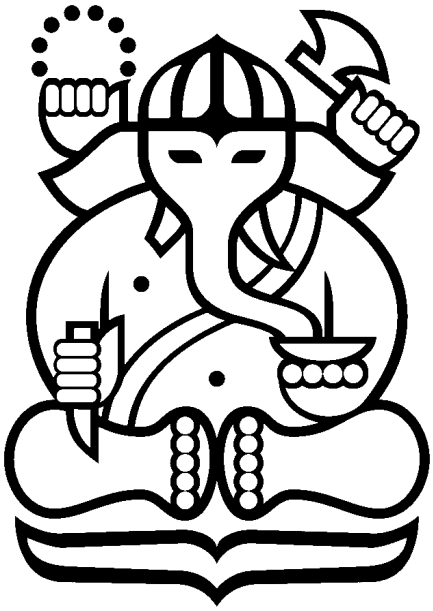
TUGAS BESAR
DASAR INTELEGENSI ARTIFISIAL IF3070
PENCARIAN SOLUSI DIAGONAL MAGIC CUBE DENGAN ALGORITMA LOCAL SEARCH



Disusun oleh:
Kelompok 70

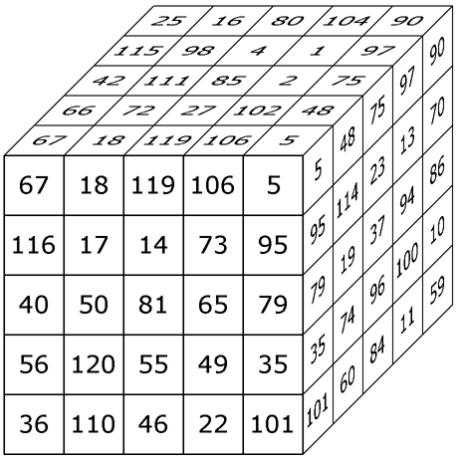
Muhammad Reffy Haykal	18222103
Samuel Franciscus Togar Hasurungan	18222131
Hanan Fitra Salam	18222133
Salsabila Azzahra	18222139

PROGRAM STUDI SISTEM DAN TEKNOLOGI INFORMASI
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

DESKRIPSI PERSOALAN



Magic cube/kubus ajaib adalah padanan 3 dimensi dari persegi ajaib, yaitu kumpulan bilangan bulat yang disusun dalam pola n × n × n sedemikian rupa sehingga jumlah angka-angka pada setiap baris, kolom, pilar dan pada setiap empat diagonal ruang utama adalah sama. Jumlah angka-angka tersebut adalah *magic number*. Di kasus ini kami menangani permasalahan diagonal magic cube berukuran 5x5x5, yang terdiri dari angka 1 hingga n^3 tanpa pengulangan, dengan n sebagai panjang sisi kubus. Kubus ini harus memenuhi beberapa properti, yaitu:

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

Dalam tugas ini, kami diharuskan mengimplementasikan algoritma local search untuk memecahkan Diagonal Magic Cube berukuran 5x5x5, dimana kami akan mengusahakan kubus agar bisa mendekati keadaan/state sempurna, yaitu properti tertentu terkait baris, kolom, tiang, serta diagonal bidang dan ruang kubus mencapai *magic number*.Untuk mencapai *magic number*

ini, setiap algoritma local search akan melakukan iterasi perubahan state dari kubus dengan langkah utama berupa penukaran posisi dua angka secara acak pada setiap iterasi.

Tiga algoritma local search yang akan diimplementasikan adalah salah satu varian dari Hill-Climbing yaitu Steepest-Ascent Hill Climbing, Simulated Annealing, dan Genetic Algorithm. kami diharapkan untuk tidak hanya menjalankan algoritma ini, tetapi juga mengeksplorasi parameter-parameter yang relevan seperti jumlah iterasi, populasi dalam Genetic Algorithm, serta batasan sideways move dalam Hill-Climbing. Tujuannya adalah untuk memahami bagaimana masing-masing algoritma bekerja dalam menemukan solusi optimal (global optima) untuk masalah Diagonal Magic Cube, dengan mengevaluasi efisiensi dan efektivitas setiap pendekatan melalui perbandingan hasil akhir, durasi pencarian, dan analisis terhadap pola stuck di local optima.

Melalui eksperimen ini, kami akan mengumpulkan data berupa nilai objective function, state awal dan akhir kubus, jumlah iterasi, serta durasi pencarian untuk setiap algoritma. Hasil eksperimen tersebut akan dianalisis untuk menilai kedekatan solusi yang dicapai oleh masing-masing algoritma terhadap global optima dan melihat bagaimana iterasi dan populasi dalam Genetic Algorithm mempengaruhi kinerja algoritma.

