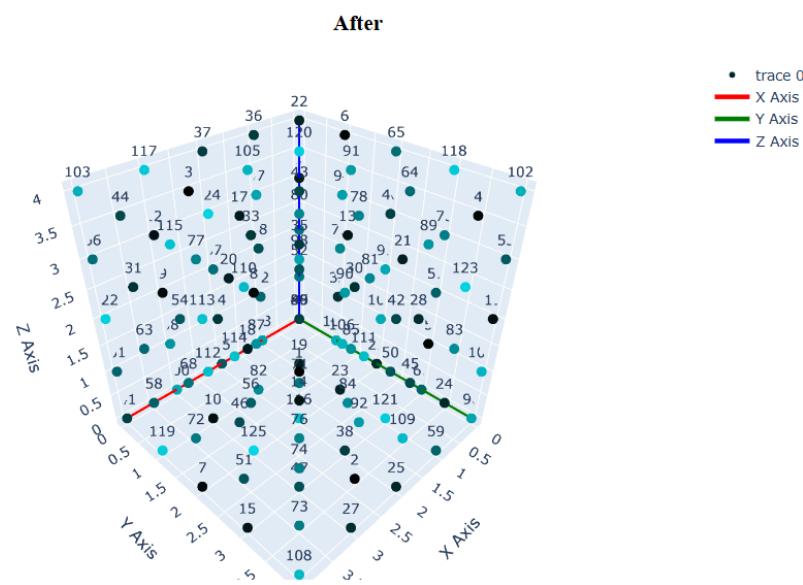


#### 2.2.4 Stochastic Hill-climbing

Berikut adalah hasil eksperimen untuk *local search* dengan algoritma *Stochastic Hill-climbing*.

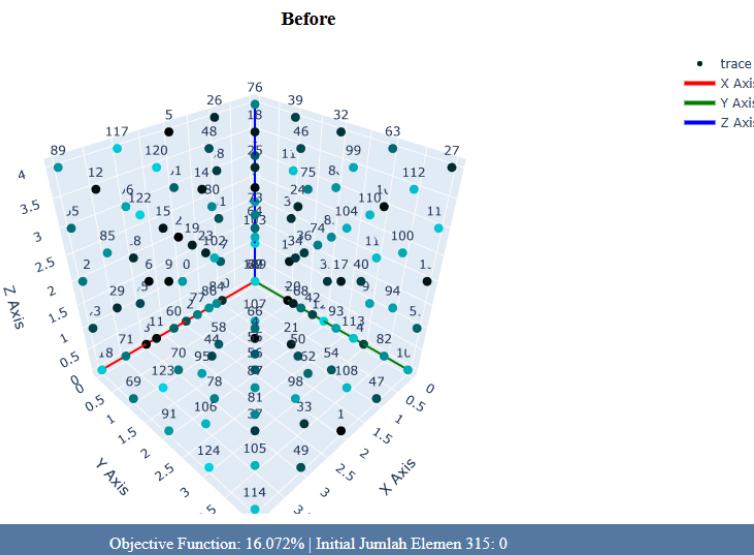
Eksperimen 1	
State awal kubus	 <p>Before</p> <ul style="list-style-type: none"> <li>trace 0</li> <li>X Axis</li> <li>Y Axis</li> <li>Z Axis</li> </ul> <p>Objective Function: 16.874%   Initial Jumlah Elemen 315: 1</p>
Nilai <i>objective function</i>	45.056%
Plot <i>objective function</i>	 <p>Progress Over Iterations</p> <p>Success Percentage</p> <p>Iteration</p>
Durasi proses pencarian	5.087 detik
Banyak iterasi	10000

State akhir kubus

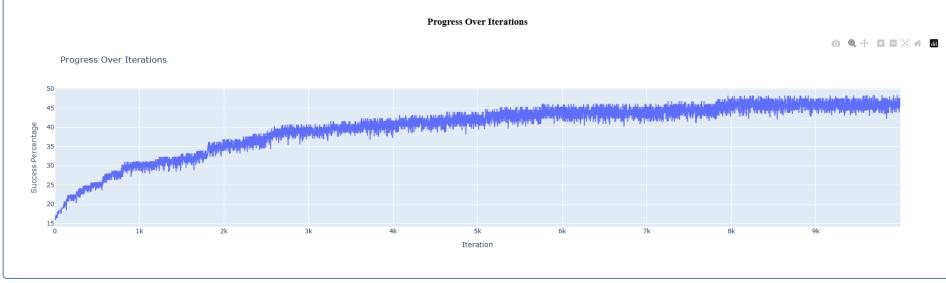
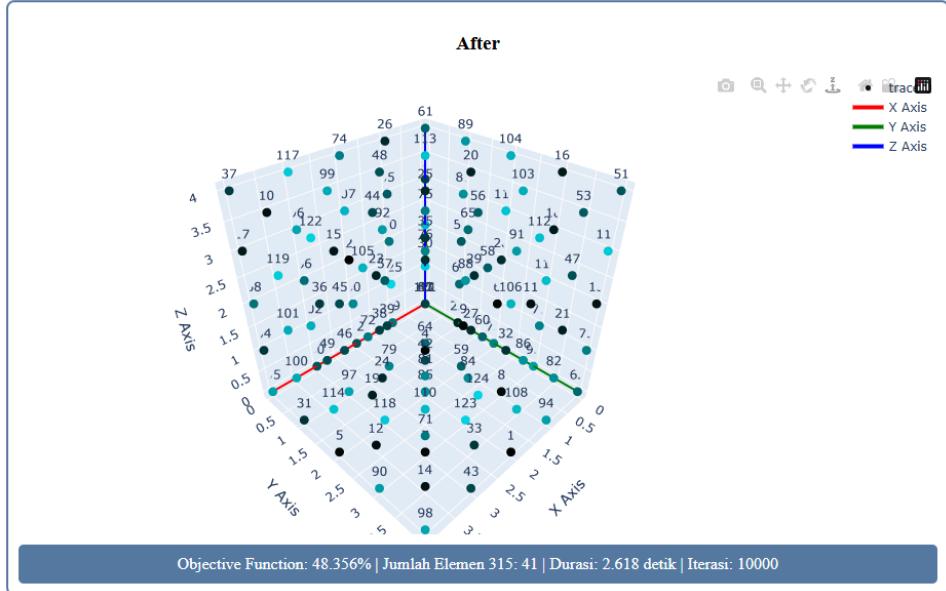


Eksperimen 2

State awal kubus



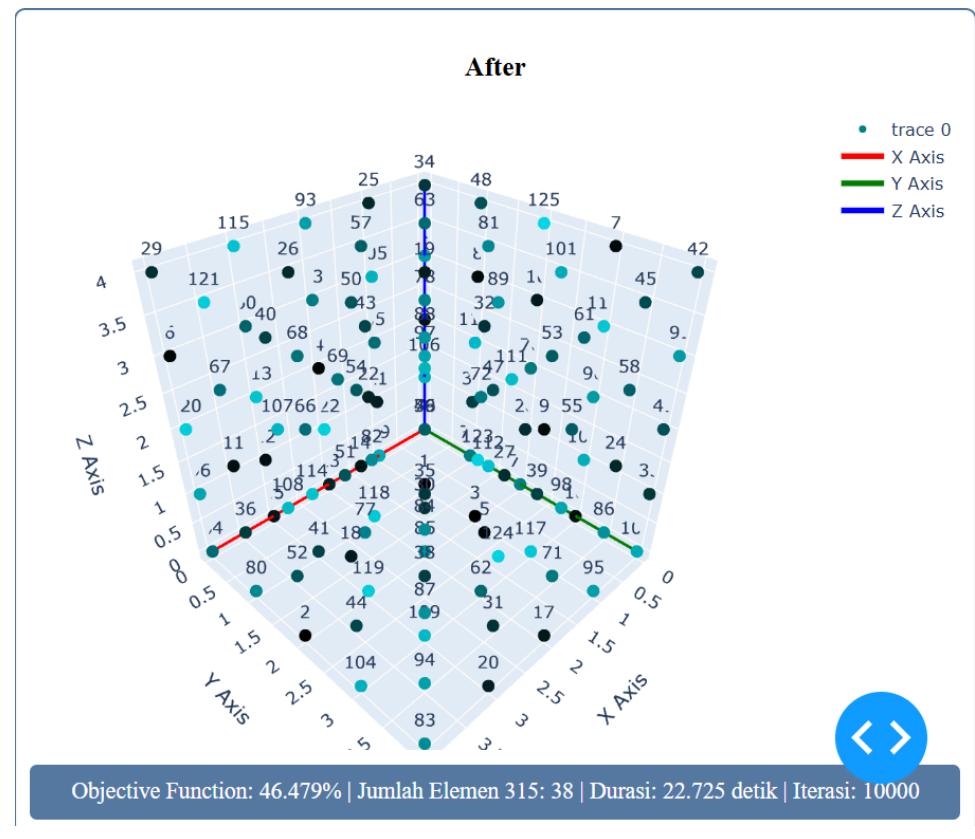
Nilai *objective function* 48.356%

<i>Plot objective function</i>	
Durasi proses pencarian	2.618 detik
Banyak iterasi	10000
State akhir kubus	

### Eksperimen 3

State awal kubus	<p>Before</p> <ul style="list-style-type: none"> <li>● trace 0</li> <li>— X Axis</li> <li>— Y Axis</li> <li>— Z Axis</li> </ul> <p>Objective Function: 15.669%   Initial Jumlah Elemen 315: 0</p>
Nilai <i>objective function</i>	46.479%
Plot <i>objective function</i>	<p>Progress Over Iterations</p> <p>Success Percentage</p> <p>Iteration</p>
Durasi proses pencarian	22.725 detik
Banyak iterasi	10000

State akhir kubus

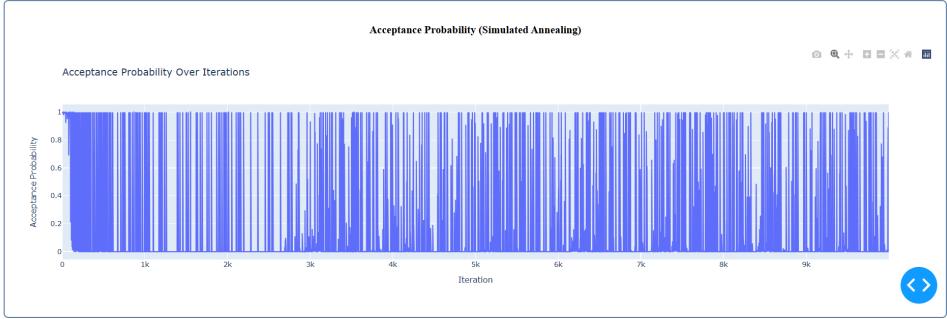


### 2.2.5 Simulated Annealing

Berikut adalah hasil eksperimen untuk *local search* dengan algoritma *Simulated Annealing*.

Eksperimen 1	
State awal kubus	<p>Before</p> <ul style="list-style-type: none"> <li>● trace 0</li> <li>— X Axis</li> <li>— Y Axis</li> <li>— Z Axis</li> </ul> <p>Objective Function: 16.874%   Initial Jumlah Elemen 315: 1</p>
Nilai <i>objective function</i>	42.014%
Plot <i>objective function</i>	<p>Progress Over Iterations</p> <p>Success Percentage</p> <p>Iteration</p>
Durasi proses pencarian	7.008 detik

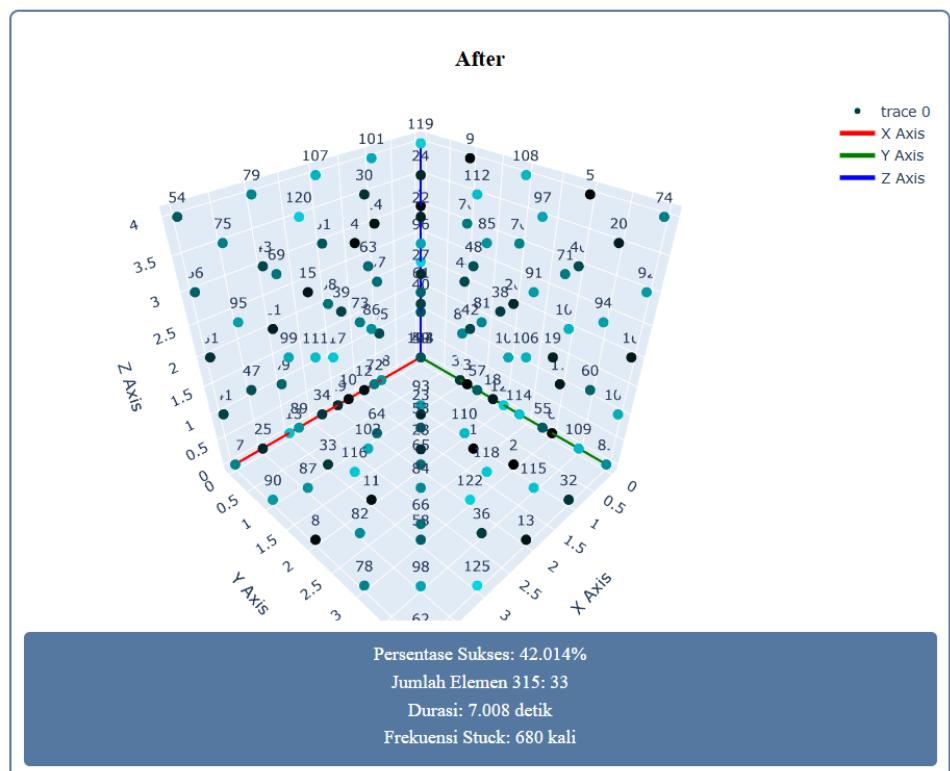
Plot  $e^{\frac{\Delta E}{T}}$  terhadap banyak iterasi yang telah dilewati