PEMBAHASAN

1. Pemilihan objective function

objective function adalah fungsi matematis yang digunakan untuk mengevaluasi seberapa baik suatu solusi memenuhi kondisi terbaik dalam masalah optimasi. Dalam permasalahan diagonal magic cube, kondisi terbaik yang bisa diraih adalah jika setiap baris, kolom, tiang, dan diagonal menghasilkan *magic number* jika dijumlahkan.

Magic number dari suatu kubus dapat dicari dengan rumus berikut:

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

Magic number kubus dilambangkan sebagai $M_3(n)$. Karna pada kasus ini kubus yang dipakai adalah kubus 5 sisi (n), maka kita akan mensubtitusi n pada rumus tersebut. Hasil magic number yang didapat pada magic cube sisi 5 ini adalah **315**.

Setelah mengetahui hasil terbaik yang ingin dicapai dari permasalahan *diagonal magic cube* ini, *objective function* dapat dibuat. Kami ingin *objective function* yang kami buat berfungsi sebagai ukuran seberapa jauh kondisi kubus saat ini dari kondisi ideal dengan melihat berapa selisih jumlah angka yang ada saat ini dengan magic number. Oleh karena itu, kami membuat *objective function* dengan menghitung selisih jumlah angka yang ada pada setiap baris, kolom, tiang, dan diagonal kubus dengan magic number, dan mengakumulasi nya. Semakin kecil selisih yang ada, semakin dekat dengan hasil terbaik. Karena menggunakan pendekatan menghitung selisih, hasil terbaik yang kami harapkan dari *objective function* ini adalah 0.

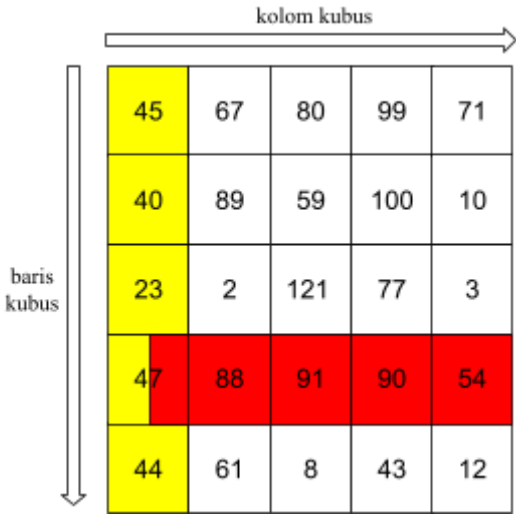
objective function yang kami gunakan untuk kasus ini adalah sebagai berikut:

$$f(c) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(|S_r(i,j) - M| + |S_c(i,j) - M| + |S_d(i,j) - M|) + |S_{D1} - M| + |S_{D2} - M| + |S_{D3} - M| + |S_{D4} - M|)$$

- $S_r(i,j)$ = Jumlah elemen pada baris ke-i dan lapisan ke-j.
- $S_c(i,j)$ = Jumlah elemen pada kolom ke-i dan lapisan ke-j.
- Sd(i,j) = Jumlah elemen pada diagonal dari lapisan ke-i dan lapisan ke-j.
- S_{D1},S_{D2},S_{D3},S_{D4} = Jumlah elemen pada empat diagonal ruang pada kubus.

Dalam rumus tersebut, kami menghitung total keseluruhan dari baris, kolom, tiang, dan diagonal karena syarat pada kubus ajaib baru terpenuhi jika semua elemen tersebut sesuai dengan magic number.

Alasan kami mendefinisikan rumus *objective function* seperti ini adalah karena pada saat angka bertukar, kondisi kubus dapat berubah dan yang terpengaruh bukan hanya satu komponen, melainkan beberapa. Sebagai contoh, kita bisa melihat gambar alas kubus berikut.



Gambar 1: Visualisasi *layer* pada kubus 5 x 5 x 5

Angka “45” dan “47” berada pada satu baris. Namun ketika angka-angka ini bertukar, yang terpengaruh bukan hanya barisnya, melainkan juga kolom, tiang, dan diagonal ruang yang memiliki "45" atau "47" sebagai anggotanya. Kondisi mereka juga ikut berubah saat pertukaran terjadi.

M sebagai magic number adalah kondisi ideal yang ingin diraih kubus. Oleh karena itu, kita bisa menghitung seberapa dekatnya kondisi kubus ke ideal melalui selisih antara magic number dan jumlah angka dari baris, kolom, tiang, dan diagonal ruang kubus. Oleh karena itu, dapat disimpulkan bahwa **solusi terbaik akan ditemukan ketika objective function memiliki nilai yang sama dengan 0**.

$$[\ f(c)\ =\ 0\ ]$$

2. Penjelasan Implementasi Algoritma Local Search

2.1 *Magic Cube*

Class MagicCube digunakan untuk membuat struktur *magic cube* ukuran 5x5x5 serta metode-metode untuk menghitung skor dan menghasilkan tetangga dari suatu kondisi kubus.