Frekuensi 'stuck' di *local optima*

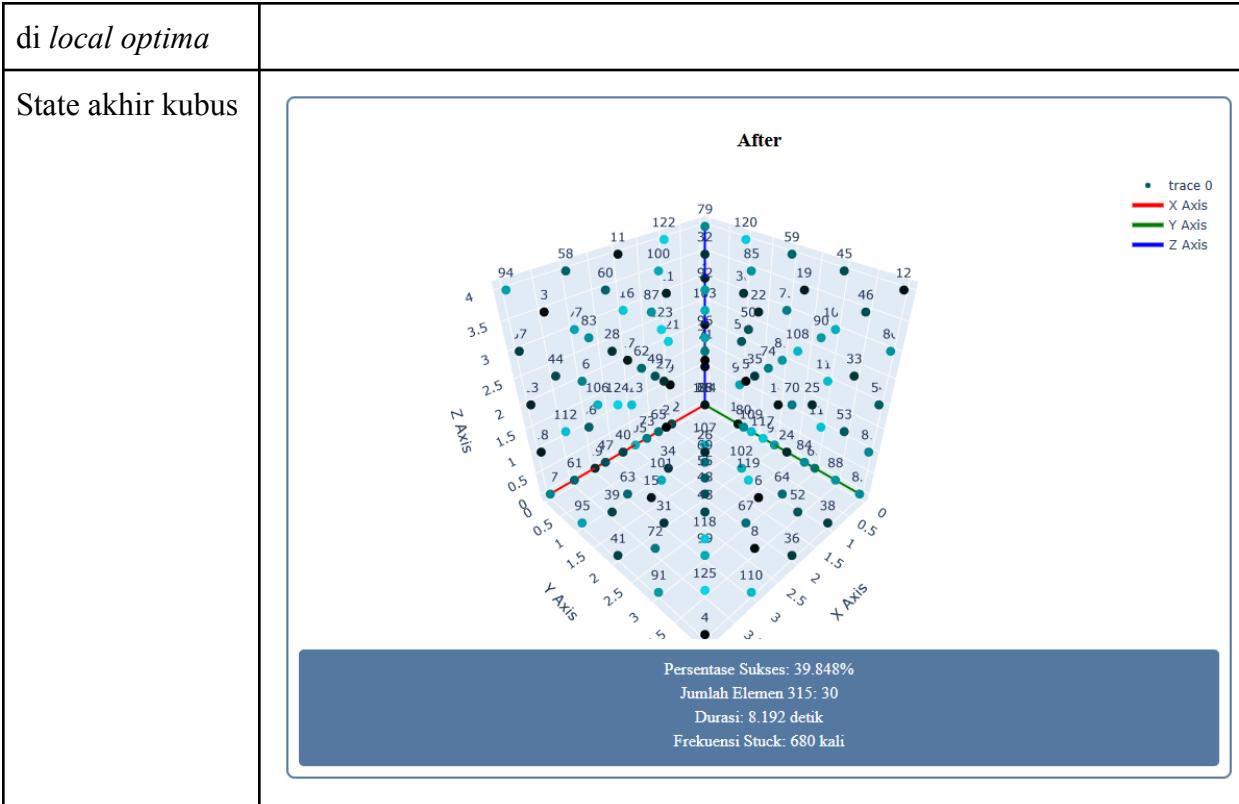
680 kali

State akhir kubus



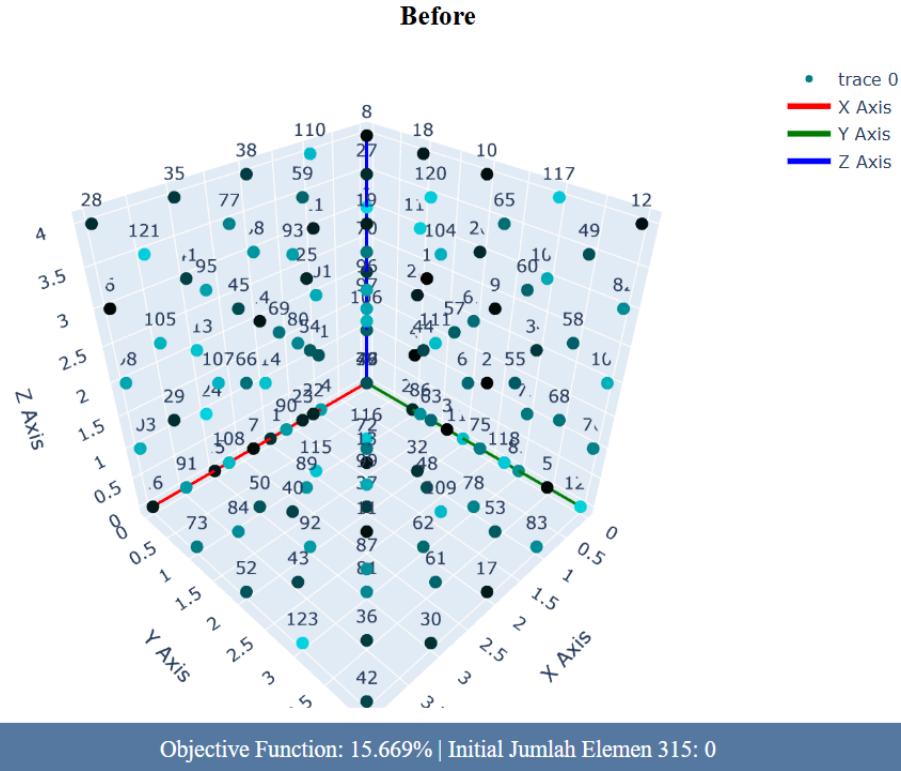
## Eksperimen 2

State awal kubus	<p>Before</p> <ul style="list-style-type: none"> <li>● trace 0</li> <li>— X Axis</li> <li>— Y Axis</li> <li>— Z Axis</li> </ul> <p>Objective Function: 16.072%   Initial Jumlah Elemen 315; 0</p>
Nilai <i>objective function</i>	39.848%
Plot <i>objective function</i>	<p>Progress Over Iterations</p> <p>Success Percentage</p> <p>Iteration</p>
Durasi proses pencarian	8.192 detik
Plot $e^{\frac{\Delta E}{T}}$ terhadap banyak iterasi yang telah dilewati	<p>Acceptance Probability (Simulated Annealing)</p> <p>Acceptance Probability</p> <p>Iteration</p>
Frekuensi ‘stuck’	680 kali



### Eksperimen 3

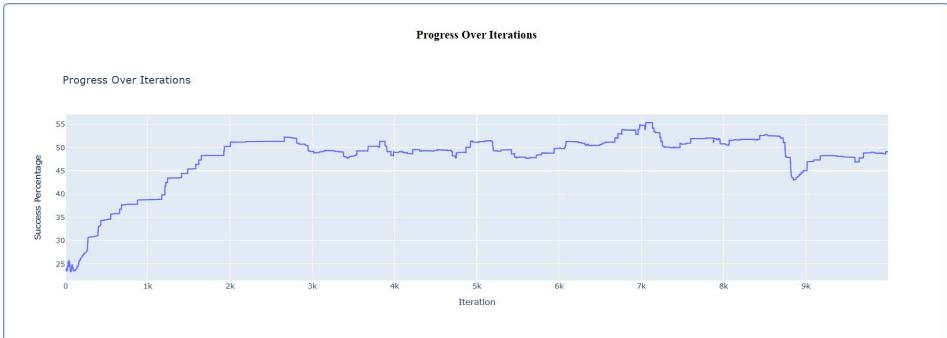
State awal kubus



Nilai *objective function*

55.32%

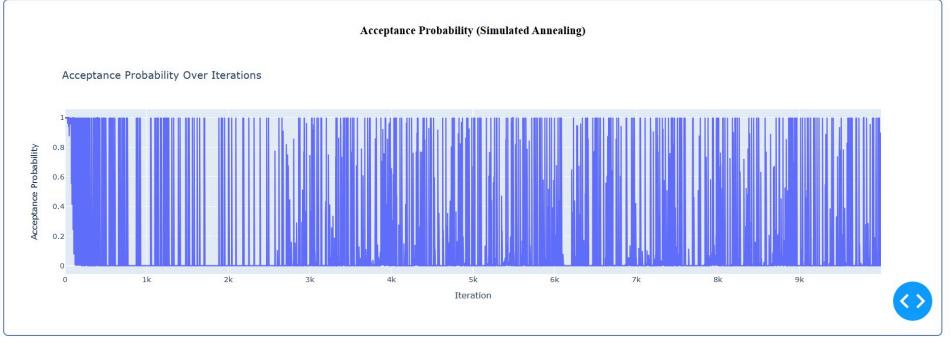
Plot *objective function*



Durasi proses pencarian

2.197 detik

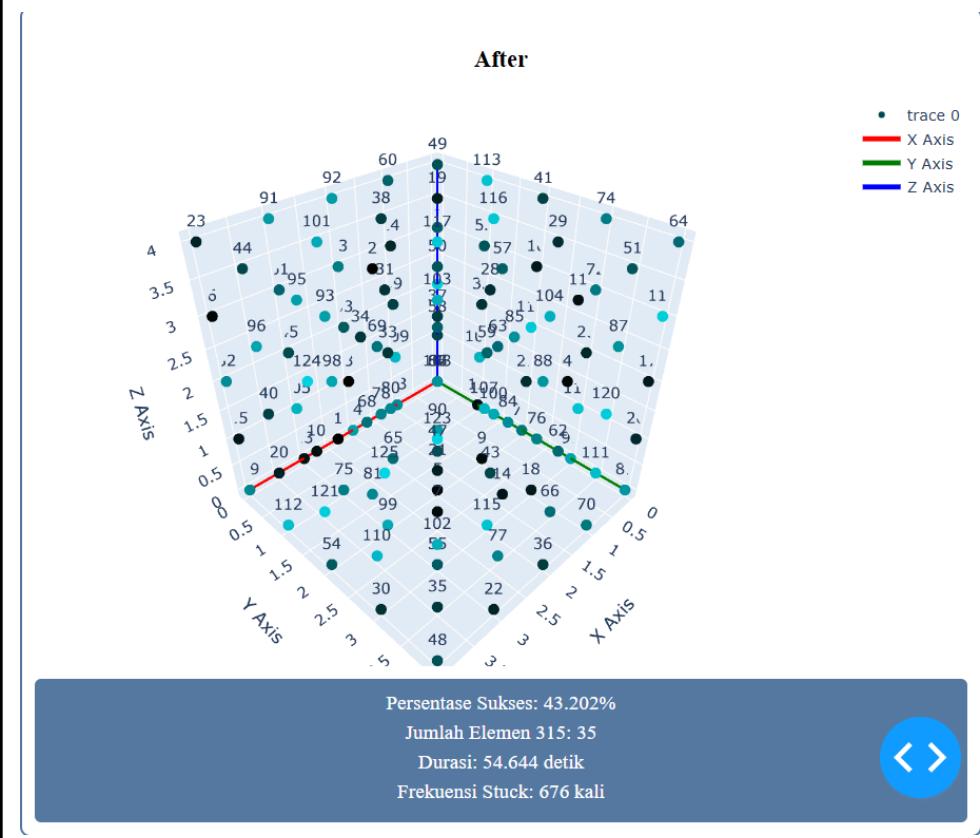
Plot  $e^{\frac{\Delta E}{T}}$  terhadap banyak iterasi yang telah dilewati



Frekuensi ‘stuck’ di *local optima*

676 kali

State akhir kubus



### 2.1.6 Genetic Algorithm

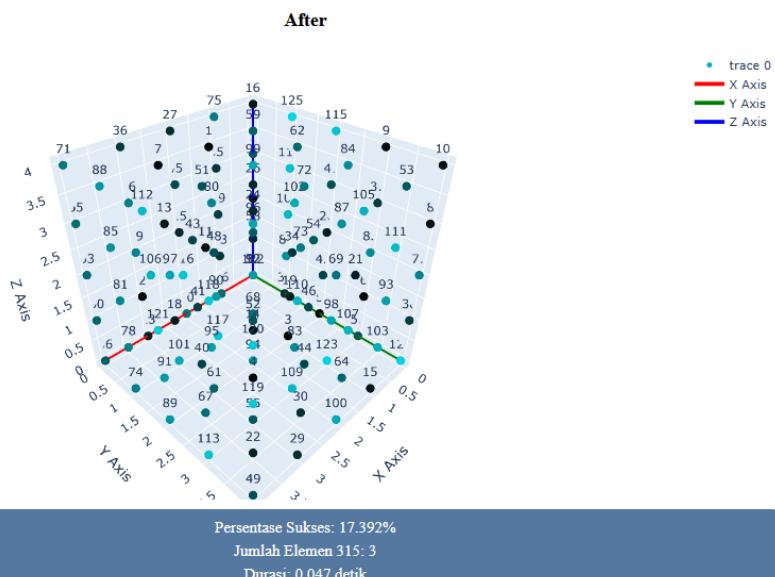
Berikut adalah hasil eksperimen untuk *local search* dengan algoritma *Genetic Algorithm* dengan variasi banyak iterasi dan jumlah populasi sebagai kontrol.

a. Variasi Iterasi (Jumlah populasi konstan = 10)

i. Banyak iterasi = 10

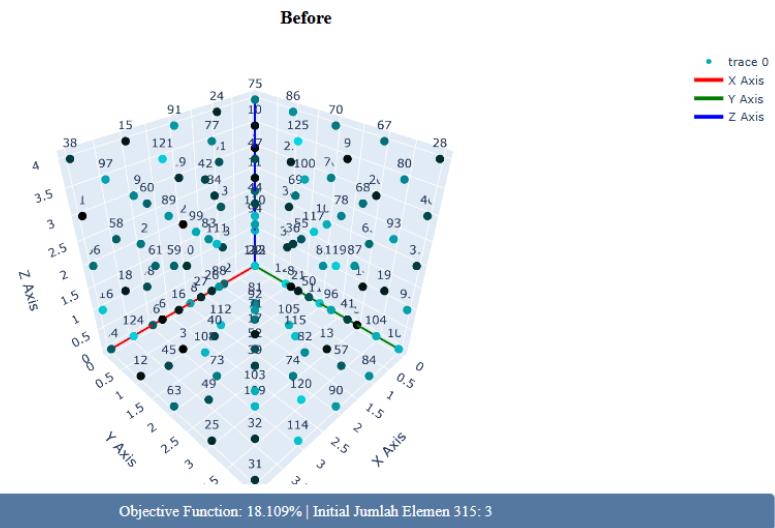
Eksperimen 1	
State awal kubus	<p>Before</p> <p>● trace 0 — X Axis — Y Axis — Z Axis</p> <p>Objective Function: 16.25%   Initial Jumlah Elemen 315: 1</p>
Nilai objective function	17.392%
Plot objective function	<p>Genetic Algorithm Progress</p> <p>Average Accuracy</p> <p>Generation</p> <p>Progress Over Iterations</p>
Jumlah populasi	10
Banyak iterasi	10
Durasi proses pencarian	0.047detik

State akhir kubus



## Eksperimen 2

State awal kubus



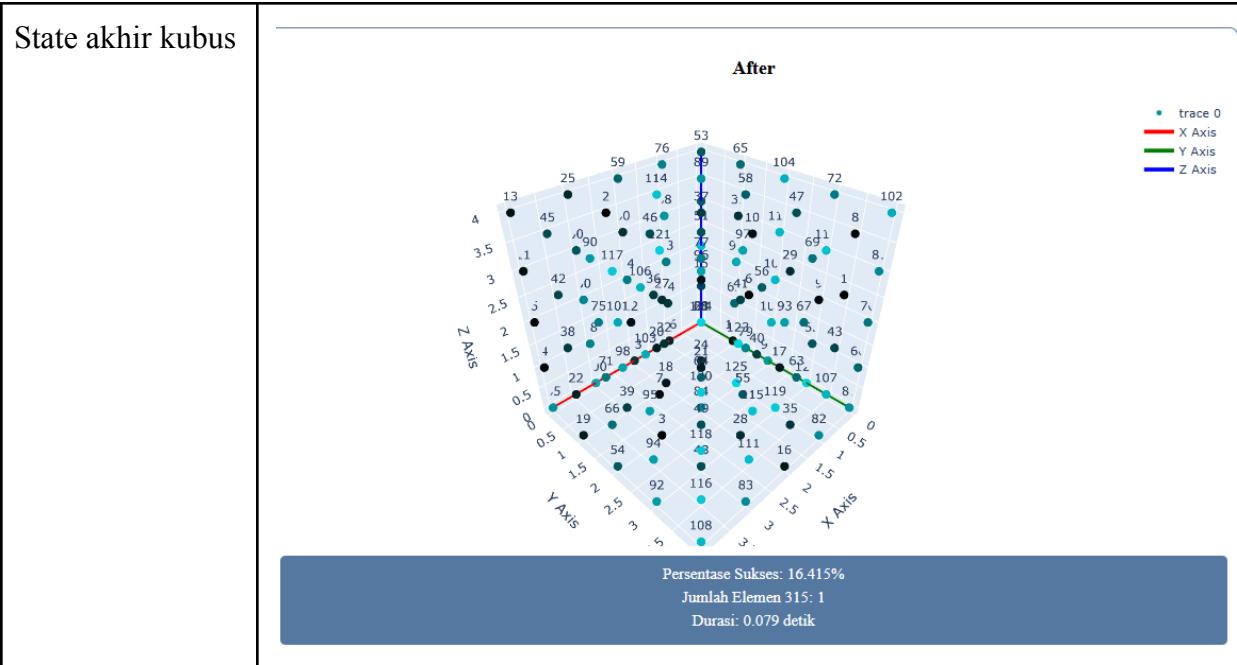
Nilai *objective function*

18.259%

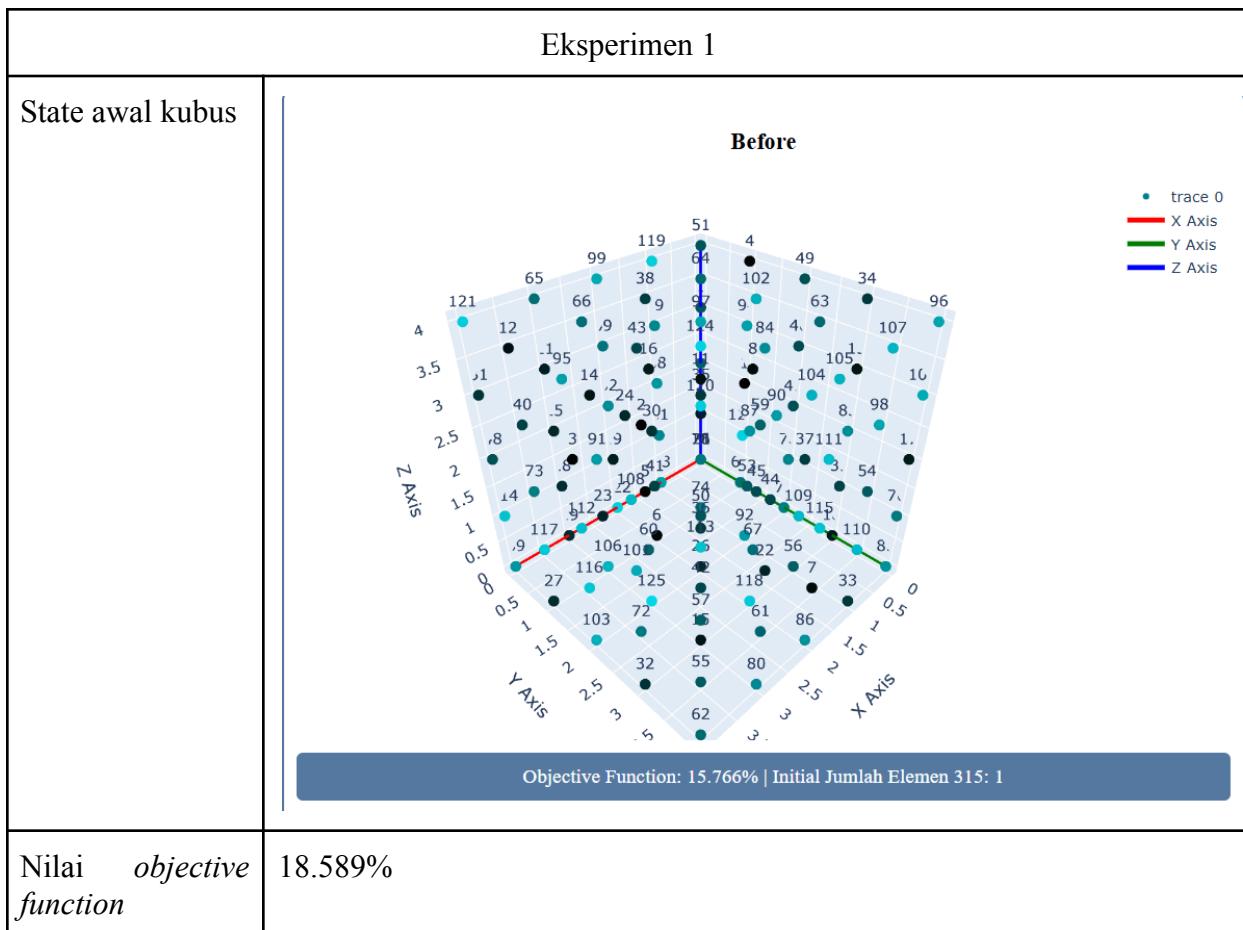
<i>Plot objective function</i>	<p>Progress Over Iterations</p> <p>Average Accuracy</p> <p>Generation</p>
Jumlah populasi	10
Banyak iterasi	10
Durasi proses pencarian	0.05 detik
State akhir kubus	<p>After</p> <p>trace 0</p> <p>X Axis</p> <p>Y Axis</p> <p>Z Axis</p> <p>Persentase Sukses: 18.259%</p> <p>Jumlah Elemen 315: 3</p> <p>Durasi: 0.05 detik</p>

### Eksperimen 3

State awal kubus	<p>Before</p> <ul style="list-style-type: none"> <li>● trace 0</li> <li>X Axis</li> <li>Y Axis</li> <li>Z Axis</li> </ul> <p>Objective Function: 15.7%   Initial Jumlah Elemen 315: 0</p>
Nilai <i>objective function</i>	16.415%
Plot <i>objective function</i>	<p>Progress Over Iterations</p> <p>Genetic Algorithm Progress</p> <p>Average Accuracy</p> <p>Generation</p>
Jumlah populasi	10
Banyak iterasi	10
Durasi proses pencarian	0.079 detik