Fungsi/ Prosedur	Prosedur *init*
Deskripsi	Inisialisasi awal objek MagicCube. Prosedur ini menerima parameter *self* untuk menunjuk *class*-nya sendiri, *N* untuk menentukan ukuran kubus, dan *magic_number* untuk menetapkan target baris, kolom, nilai total awal kubus. Setelah itu, prosedur ini akan menyimpan ukuran kubus, menentukan target nilai magic number, dan inisialisasi kubus dengan angka acak.
Kode	`def __init__(self, N=5, magic_number=315):` `  self.N = N` `  self.MAGIC_NUMBER = magic_number` `  self.cube = self.initialize_cube()`

Fungsi/ Prosedur	Fungsi *initialize_cube*
Deskripsi	*Function initialize_cube* digunakan untuk menginisialisasi kubus secara acak. Menerima parameter *slef* untuk menunjuk kelasnya sendiri. Fungsi ini akan membuat array dengan angka 1 hingga N^3. Lalu akan mengacak angka dan mengembalikan array yang sudah diacak membentuk N x N x N.
Kode	`def initialize_cube(self):` `  numbers = np.arange(1, self.N**3 + 1)` `  np.random.shuffle(numbers)` `  return numbers.reshape((self.N, self.N, self.N))`

Fungsi/ Prosedur	Fungsi *search*
Deskripsi	Fungsi ini adalah fungsi yang menghitung skor. Perhitungan skor dilakukan dengan *for loop* indeks i pada setiap layer, menghitung perbedaan antara jumlah elemen dalam setiap baris, kolom, dan diagonal

	dengan magic number, lalu menambahkannya pada variabel *score*. Fungsi ini akan mengembalikan skor.
Kode	

```
def fitness(self, cube=None):
    if cube is None:
        cube = self.cube

    cube = np.array(cube)

    if cube.shape != (self.N, self.N, self.N):
        raise ValueError(f"Ukuran kubus adalah {cube.shape},
seharusnya {(self.N, self.N, self.N)}")

    # Menghitung skor dari setiap baris, kolom, dan diagonal
    score = 0

    # Menghitung skor dari diagonal kubus
    score += abs(np.sum([cube[i, i, i] for i in
range(self.N)]) - self.MAGIC_NUMBER)
    score += abs(np.sum([cube[i, i, self.N - i - 1] for i in
range(self.N)]) - self.MAGIC_NUMBER)

    target = self.MAGIC_NUMBER * self.N
    for i in range(self.N):
        # Menghitung skor dari diagonal
        score += abs(np.sum(np.diagonal(cube[i, :, :])) -
self.MAGIC_NUMBER)
        score += abs(np.sum(np.diagonal(np.fliplr(cube[i, :,
:]))) - self.MAGIC_NUMBER)
        score += abs(np.sum(np.diagonal(cube[:, i, :])) -
self.MAGIC_NUMBER)
        score += abs(np.sum(np.diagonal(np.fliplr(cube[:, i,
:]))) - self.MAGIC_NUMBER)
        score += abs(np.sum(np.diagonal(cube[:, :, i])) -
self.MAGIC_NUMBER)
        score += abs(np.sum(np.diagonal(np.fliplr(cube[:, :,
i]))) - self.MAGIC_NUMBER)
```

	`` score += abs(np.sum(cube[i, :, :]) - target) score += abs(np.sum(cube[:, i, :]) - target) score += abs(np.sum(cube[:, :, i]) - target) return score ```

Fungsi/ Prosedur	Fungsi *neigbors_function*
Deskripsi	Digunakan untuk menghasilkan solusi tetangga, yang merupakan konfigurasi kubus di mana dua elemen dipertukarkan. Langkah-langkah utama dalam fungsi ini antara lain: 1. Inisialisasi *position* dengan koordinat x, y ,z. 2. Iterasi tiap elemen untuk memeriksa dan menukar posisi tetangga. 3. Mengembalikan *neighbor* yang ditukar.
Kode	``` def neighbors_function(self, cube=None): if cube is None: cube = self.cube neighbors = [] positions = np.array(list(product(range(self.N), repeat=3))) for x, y, z in positions: for dx, dy, dz in product([-1, 0, 1], repeat=3): if dx == 0 and dy == 0 and dz == 0: continue nx, ny, nz = x + dx, y + dy, z + dz if 0 <= nx < self.N and 0 <= ny < self.N and 0 <= nz < self.N: swap_cube = cube.copy() swap_cube[x, y, z], swap_cube[nx, ny, nz] = swap_cube[nx, ny, nz], swap_cube[x, y, z] neighbors.append(swap_cube) return neighbors ```

2.2 *Genetic Cube*

gencube.py digunakan khusus pada algoritma genetic untuk membuat struktur *magic cube* ukuran 5x5x5 serta metode-metode untuk menghitung *objective function* dan melaksanakan tahap-tahap penting dalam algoritma tersebut, yaitu pemuatan populasi, seleksi, crossover, serta mutasi.

Fungsi/ Prosedur	Fungsi *create_magic_cube*
Deskripsi	Fungsi ini menghasilkan magic cube dalam bentuk *array of list* 1 dimensi dengan bilangan dari angka 1 hingga 125 yang disusun secara acak. List ini nantinya akan diubah menjadi array 3D berukuran 5x5x5
Kode	`def create_magic_cube():` `  return random.sample(range(1, 126), 125)`

Fungsi/ Prosedur	Fungsi *transform_to_3d*
Deskripsi	Fungsi ini mengubah list 1 dimensi berukuran 125 menjadi array 3D 5x5x5 yang dapat digunakan untuk mempermudah akses elemen baris, kolom, atau diagonal dalam magic cube
Kode	`def initialize_cube(self):` `  numbers = np.arange(1, self.N**3 + 1)` `  np.random.shuffle(numbers)` `  return numbers.reshape((self.N, self.N, self.N))`

Fungsi/ Prosedur	Fungsi *objective_function*
Deskripsi	Fungsi ini mengecek tiap baris, kolom, tiang, serta beberapa diagonal di setiap potongan bidang dari kubus dan juga diagonal antar ruang. Setiap perbedaan antara jumlah elemen dengan konstanta MAGIC_CONST yang bernilai 315 dihitung sebagai penalti dalam point. Semakin rendah nilai point, semakin dekat kubus ini menjadi magic cube yang ideal
Kode	`def objective_function(magic_cube):` `  point = 0` `  for k in range(5):`

```
        for j in range(5):
            line_sum_1 = line_sum_2 = line_sum_3 = 0
            for i in range(5):
                line_sum_1 += magic_cube[25 * k + 5 * j + i]
                line_sum_2 += magic_cube[25 * k + 5 * i + j]
                line_sum_3 += magic_cube[25 * j + 5 * i + k]
            point += abs(line_sum_1 - MAGIC_CONST)
            point += abs(line_sum_2 - MAGIC_CONST)
            point += abs(line_sum_3 - MAGIC_CONST)



    for j in range(5):
        line_sum_1 = 0
        line_sum_2 = 0
        line_sum_3 = 0
        line_sum_4 = 0
        line_sum_5 = 0
        line_sum_6 = 0
        for i in range(5):
            mirr = 4 - i
            line_sum_1 += magic_cube[25 * j + 5 * i + i]
            line_sum_2 += magic_cube[25 * j + 5 * i + mirr]
            line_sum_3 += magic_cube[25 * i + 5 * j + i]
            line_sum_4 += magic_cube[25 * i + 5 * j + mirr]
            line_sum_5 += magic_cube[25 * i + 5 * i + j]
            line_sum_6 += magic_cube[25 * i + 5 * mirr + j]

        point += abs(line_sum_1 - MAGIC_CONST)
        point += abs(line_sum_2 - MAGIC_CONST)
        point += abs(line_sum_3 - MAGIC_CONST)
        point += abs(line_sum_4 - MAGIC_CONST)
        point += abs(line_sum_5 - MAGIC_CONST)
        point += abs(line_sum_6 - MAGIC_CONST)


    line_sum_1 = 0
    line_sum_2 = 0
```

	`` line_sum_3 = 0 line_sum_4 = 0 for i in range(5): mirr = 4 - i line_sum_1 += magic_cube[25 * i + 5 * i + i] line_sum_2 += magic_cube[25 * i + 5 * i + mirr] line_sum_3 += magic_cube[25 * mirr + 5 * i + i] line_sum_4 += magic_cube[25 * mirr + 5 * i + mirr] point += abs(line_sum_1 - MAGIC_CONST) point += abs(line_sum_2 - MAGIC_CONST) point += abs(line_sum_3 - MAGIC_CONST) point += abs(line_sum_4 - MAGIC_CONST) return float(point) ```

Fungsi/ Prosedur	Prosedur *mutate*
Deskripsi	Prosedur ini bertugas untuk melaksanakan proses mutasi pada algoritma genetic, yaitu memperkenalkan variasi pada konfigurasi *magic cube* dengan menukar posisi elemen-elemen di dalamnya. Ini bertujuan agar solusi tidak terjebak di *local optimum*, memungkinkan terbukanya ruang baru eksplorasi.
Kode	``` def mutate(cube, mutation_rate=0.05): """Mutasi magic cube dengan menukar beberapa angka secara acak, berdasarkan mutation_rate.""" num_swaps = random.randint(1, 3) for _ in range(num_swaps): if random.random() < mutation_rate: i, j = random.sample(range(125), 2) cube[i], cube[j] = cube[j], cube[i] parent_pos += 1 ```

2.3 *Steepest Ascent Hill-climbing*

Algoritma pencarian ini merupakan variasi dari Hill Climbing yang mempertimbangkan semua gerakan dari kondisi saat ini dan memilih yang terbaik sebagai kondisi berikutnya. Algoritma ini memeriksa semua tetangga dari status saat ini dan memilih satu tetangga yang paling dekat dengan status tujuan.