ii. Banyak iterasi = 1000



*Plot objective function*



Jumlah populasi

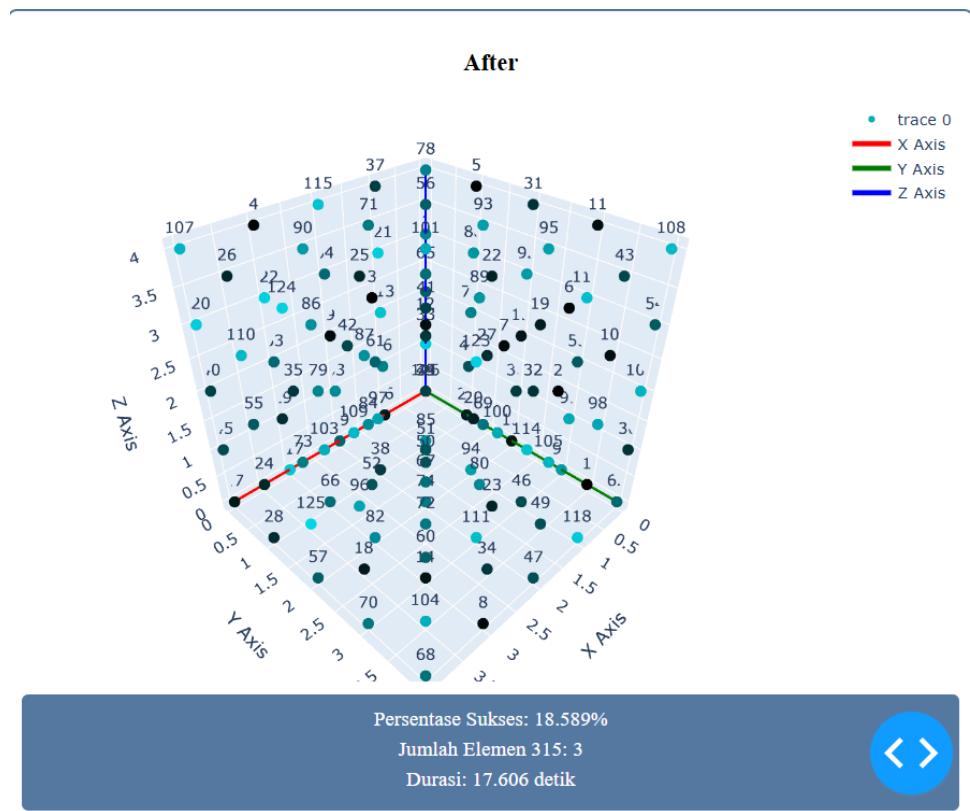
10

Banyak iterasi

1000

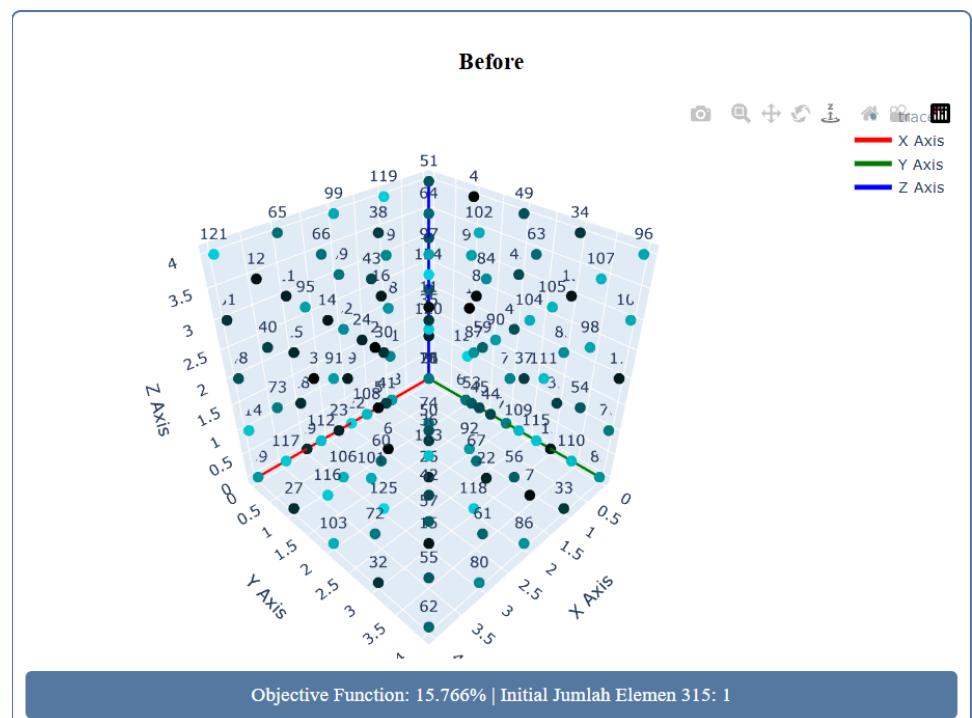
Durasi proses pencarian

State akhir kubus



## Eksperimen 2

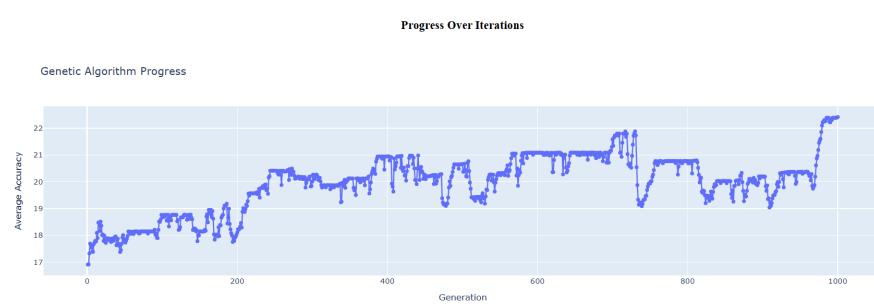
State awal kubus



Nilai *objective function*

22.427%

Plot *objective function*



Jumlah populasi

10

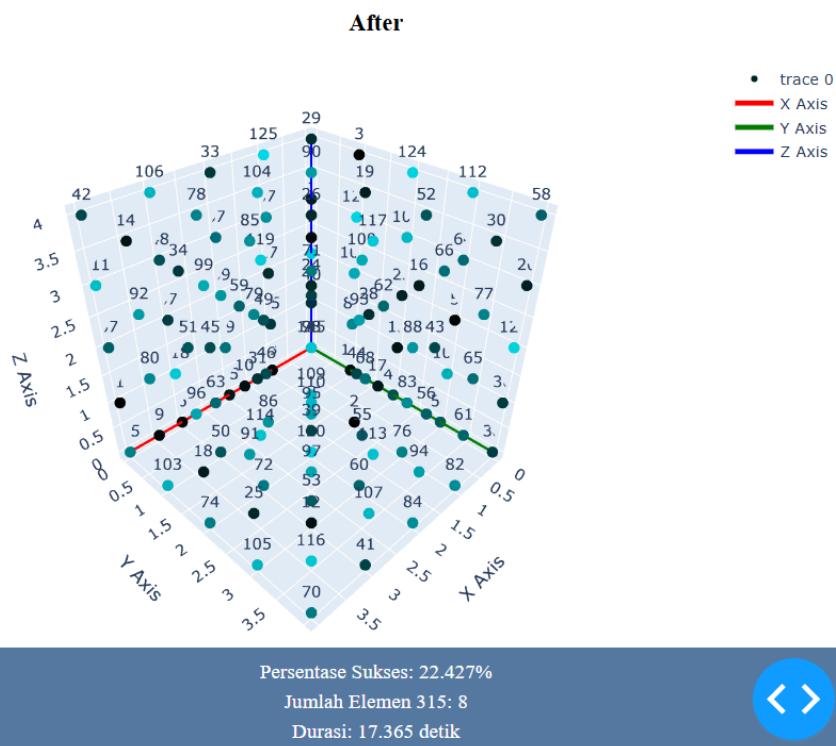
Banyak iterasi

1000

Durasi proses pencarian

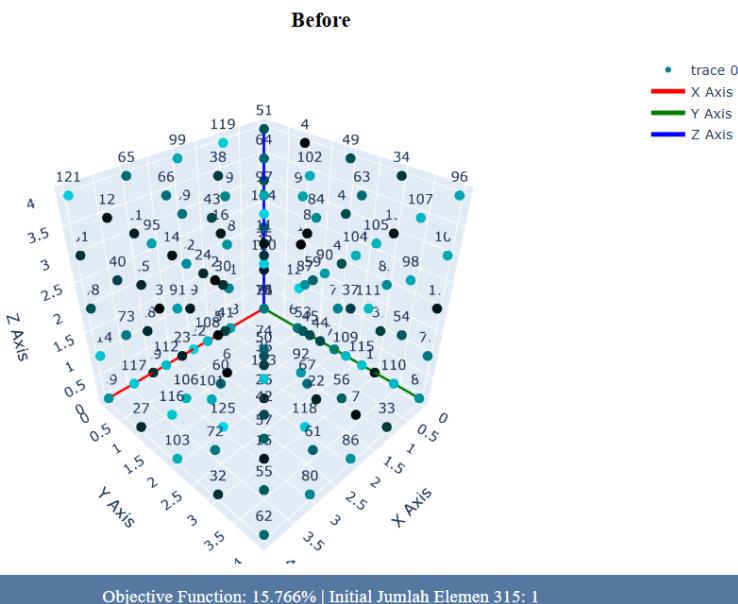
17.365 detik

### State akhir kubus



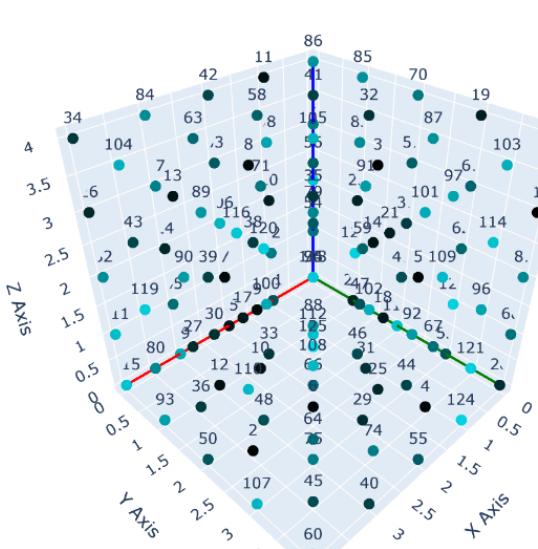
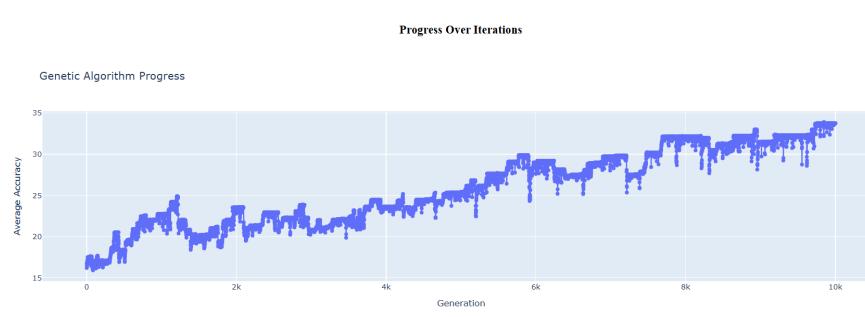
### Eksperimen 3

#### State awal kubus

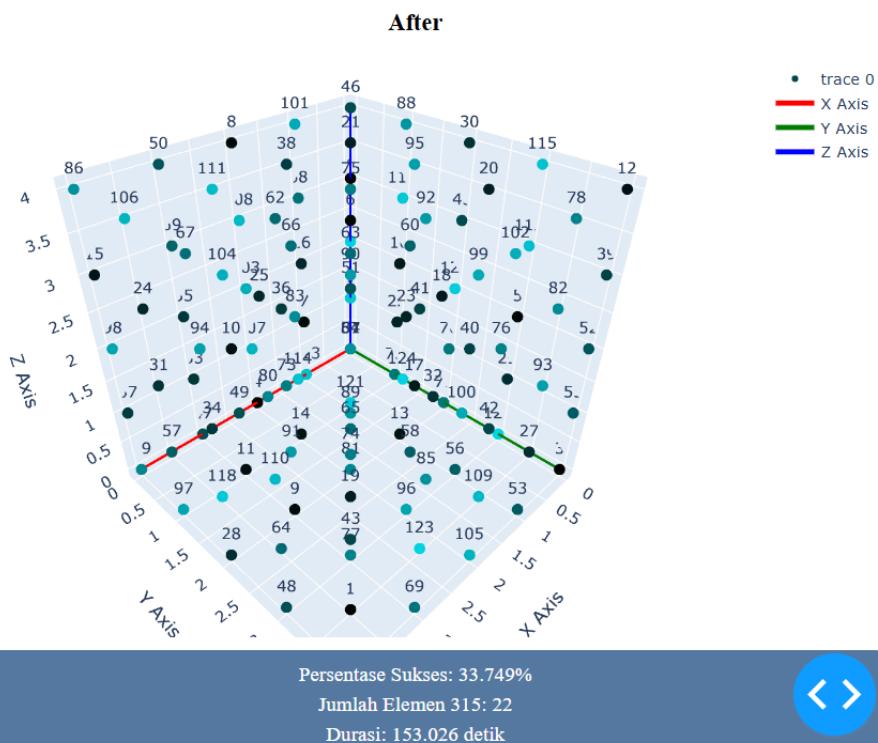


Nilai <i>objective function</i>	20.047%
Plot <i>objective function</i>	
Jumlah populasi	10
Banyak iterasi	1000
Durasi proses pencarian	17.931 detik
State akhir kubus	<p style="text-align: center;"><b>After</b></p> <p style="text-align: center;">Percentase Sukses: 20.047%</p> <p style="text-align: center;">Jumlah Elemen 315: 4</p> <p style="text-align: center;">Durasi: 17.931 detik</p>

iii. Banyak iterasi = 10000

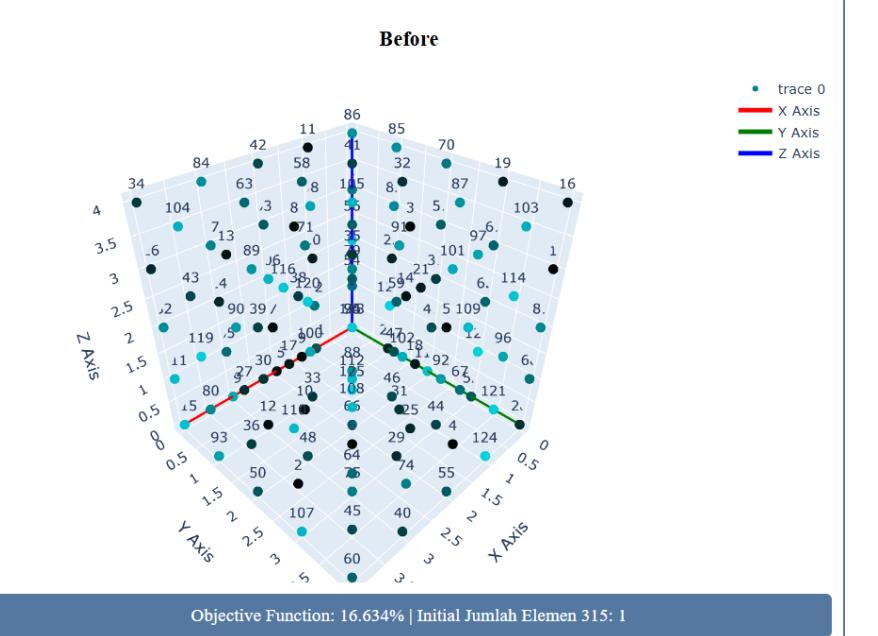
Eksperimen 1	
State awal kubus	<p><b>Before</b></p>  <p>Objective Function: 16.634%   Initial Jumlah Elemen 315: 1</p>
Nilai <i>objective function</i>	33.749%
Plot <i>objective function</i>	<p>Progress Over Iterations</p> 
Jumlah populasi	10
Banyak iterasi	10000
Durasi proses pencarian	153.026 detik