Class SteepestHillClimbing digunakan untuk mengimplementasikan algoritma ini ke *magic cube* yang telah di-*generate.*

Fungsi/ Prosedur	Prosedur *init*
Deskripsi	Digunakan untuk menjadi konstruktor untuk inisialisasi awal ketika objek *SteepestHillClimbing* dibuat. Prosedur ini memiliki parameter *self* untuk menunjuk *class* diri sendiri dan *max_iteration=1000* untuk membuat batas iterasi. Setelah itu, prosedur ini akan membuat objek *solver* untuk inisialisasi kubus dan fungsinya, *initial_state* untuk menyimpan keadaan awal dari kubus, *objective_function* untuk menghitung skor awal dengan fungsi *fitness, max_iterarions* untuk menentukan batas iterasi, dan *array history* untuk menyimpan perubahan skor tiap iterasi.
Kode	`def __init__(self, max_iterations=1000):` `  self.solver = MagicCube()` `  self.initial_state = self.solver.cube` `  self.objective_function = self.solver.fitness(self.initial_state)` `  self.max_iterations = max_iterations` `  self.history = []`

Fungsi/ Prosedur	Fungsi *search*
Deskripsi	*Function search* digunakan untuk menjalankan algoritma *Steepest Ascent Hill-Climbing*. Fungsi ini mencari solusi terbaik secara bertahap di setiap iterasi dan menyimpan skornya. Langkah-langkah utama dalam fungsi ini antara lain: 1. Iteration_count dan start_time untuk inisiasi iterasi dan waktu. 2. Fungsi *neighbors_function* untuk menghasilkan skor tetangga sekarang. 3. Memilih tetangga terbaik dengan *for loop*. mengevaluasi skor

	dengan *fitness*. 4. Memperbarui *initial_state, objective_function,* dan menyimpan *history* skor. 5. Mengembalikan *initial_state, objective_function, execute_time, history,* dan *iteration_count.*
Kode	

```
def search(self):
    iteration_count = 1
    start_time = time.time()

    # Mencari tetangga terbaik dari kubus
    while iteration_count <= self.max_iterations:
        neighbors =
self.solver.neighbors_function(self.initial_state)
        best_neighbor = min(neighbors,
key=self.solver.fitness)
        best_score = self.solver.fitness(best_neighbor)

        # Mengganti kubus awal dengan kubus tetangga terbaik
        if best_score < self.objective_function:
            self.initial_state = best_score
            self.objective_function = best_score
            self.history.append(self.objective_function)
            print(f"Iterasi {iteration_count}: Skor sekarang =
{self.objective_function}")

            if self.is_solved(self.objective_function):
                break

        else:
            break

        iteration_count += 1

    execute_time = time.time() - start_time
    return self.initial_state, self.objective_function,
execute_time, self.history, iteration_count
```

Fungsi/ Prosedur	Fungsi *is_solved*
Deskripsi	Memeriksa apakah solusi sudah mencapai hasil optimum.
Kode	`def is_solved(self, score):` `    return score == 0`

2.3 *Simulated Annealing*

Algoritma pencarian ini merupakan algoritma pencarian lokal yang akan memilih tetangga secara acak dan dihitung seberapa baik solusi tetangga tersebut dengan menghitung selisih skor heuristik dengan delta E. Jika lebih baik, maka solusi tersebut dapat dipilih menjadi *current state.* Solusi yang lebih buruk mungkin saja dipilih dengan menghitung terlebih dahulu probabilitas solusi tersebut dengan rumus … Setiap pencarian solusi (iterasi) yang dilakukan, suhu akan menurun. Pencarian akan diterminasi jika suhu sudah mendekati 0.

Dalam implementasi penggunaan algoritma simulated annealing, *class SimulatedAnnealing* digunakan untuk mengimplementasikan algoritma ini ke *magic cube* yang telah di-*generate.* Berikut fungsi/prosedur yang digunakan dalam kelas *SimulateAnnealing*.

Fungsi/ Prosedur	Prosedur *init*
Deskripsi	Konstruktor dari kelas *SimulatedAnneling.* Fungsi ini memiliki masukan *self* untuk menunjuk class diri sendiri dan juga memiliki parameter yang diinisialisasi, yaitu: 1. *initial_temperature*=1000 2. *cooling_rate*=0.99 3. temperature_threshold=0.001 4. stagnation_threshold=50 Dalam fungsi ini akan membuat objek *solver* yang merupakan objek dari kelas *MagicCube* yang akan menjalankan fungsi-fungsi dari kelas tersebut, seperti inisialisasi kubus, menghitung skor heuristik, dan memilih tetangga.
Kode	`def __init__(self, initial_temperature=1000,` `cooling_rate=0.99, temperature_threshold=0.001,` `stagnation_threshold=50):` `        self.solver = MagicCube()`

	`` self.current_state = self.solver.cube self.current_score = self.solver.fitness(self.current_state) self.temperature = initial_temperature self.cooling_rate = cooling_rate self.temperature_threshold = temperature_threshold self.stagnation_threshold = stagnation_threshold self.iteration_scores = [] self.probabilities = [] self.total_stuck_count = 0 ```

Fungsi/ Prosedur	Fungsi *search*
Deskripsi	Proses pencarian solusi menggunakan simulated annealing dilakukan dalam fungsi ini. Fungsi ini memiliki masukan *self* dan memiliki keluaran *self.current_state*, kondisi kubus saat ini setiap pencarian solusi dilakukan, dan *self.current_score*, skor heuristik untuk setiap solusi yang dihasilkan. Langkah-langkah utama dalam fungsi search 1. Menjalankan looping dengan kondisi suhu masih lebih besar dari batas *temperature_threshold* 2. Mengambil tetangga secara acak dan menghitung *delta_E* 3. Jika *delta_E* lebih kecil dari 0 (solusi lebih baik dari current state), maka current state dan current score akan diperbarui 4. Jika delta_E lebih besar dari 0,akan dihitung probabilitas penerimaan solusi buruk berdasarkan persamaan *math.exp(-delta_E / temperature)* 5. Jika tidak ada perbaikan selama beberapa iterasi berturut-turut, algoritma mencatat bahwa ia *stuck* pada *local optima*. 6. Setelah setiap iterasi, suhu dikurangi sesuai *cooling_rate*. Loop akan berhenti jika solusi optimal ditemukan atau suhu mencapai batas.
Kode	``` def search(self): iteration_count = 1 stagnation_counter = 0 start_time = time.time() ```

```
        while self.temperature > self.temperature_threshold:
            neighbors =
self.solver.neighbors_function(self.current_state)
            next_state = random.choice(neighbors)
            next_score = self.solver.fitness(next_state)

            delta_E = next_score - self.current_score
            probability = 1.0

            if delta_E < 0:
                self.current_state = next_state
                self.current_score = next_score
                stagnation_counter = 0
            else:
                probability = math.exp(-delta_E /
self.temperature)
                if random.random() < probability:
                    self.current_state = next_state
                    self.current_score = next_score
                stagnation_counter += 1

            self.iteration_scores.append(self.current_score)
            self.probabilities.append(probability)

            if stagnation_counter >=
self.stagnation_threshold:
                self.total_stuck_count += 1
                print(f"Stuck di local optima pada iterasi
{iteration_count} dengan skor {self.current_score}")

            print(f"Iterasi {iteration_count}: Skor Heuristik
= {self.current_score}, Temperatur = {self.temperature:.4f},
Probabilitas = {probability:.4f}")

            if self.is_solved(self.current_score):
                break

            self.temperature *= self.cooling_rate
```

	`` iteration_count += 1 end_time = time.time() print(f"Waktu yang dibutuhkan: {end_time - start_time} detik") print(f"Total frekuensi stuck di local optima: {self.total_stuck_count}") return self.current_state, self.current_score ```

Fungsi/ Prosedur	Fungsi *is_solved*
Deskripsi	Mengevaluasi apakah solusi sudah optimal dengan mengembalikan nilai True jika skor heuristik yang didapat sama dengan 0. Jika True, maka fungsi *search* akan berhenti.
Kode	``` def is_solved(self, score): return score == 0 ```

Fungsi/ Prosedur	Prosedur *plot_objective_function*
Deskripsi	Melakukan visualisasi selama proses pencarian solusi yang dilakukan. Hasil dari visualisasi ini adalah grafik objective function solusi yang dihasilkan (sumbu y) untuk setiap iterasi (sumbu x).
Kode	``` def plot_objective_function(self): plt.figure(figsize=(10, 5)) # Plotting Objective Function Value over Iterations plt.subplot(2, 1, 1) plt.plot(self.iteration_scores, label='Objective Function (Score)') plt.xlabel("Iteration") plt.ylabel("Objective Function Value (Score)") plt.title("Objective Function Value over Iterations in Simulated Annealing") plt.legend() ```