### State akhir kubus



### Eksperimen 2

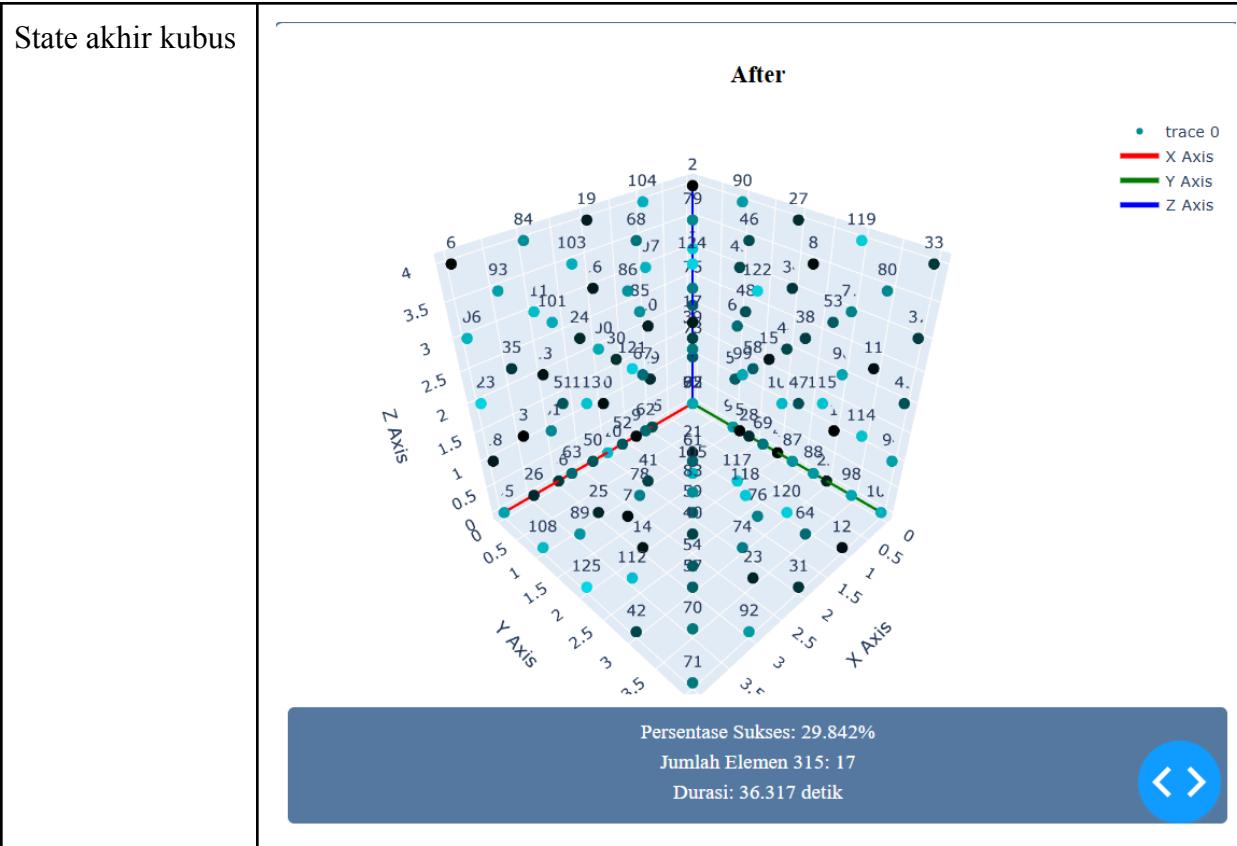
#### State awal kubus



Nilai <i>objective function</i>	28.528%
Plot <i>objective function</i>	
Jumlah populasi	10
Banyak iterasi	10000
Durasi proses pencarian	86.74 detik
State akhir kubus	<p style="text-align: center;"><b>After</b></p> <p style="text-align: center;">Percentase Sukses: 28.528% Jumlah Elemen 315: 15 Durasi: 86.74 detik</p>

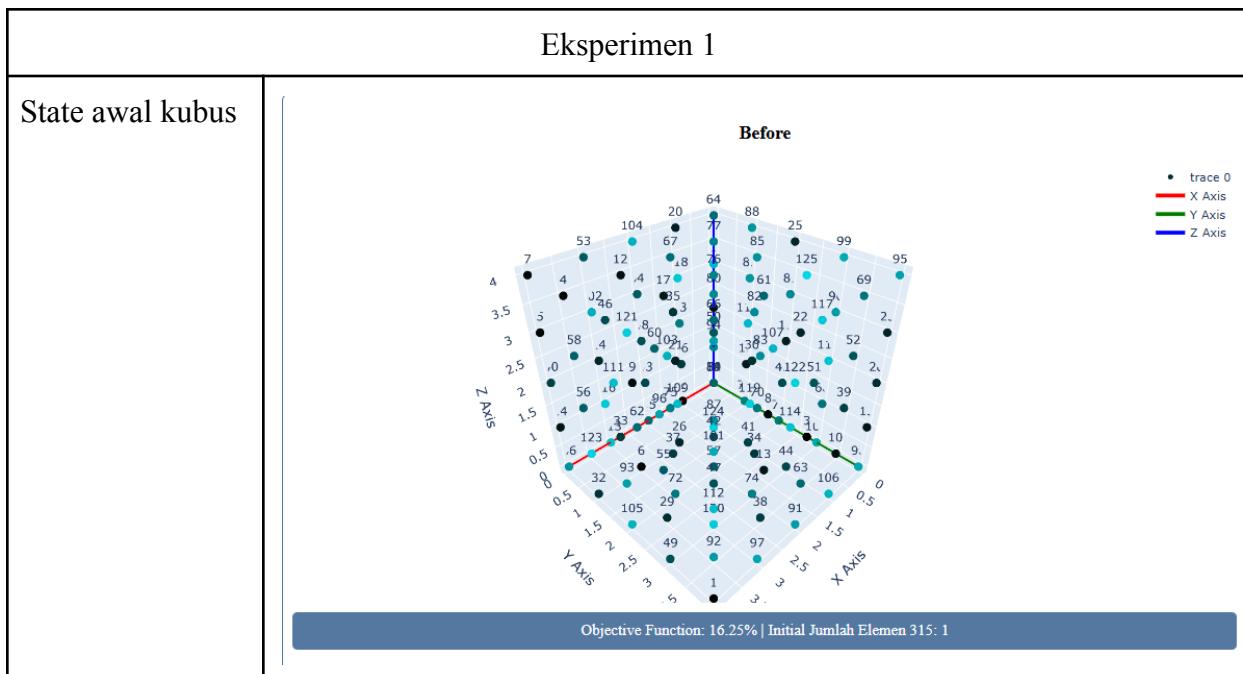
### Eksperimen 3

State awal kubus	<p>Before</p> <p>trace 0 X Axis Y Axis Z Axis</p> <p>Objective Function: 16.634%   Initial Jumlah Elemen 315: 1</p>
Nilai <i>objective function</i>	29.842%
Plot <i>objective function</i>	<p>Progress Over Iterations</p> <p>Average Accuracy</p> <p>Generation</p>
Jumlah populasi	10
Banyak iterasi	10000
Durasi proses pencarian	36.317 detik



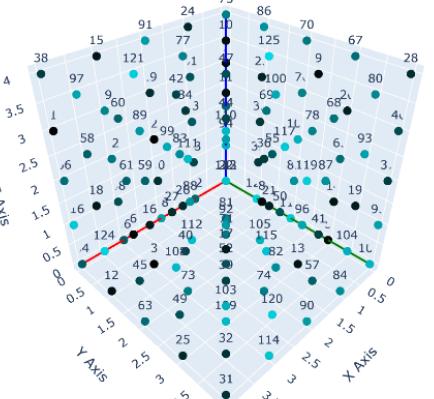
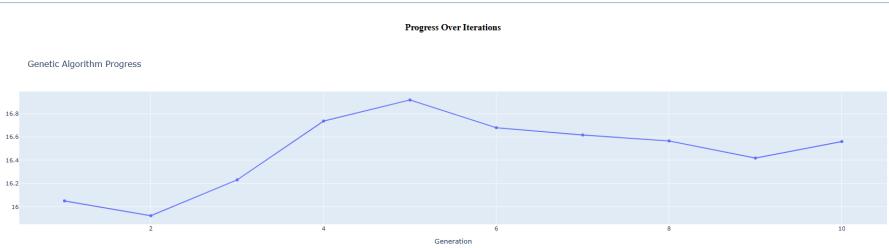
b. Variasi Populasi (Jumlah iterasi konstan = 10)

i. Banyak populasi = 10

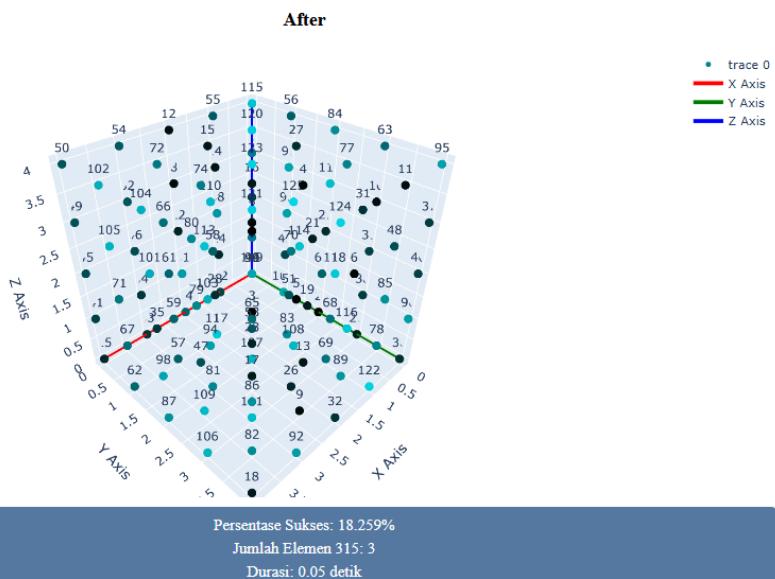


Nilai <i>objective function</i>	17.392%
Plot <i>objective function</i>	<p>Progress Over Iterations</p> <p>Genetic Algorithm Progress</p> <p>Average Accuracy</p> <p>Generation</p>
Jumlah populasi	10
Banyak iterasi	10
Durasi proses pencarian	0.047detik
State akhir kubus	<p>After</p> <p>trace 0</p> <p>X Axis</p> <p>Y Axis</p> <p>Z Axis</p> <p>Average Accuracy: 17.392%</p> <p>Jumlah Elemen: 315 : 3</p> <p>Durasi: 0.047 detik</p>

## Eksperimen 2

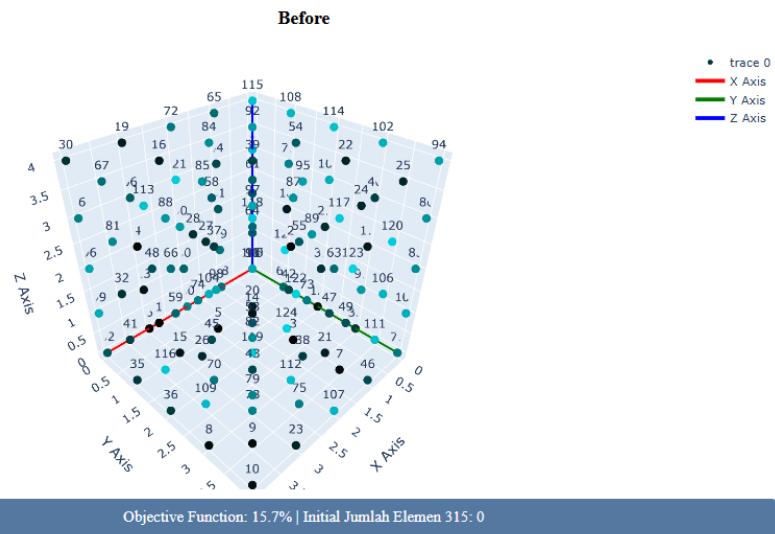
State awal kubus	 <p>Before</p> <p>trace 0 X Axis Y Axis Z Axis</p> <p>Objective Function: 18.109%   Initial Jumlah Elemen 315: 3</p>
Nilai <i>objective function</i>	18.259%
Plot <i>objective function</i>	 <p>Progress Over Iterations</p> <p>Genetic Algorithm Progress</p> <p>Karakteristik Objektif</p> <p>Generation</p>
Jumlah populasi	10
Banyak iterasi	10
Durasi proses pencarian	0.05 detik