	`` plt.tight_layout() plt.show() ```

Fungsi/ Prosedur	Prosedur *plot_probability*
Deskripsi	Melakukan visualisasi selama proses pencarian solusi yang dilakukan. Hasilnya dari visualisasi ini adalah grafik nilai probabilitas solusi untuk setiap iterasi.
Kode	``` def plot_probability(self): plt.figure(figsize=(10, 5)) # Plotting Probability e^(-delta_E / T) over Iterations plt.subplot(2, 1, 2) plt.plot(self.probabilities, label='Probability (e^(-delta_E / T))', color='orange') plt.xlabel("Iteration") plt.ylabel("Probability") plt.title("Probability of Accepting Worse Solution over Iterations") plt.legend() plt.tight_layout() plt.show() ```

2.4 *Genetic Algorithm*

Algoritma ini dimulai dengan membentuk populasi awal yang terdiri dari banyak kubus yang di*generate* dengan angka acak dari 1 sampai 125. Pada kali ini, populasi awal saya tentukan jumlahnya menjadi 2000 kubus. Dalam populasi ini dipilih 20 kandidat secara acak, dan dari kandidat dipilih dua parent terbaik untuk menjadi kandidat dalam proses *crossover*, di mana karakteristik dari kedua parent digabungkan untuk membentuk dua anak baru. Setelah anak baru terbentuk, proses mutasi akan diterapkan ke kedua anak tersebut untuk memperkenalkan variasi kecil yang tujuannya untuk mencegah solusi

stagnan pada nilai yang kurang optimal. Setelah itu, anak-anak yang telah melalui proses mutasi ditambahkan ke populasi baru. Proses akan terulang kembali sampai populasi baru sudah mencapai kuantitas yang sama dengan populasi awal. Jika sudah sama, populasi lama digantikan oleh populasi baru, dan populasi baru ini akan digunakan ke generasi berikutnya.

Pada akhir setiap generasi, algoritma mengevaluasi solusi terbaik saat ini berdasarkan nilai objektifnya, dan jika solusi terbaik mendekati atau mencapai nilai target, proses akan dihentikan lebih awal. Proses ini terus diulang hingga mencapai jumlah generasi yang ditentukan atau hingga menemukan solusi optimal. Selama proses ini, nilai objektif terbaik dari setiap generasi dicatat untuk melihat perkembangan. Setelah proses selesai, kode menampilkan konfigurasi terbaik dari cube yang ditemukan, persentase kesempurnaannya terhadap target, dan grafik regresi perkembangan nilai objektif dari setiap generasi.

Implementasi dari algoritma ini dilaksanakan pada kode *genetic.py*

Fungsi/ Prosedur	Fungsi *select_parent*
Deskripsi	Fungsi ini bertugas untuk mengambil 20 kandidat secara acak dari 2000 populasi awal, dan memilih satu *parent* terbaik dari kandidat untuk *crossover*. Parent yang terpilih adalah kandidat terbaik, yaitu individu dengan nilai objektif paling dekat dengan target
Kode	`def select_parent(population):` `    candidates = random.sample(population, 20)` `    candidates.sort(key=lambda cube:` `abs(objective_function(cube) - TARGET_OBJECTIVE))` `    return candidates[0]`

Fungsi/ Prosedur	Prosedur *crossover*
Deskripsi	Prosedur ini bertugas untuk mengeksekusi persilangan antar parent, dan menghasilkan 2 anak baru dari proses tersebut. Di proses ini, segmen tertentu dari parent pertama digabungkan ke child pertama, dan segmen dari parent kedua ke *child* kedua, lalu sisanya dilengkapi dengan elemen-elemen dari parent yang lain tanpa duplikasi.
Kode	`def crossover(parent1, parent2):`

	`` size = len(parent1) child1, child2 = [None]*size, [None]*size start, end = sorted(random.sample(range(size), 2)) # copy segment dari parent1 ke child1, terus parent2 ke child2 child1[start:end+1] = parent1[start:end+1] child2[start:end+1] = parent2[start:end+1] # ngisi posisi yang masih kosong di child1 dari parent2 and vice versa tanpa ada duplikat fill_remaining(child1, parent2, start, end) fill_remaining(child2, parent1, start, end) return child1, child2 ```

Fungsi/ Prosedur	Prosedur *fill_remaining*
Deskripsi	Fungsi ini bertugas untuk mengisi bagian-bagian kosong dalam *child* dengan elemen-elemen dari *parent* yang lain, tanpa mengulangi elemen yang sudah ada.
Kode	``` def fill_remaining(child, parent, start, end): """Fills remaining positions in child array.""" size = len(parent) child_pos = end + 1 parent_pos = end + 1 while None in child: if parent[parent_pos % size] not in child: child[child_pos % size] = parent[parent_pos % size] child_pos += 1 parent_pos += 1 ```