State akhir kubus



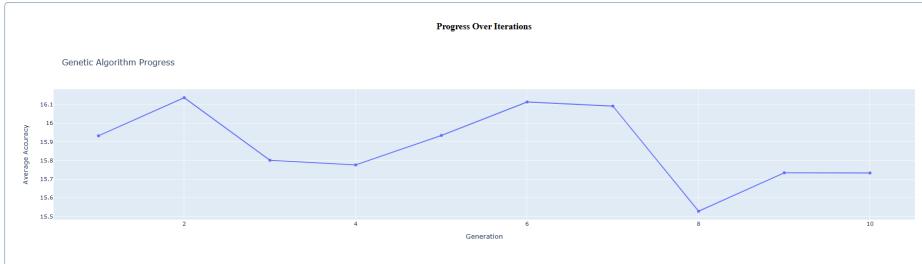
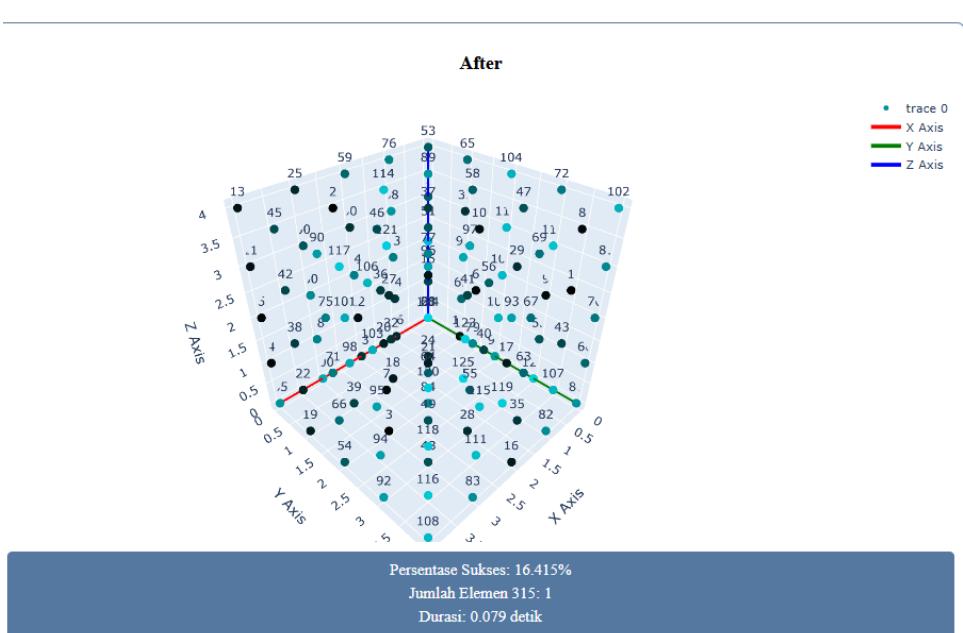
### Eksperimen 3

State awal kubus



Nilai *objective function*

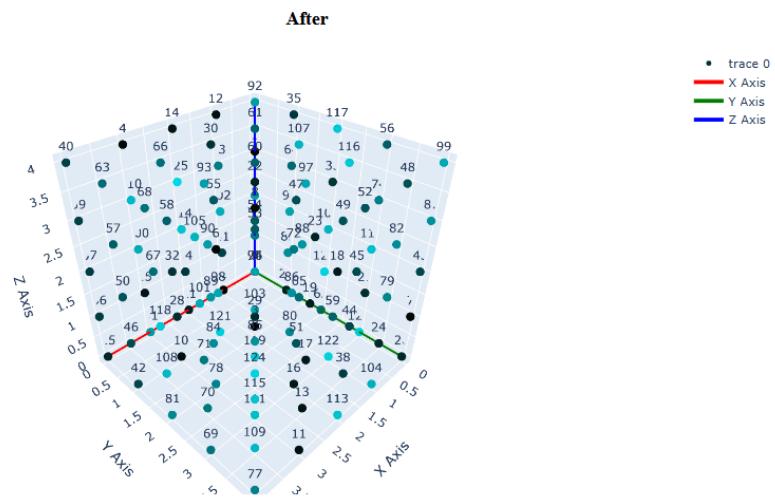
16.415%

<i>Plot objective function</i>	
Jumlah populasi	10
Banyak iterasi	10
Durasi proses pencarian	0.079 detik
State akhir kubus	 <p>Percentase Sukses: 16.415%  Jumlah Elemen: 315  Durasi: 0.079 detik</p>

ii. Banyak populasi = 1000

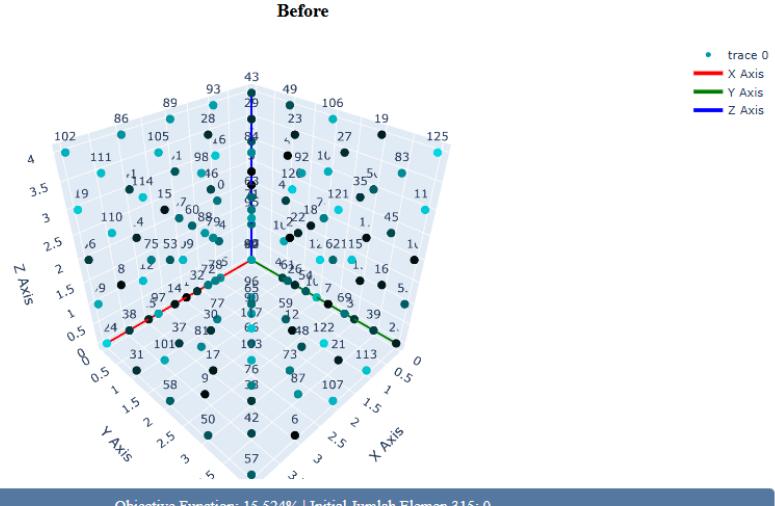
Eksperimen 1	
State awal kubus	<p>Before</p> <p>trace 0 X Axis Y Axis Z Axis</p> <p>Objective Function: 16.557%   Initial Jumlah Elemen 315: 1</p>
Nilai <i>objective function</i>	20.089%
Plot <i>objective function</i>	<p>Genetic Algorithm Progress</p> <p>Average Accuracy</p> <p>Progress Over Iterations</p> <p>Generation</p>
Jumlah populasi	1000
Banyak iterasi	10
Durasi proses pencarian	6.221 detik

State akhir kubus



Eksperimen 2

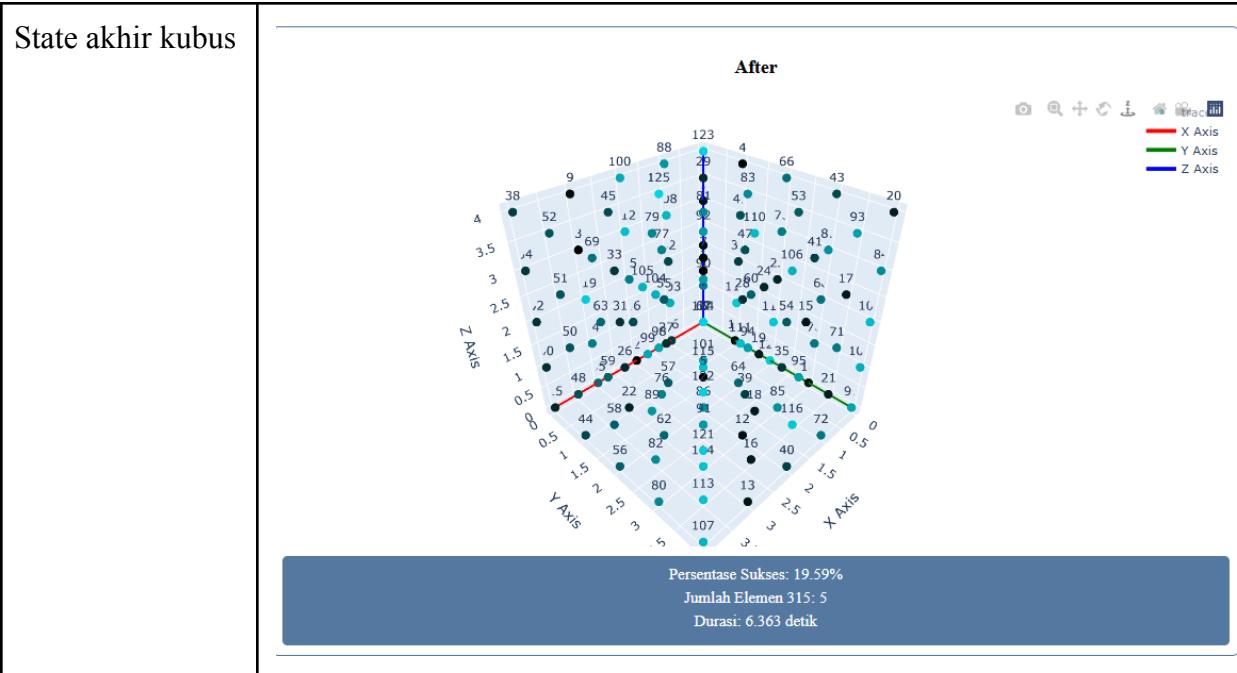
State awal kubus



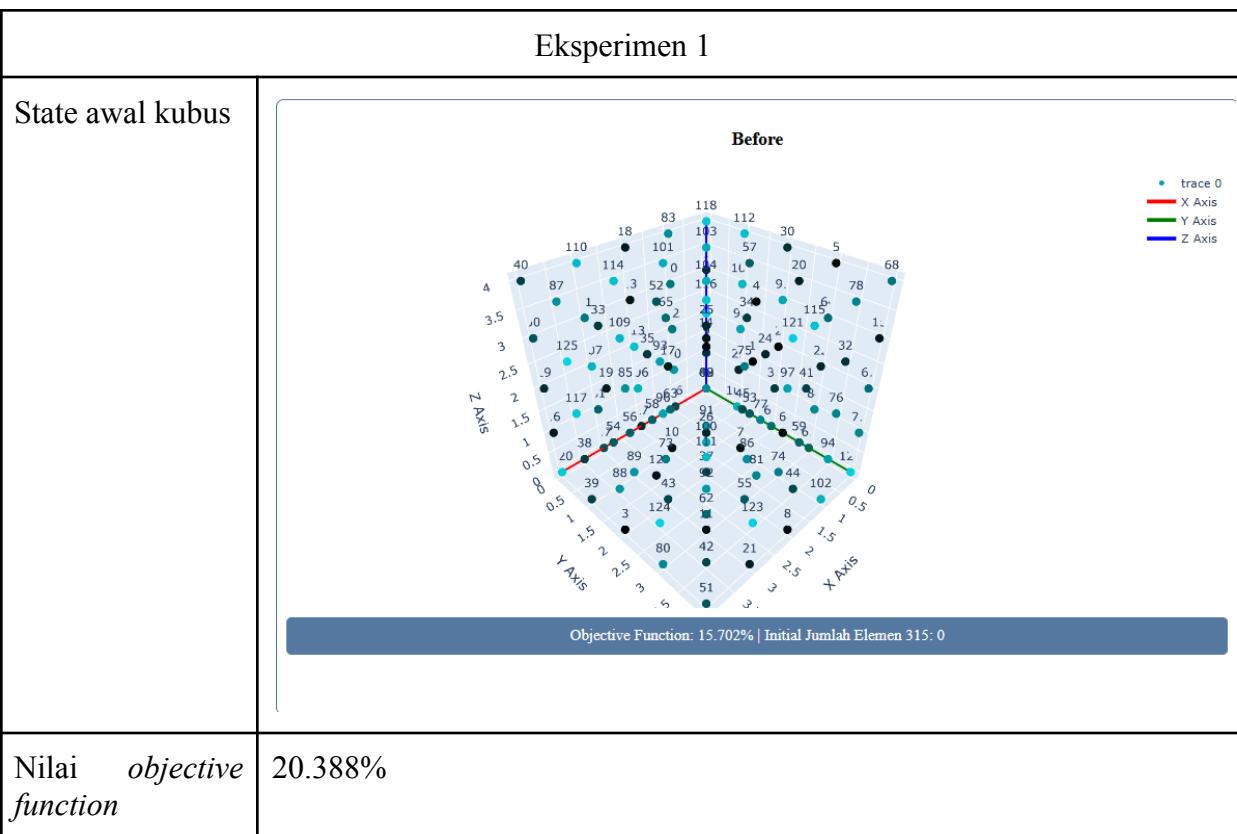
Nilai *objective function* 20.611%

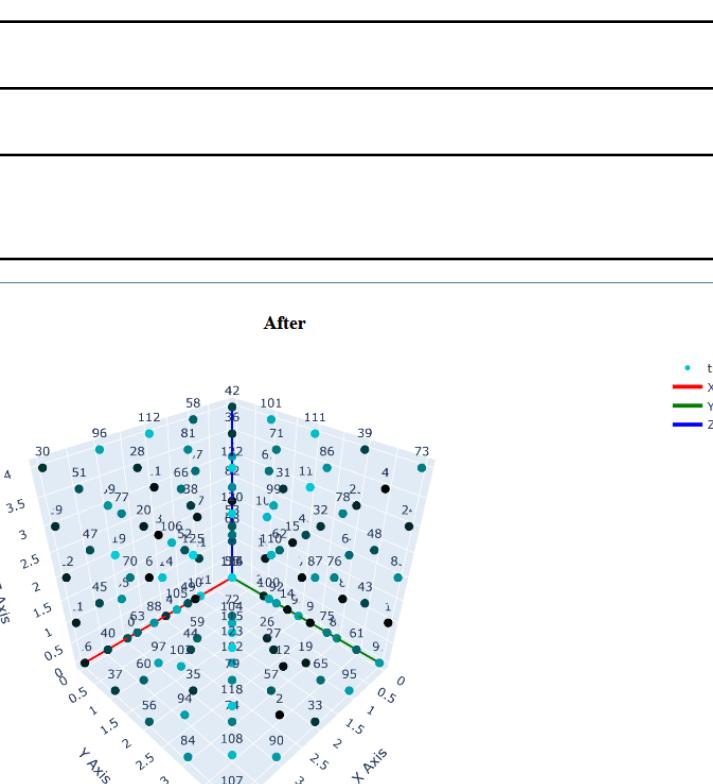
<i>Plot objective function</i>	<p>Genetic Algorithm Progress</p> <p>Average Accuracy</p> <p>Generation</p> <p>Progress Over Iterations</p>
Jumlah populasi	1000
Banyak iterasi	10
Durasi proses pencarian	6.249 detik
State akhir kubus	<p>After</p> <ul style="list-style-type: none"> <li>● trace 0</li> <li>— X Axis</li> <li>— Y Axis</li> <li>— Z Axis</li> </ul> <p>Persentase Sukses: 20.611%</p> <p>Jumlah Elemen: 315: 7</p> <p>Durasi: 6.249 detik</p>

Eksperimen 3	
State awal kubus	<p>Before</p> <p>trace 0 X Axis Y Axis Z Axis</p> <p>Objective Function: 16.588%   Initial Jumlah Elemen 315: 1</p>
Nilai <i>objective function</i>	19.59%
Plot <i>objective function</i>	<p>Progress Over Iterations</p> <p>Genetic Algorithm Progress</p> <p>AVERAGE ACCURACY</p> <p>Generation</p>
Jumlah populasi	1000
Banyak iterasi	10
Durasi proses pencarian	6.363 detik



iii. Banyak populasi = 10000

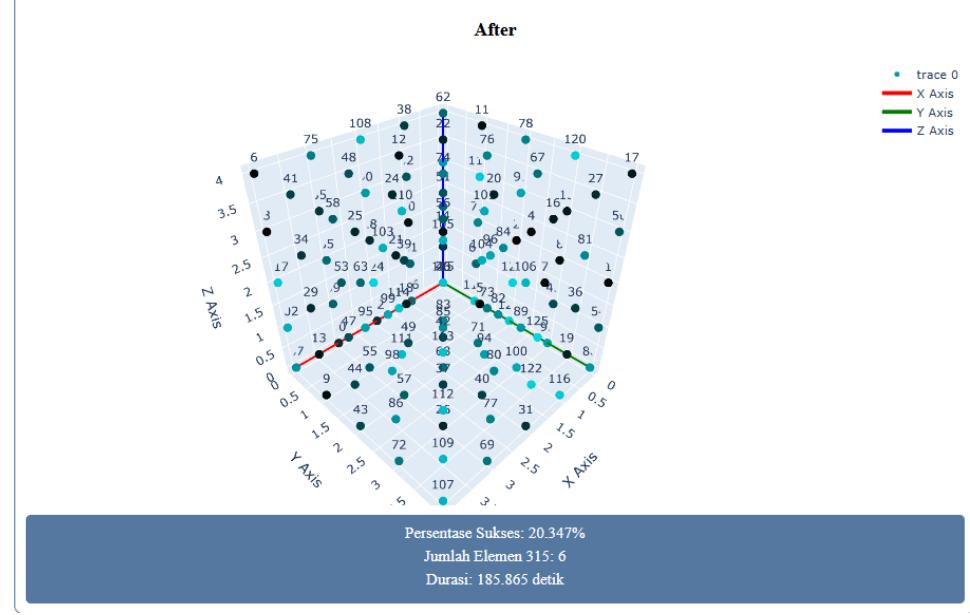


Plot objective function	
	
Jumlah populasi	10000
Banyak iterasi	10
Durasi proses pencarian	785.045 detik
State akhir kubus	 <p>Percentase Sukses: 20.388% Jumlah Elemen 315: 6 Durasi: 785.045 detik</p>

## Eksperimen 2

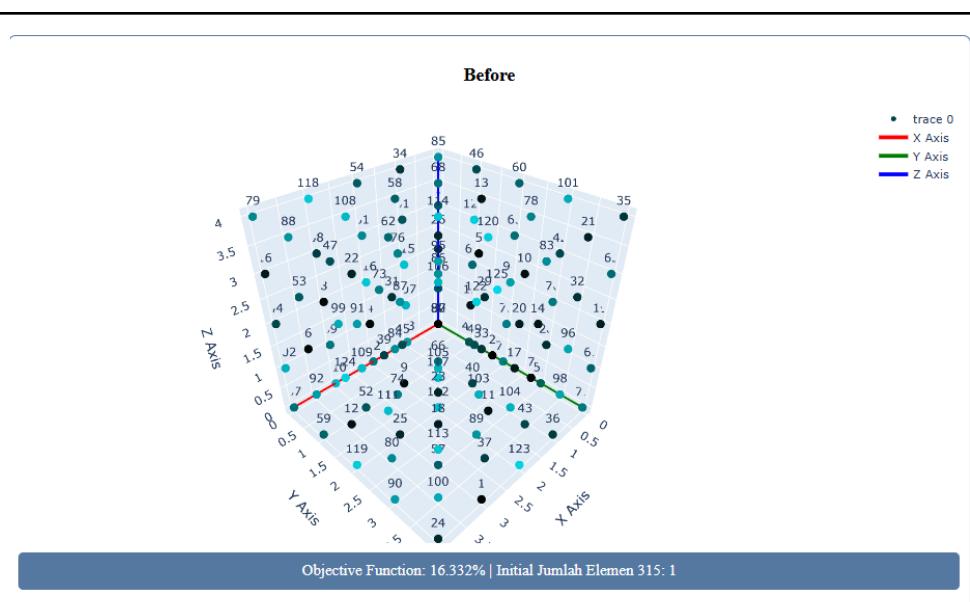
State awal kubus	
Nilai <i>objective function</i>	20.34%
<i>Plot objective function</i>	
Jumlah populasi	1000
Banyak iterasi	10
Durasi proses pencarian	185.865 detik

State akhir kubus



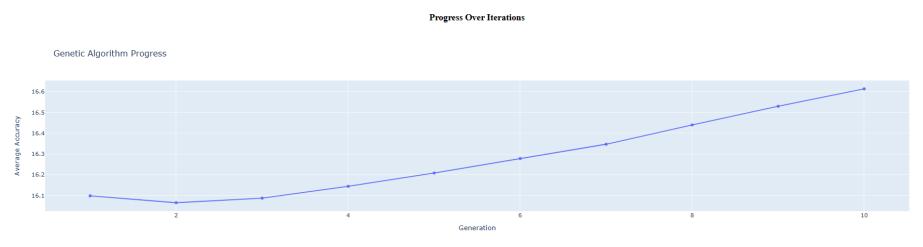
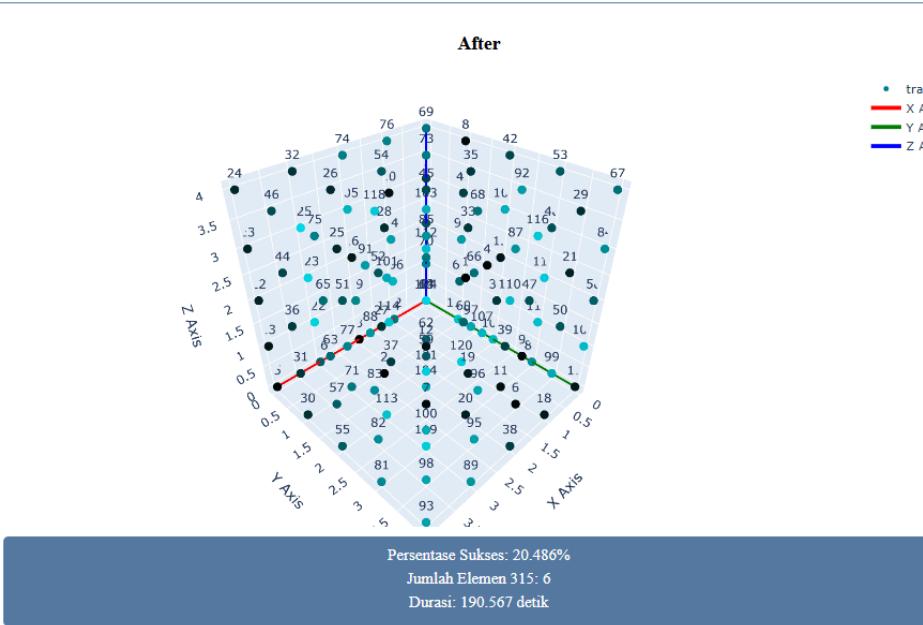
### Eksperimen 3

State awal kubus



Nilai objective function

190.567 detik

Plot <i>objective function</i>	
Jumlah populasi	10000
Banyak iterasi	10
Durasi proses pencarian	20.486%
State akhir kubus	

Berikut merupakan hasil analisis berdasarkan hasil eksperimen yang telah dilakukan di atas.

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

- *Steepest Ascent Hill-climbing*

Untuk mendekati *global optima*, algoritma ini mencari peningkatan terbesar pada setiap langkahnya. Hasil eksperimen menunjukkan konsistensi pada nilai *objective function* yang relatif tinggi. Namun, pendekatan ini memiliki risiko

terjebak pada *local optima* karena algoritma ini selalu mencari kenaikan tertinggi dan tidak mungkin ada penurunan sementara sehingga algoritma ini mungkin sulit mencapai *global optima* jika *objective function* memiliki banyak puncak lokal.

- *Hill-climbing with Sideways Move*

Algoritma ini memungkinkan langkah ke samping jika tidak ada peningkatan langsung sehingga mampu mendekati *global optima*. Dengan *sideways move* yang diperbolehkan hingga jumlah langkah tertentu, algoritma ini dapat keluar dari kondisi *stuck* di beberapa *local optima* yang lebih baik namun mungkin bukan *global optima*.

- *Random Restart Hill-climbing*

Algoritma ini memiliki mampu mendekati *global optima* dengan cukup baik karena melakukan *restart* dari berbagai titik awal yang berbeda. Dengan memulai pencarian dari beberapa *random initial state*, algoritma ini memiliki kemampuan untuk keluar dari *local optima* untuk mengeksplorasi berbagai ruang solusi dalam upaya untuk menemukan solusi yang paling optimal. Perlu dipahami bahwa setiap iterasi dapat mencapai *local optima*, namun kelemahan ini diatasi dengan strategi *restart*.

- *Stochastic Hill-Climbing*

Algoritma ini memilih langkah berikutnya secara acak dengan probabilitas berdasarkan nilai *objective* yang meningkat, tetapi karena pilihan acak, terkadang menyebabkan langkah-langkah yang kurang optimal pada waktu tertentu. Algoritma ini lebih fleksibel dalam keluar dari *local optima*, tetapi pendekatan acak membuatnya lebih lambat dan kurang presisi dalam mencapai puncak yang lebih tinggi, sehingga jarang mendekati *global optima*.

- *Simulated Annealing*

Algoritma ini menggunakan mekanisme penurunan suhu yang memungkinkan langkah mundur untuk menghindari terjebak di *local optima* yang membuatnya memiliki peluang yang lebih besar untuk mendekati *global optima*. Berdasarkan eksperimen, hasil yang didapat bervariasi karena algoritma ini mampu mendekati *objective function* yang lebih tinggi karena adanya mekanisme penurunan suhu yang dapat diatur dengan tepat. Variabilitas pada hasilnya menunjukkan bahwa

algoritma ini tidak selalu optimal, tapi bisa mendekati global optima dalam beberapa skenario.

- *Genetic Algorithm*

Pendekatan yang dilakukan oleh *genetic algorithm* menuju *global optima* dilakukan melalui seleksi, mutasi, dan *crossover* dalam populasi solusi. Algoritma ini memungkinkan pencarian dalam ruang solusi yang lebih luas sehingga memiliki peluang untuk mencapai *global optima*. Namun, hasil eksperimen menunjukkan bahwa *genetic algorithm* membutuhkan jumlah iterasi dan populasi yang besar untuk mendekati nilai optimal yang lebih tinggi serta penyesuaian parameter yang tepat untuk mencapai performa optimal.

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

- *Steepest Ascent Hill-climbing*

Rata-rata akurasi untuk algoritma ini berkisar antara 19% hingga 20.6%, menggunakan 1000 populasi dengan 10 iterasi, algoritma mencapai akurasi hingga 20.6%. Dengan populasi 10.000, akurasi mencapai sekitar 20.3-20.4%, tetapi durasi proses pencarian meningkat secara signifikan

- *Hill-climbing with Sideways Move*

Rata-rata akurasi untuk algoritma ini berkisar antara 53.96% hingga 56.1%, algoritma ini menunjukkan hasil yang konsisten dengan variansi kecil pada tiap percobaan, sehingga dapat dianggap cukup stabil.

- *Random Restart Hill-climbing*

Rata-rata akurasi untuk algoritma ini berkisar antara 52.705% hingga 54.316%. Dengan 3 kali *restart*, algoritma ini menunjukkan kemampuan untuk mencapai nilai akurasi yang lebih tinggi dibandingkan *Steepest Ascent Hill-climbing*, meskipun memakan waktu komputasi yang lebih lama.

- *Stochastic Hill-Climbing*

Algoritma ini menggapai nilai akurasi dengan rentang 45.056% hingga 48.356% dalam eksperimen. Algoritma ini cenderung lebih cepat dengan iterasi sekitar 10.000 percobaan, meskipun hasilnya tidak setinggi *Sideways Move*.

- *Simulated Annealing*

Algoritma ini bervariasi dari 39.848% hingga 55.32%. Algoritma ini dirancang untuk menghindari jebakan di *local optima* dengan menurunkan probabilitas untuk menerima solusi yang lebih buruk seiring iterasi, menghasilkan variasi akurasi yang lebih luas dibandingkan algoritma *hill-climbing* yang lain.

- *Genetic Algorithm*

Akurasi algoritma mulai dari 17.392% untuk populasi awal yang lebih kecil. Algoritma ini menunjukkan kemampuan evolusi melalui seleksi dan mutasi, yang dapat berkontribusi pada peningkatan akurasi seiring iterasi, walaupun lebih lambat dibandingkan metode-metode pencarian lokal lainnya dalam mencapai *global optima*.

### 3. Bagaimana perbandingan durasi proses pencarian tiap algoritma relatif terhadap algoritma lainnya?

- *Steepest Ascent Hill-climbing*

Durasi yang dihasilkan algoritma dalam beberapa eksperimen dengan durasi sekitar 203.576 detik, 252.106 detik, dan 161.231 detik.

- *Hill-climbing with Sideways Move*

Durasi yang dihasilkan algoritma dalam beberapa eksperimen dengan durasi sekitar 311.32 detik, 253.857 detik, dan 284.246 detik.

- *Random Restart Hill-climbing*

Durasi yang dihasilkan algoritma dalam beberapa eksperimen dengan durasi sekitar 718.266 detik, 532.77 detik, dan 682.264 detik.

- *Stochastic Hill-climbing*

Durasi yang dihasilkan algoritma dalam beberapa eksperimen dengan durasi sekitar 5.087 detik, 2.618 detik, dan 22.725 detik.

- *Simulated Annealing*

Durasi yang dihasilkan algoritma dalam beberapa eksperimen dengan durasi sekitar 7.008 detik, 8.192 detik, dan 2.197 detik.

- *Genetic Algorithm*

Durasi untuk algoritma genetika sangat bervariasi tergantung pada jumlah populasi dan iterasi. Misalnya, durasi berkisar dari 0.047 detik untuk iterasi dan

populasi kecil hingga 153.026 detik untuk skala yang lebih besar. Dibutuhkan waktu komputasi yang tinggi pada skala iterasi/populasi yang besar.

*Stochastic Hill-climbing* dan *Simulated Annealing* cenderung menawarkan durasi yang lebih cepat untuk proses pencarian, sedangkan *Hill-climbing with Sideways Move* dan *Genetic Algorithm* dapat menghasilkan durasi yang lebih lama dalam kondisi tertentu.

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

- *Steepest Ascent Hill-climbing*

Algoritma ini menunjukkan hasil yang cukup konsisten. Eksperimen menghasilkan nilai *objective function* berturut-turut 53.96%, 56.111%, dan 51.037%. Dengan perbedaan hasil yang tidak terlalu jauh, algoritma ini cenderung konsisten dalam mencapai nilai *objective* yang tinggi dalam rentang waktu dan iterasi yang relatif stabil.

- *Hill-climbing with Sideways Move*

Algoritma ini juga menunjukkan konsistensi yang baik dengan hasil akhir berturut-turut 53.96%, 56.111%, dan 50.685%. Walaupun ada sedikit variasi pada hasil akhir eksperimen, algoritma ini masih mempertahankan rentang hasil yang serupa. Namun, prosesnya membutuhkan lebih banyak iterasi dan waktu daripada *Steepest Ascent*, karena adanya *sideways moves*.

- *Random Restart Hill-climbing*

Algoritma ini menunjukkan tingkat konsistensi yang cukup baik, ditunjukkan dengan hasil dari tiga eksperimen yang menunjukkan rentang nilai akurasi yang relatif tidak jauh antara satu sama lain, sebesar 52.705%, 54.316%, dan 54.286%.

- *Stochastic Hill-climbing*

Algoritma ini memberikan hasil akhir yang konsisten namun dengan hasil *objective function* yang lebih rendah dibandingkan dua algoritma sebelumnya. Nilai akhir dari tiga eksperimen adalah 45.056%, 48.356%, dan 46.479%, menunjukkan rentang yang lebih kecil namun tetap konsisten dalam pendekatannya dan hasilnya.

- *Simulated Annealing*

Algoritma *Simulated Annealing* memberikan hasil yang lebih bervariasi dan kurang konsisten dibandingkan algoritma lain. Hasil dari tiga eksperimen adalah 42.014%, 39.848%, dan 55.32%. Variasi yang besar di antara eksperimen menunjukkan bahwa algoritma ini lebih sensitif terhadap kondisi awal atau parameter yang digunakan dalam proses pencarian.

- *Genetic Algorithm*

Untuk *Genetic Algorithm*, hasil eksperimen yang disajikan menunjukkan bahwa algoritma ini menghasilkan nilai *objective function* yang bervariasi tergantung pada jumlah iterasi atau populasi. Hasilnya cenderung rendah pada jumlah iterasi atau populasi kecil dan meningkat seiring bertambahnya iterasi atau ukuran populasi, namun tidak sepenuhnya konsisten pada setiap pengaturan.

Secara keseluruhan, konsistensi dan inkonsistensi hasil setiap algoritma bergantung pada pendekatan pencarinya. Algoritma seperti *Steepest Ascent Hill-climbing* atau *Hill-climbing with Sideways Move* menunjukkan konsistensi yang baik karena selalu mengikuti langkah yang terbaik berdasarkan kondisi saat ini, tetapi cenderung terjebak di dalam *local optima*. Sebaliknya, *Simulated Annealing* atau *Genetic Algorithm* memungkinkan elemen acak atau langkah mundur atau variasi populasi yang meningkatkan kemampuan mereka mendekati global optima meskipun dengan hasil yang lebih bervariasi. Inkonsistensi dalam algoritma-algoritma ini disebabkan oleh adanya parameter-parameter yang sensitif terhadap pengaturan awal dan proses acak, sehingga menghasilkan pencarian yang berbeda setiap kali eksperimen dijalankan.

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

Banyak iterasi dan jumlah populasi sangat berpengaruh terhadap hasil akhir pencarian. Dengan melakukan banyak iterasi, algoritma akan memiliki kesempatan yang lebih besar untuk mencari dan menghasilkan solusi yang lebih optimal. Dengan banyaknya generasi/iterasi, maka akan semakin banyak peluang bagi algoritma untuk melakukan proses seleksi, reproduksi, dan mutasi untuk menghasilkan individu yang lebih baik dalam populasi.

Ukuran populasi yang besar juga tentunya akan menimbulkan hasil yang lebih positif dibandingkan ukuran populasi yang kecil. Hal ini dapat terjadi karena ukuran populasi yang besar akan menyediakan variasi genetika yang lebih beragam sehingga algoritma memiliki lebih banyak pilihan yang dapat digunakan untuk pengoptimalisasi individu (kubus). Ukuran populasi yang besar juga akan mencegah algoritma untuk terjebak di dalam *local optima* dikarenakan variansi genetika yang beragam yang membuat kemungkinan untuk menemukan *global optima* menjadi lebih besar dan kemungkinan untuk terjebak di *local optima* menjadi lebih kecil.

## **BAB III**

### **KESIMPULAN DAN SARAN**

Algoritma *Hill-Climbing with Sideways Move* terbukti merupakan algoritma yang paling efektif dalam mencapai solusi paling optimal untuk masalah ini. Algoritma ini menunjukkan akurasi yang tinggi dan stabil, berkisar antara 53.96% hingga 56.1% dengan variansi kecil dalam hasil akhir, sehingga memberikan konsistensi yang baik di beberapa eksperimen. Meskipun membutuhkan durasi yang lebih lama dibandingkan *Steepest Ascent*, kemampuannya untuk mencapai nilai *objective function* yang optimal secara konsisten membuatnya unggul dibandingkan metode lainnya.

*Genetic Algorithm* menjadi algoritma yang kurang efektif untuk digunakan dalam eksperimen ini. Algoritma ini memiliki potensi yang besar untuk menemukan *global optima* namun untuk mencapai hal tersebut, algoritma ini perlu dimasukkan iterasi dan populasi yang sangat banyak sehingga waktu penggerajannya menjadi sangat lama dan memiliki proses-proses yang cukup lama sehingga kurang efisien untuk digunakan. Akurasinya dengan populasi serta iterasi yang sedikit menghasilkan akurasi yang kecil, yang menunjukkan algoritma ini kurang stabil dan efisien dalam mencapai solusi optimal dalam skala yang kecil dibandingkan algoritma pencarian lainnya.

## **BAB IV**

### **PEMBAGIAN TUGAS**

<b>Nama</b>	<b>NIM</b>	<b>Tugas</b>
Arvyno Pranata Limahardja	18222007	<ol style="list-style-type: none"><li>1. Optimasi algoritma semua varian algoritma Hill-climbing dan Simulated Annealing</li><li>2. Optimasi Objective Function</li><li>3. Pengeraaan Laporan</li></ol>
Bastian Natanael Sibarani	18222053	<ol style="list-style-type: none"><li>1. Optimasi algoritma Genetic Algorithm</li><li>2. Membantu optimasi algoritma Simulated Annealing</li><li>3. Optimasi Objective Function</li><li>4. Pengeraaan Laporan</li></ol>
Naomi Pricilla Agustine	18222065	<ol style="list-style-type: none"><li>1. Desain Website</li><li>2. Pengeraaan Laporan</li></ol>
Micky Valentino	18222093	<ol style="list-style-type: none"><li>1. Pengeraaan Website</li><li>2. Debugging Random Restart Hill Climbing</li></ol>

## REFERENSI

Artificial Intelligence A Modern Approach 4th Edition. (2020). Norvig, Peter. Russell, Stuart. Diakses pada 4 November 2024.

Genetic Algorithms - Parent Selection. (2023). Diakses pada 4 November 2024 dari [https://www.tutorialspoint.com/genetic\\_algorithms/genetic\\_algorithms\\_introduction.html](https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_introduction.html)

Generate pseudo-random numbers. (2017). Diakses pada 6 November 2024 dari <https://python.readthedocs.io/en/stable/library/random.html>

Magic Cube. (2024). Diakses pada 6 November 2024 dari [https://en.m.wikipedia.org/wiki/Magic\\_cube](https://en.m.wikipedia.org/wiki/Magic_cube)

Math - Mathematical functions. (2024). Diakses pada 5 November 2024. <https://docs.python.org/3/library/math.html>

NumPy Documentation. (2022). Diakses pada 5 November 2024 dari <https://numpy.org/doc/>  
Pengantar Panjat Tebing | Kecerdasan Buatan. (2023). Diakses pada 5 November 2024 dari <https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/>

Simulated Annealing. (2024). Diakses pada 6 November 2024 dari <https://www.geeksforgeeks.org/simulated-annealing/>