Fungsi/ Prosedur	Prosedur *genetic_algorithm*
Deskripsi	Prosedur ini merupakan fungsi utama yang akan mengimplementasikan seluruh proses algoritma genetika untuk menemukan *state* magic cube yang terbaik. Pada fungsi ini, pengguna pertama-tama menentukan jumlah

	generasi (iterasi) dan ukuran populasi awal. Program memulai dengan memilih salah satu *state* acak dari populasi sebagai titik awal. Di setiap generasi, proses seleksi memilih dua *parent* terbaik, yang kemudian menjalani *crossover* untuk menghasilkan dua anak. Anak-anak ini mengalami mutasi untuk menjaga keragaman dalam populasi. Setelahnya, populasi kemudian diperbarui dengan memasukkan anak-anak baru tersebut. Best cube dan nilai objektif terbaik diperbarui setiap kali ditemukan konfigurasi dengan nilai objektif yang lebih baik. Jika konfigurasi memenuhi TARGET_OBJECTIVE yaitu 0, program akan berhenti. Terakhir, prosedur ini akan menampilkan plot regresi yang bertujuan untuk menunjukkan hubungan antara nilai maksimum dan rata-rata objective function dengan setiap generasi atau iterasi.
Kode	```def genetic_algorithm(): """Menjalankan algoritma genetika untuk menemukan magic cube terbaik.""" iterations = int(input("Berikan jumlah iterasi/generasi: ")) populations = int(input("Berikan jumlah populasi yang diinginkan: ")) start_time = time.time() population = [create_magic_cube() for _ in range(populations)] best_cube = None best_objective_value = float('inf') max_objective_value_reg = [] avg_objective_value_reg = [] time.sleep(1) print() print("State kubus awal:") rand_pop = random.sample(population,1) print(transform_to_3d(rand_pop)) print() if (iterations <= 0):```

```
        print("jumlah iterasi tidak boleh 0 atau kurang!
jalankan ulang program")
    else:
        print("mengonfigurasi kubus.....")
        time.sleep(2)

        for generation in range(iterations):
            print(f"Generasi: {generation+1}")

            new_population = []

            while len(new_population) < populations:

                parent1 = select_parent(population)
                parent2 = select_parent(population)

                child1, child2 = crossover(parent1, parent2)

                mutate(child1)
                mutate(child2)

                new_population.append(child1)
                new_population.append(child2)



            population = new_population

            current_best = min(population, key=lambda cube:
abs(objective_function(cube) - TARGET_OBJECTIVE))
            current_best_value =
objective_function(current_best)
            average_objective_value =
mean([objective_function(cube) for cube in population])

            if abs(current_best_value - TARGET_OBJECTIVE) <
abs(best_objective_value - TARGET_OBJECTIVE):
                best_cube = current_best
```

```
                best_objective_value = current_best_value




            average_objective_value =
mean([objective_function(cube) for cube in population])


max_objective_value_reg.append(best_objective_value)

avg_objective_value_reg.append(average_objective_value)
            print(f"Nilai objektif terbaik saat ini:
{best_objective_value}")
            print(f"Nilai Objektif rata-rata:
{average_objective_value}")
            print()


            if best_objective_value == TARGET_OBJECTIVE:
                print("Solusi optimal ditemukan!")
                break

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


        print("Selesai!")
        time.sleep(1)
        print(f"Durasi pencarian: {duration:.2f} detik")
        print("\nState Kubus akhir (terbaik):")
        print(transform_to_3d(best_cube))
        time.sleep(2)
        print()

        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

        ax1.plot(max_objective_value_reg, marker='o')
        ax1.set_title("Max Objective Function Progression")
```

	`` ax1.set_xlabel("Generation") ax1.set_ylabel("Max Objective Function") ax2.plot(avg_objective_value_reg, marker='o') ax2.set_title("Average Objective Function Progression") ax2.set_xlabel("Generation") ax2.set_ylabel("Average Objective Function") plt.subplots_adjust(wspace=1) plt.tight_layout() plt.show() ```
mutate	``` def mutate(cube, mutation_rate=0.05): """Mutasi magic cube dengan menukar beberapa angka secara acak, berdasarkan mutation_rate.""" num_swaps = random.randint(1, 3) for _ in range(num_swaps): if random.random() < mutation_rate: i, j = random.sample(range(125), 2) cube[i], cube[j] = cube[j], cube[i] ```

fungsi/prosedur	Deskripsi
select_parent	Fungsi ini bertugas untuk mengambil 20 kandidat secara acak dari 2000 populasi awal, dan memilih satu *parent* terbaik dari kandidat untuk *crossover*. Parent yang terpilih adalah kandidat terbaik, yaitu individu dengan nilai objektif paling dekat dengan target
crossover	Fungsi ini bertugas untuk mengeksekusi persilangan antar parent, dan menghasilkan 2 anak baru dari proses tersebut. Di proses ini, segmen tertentu dari parent pertama digabungkan ke child pertama, dan segmen dari parent kedua ke *child* kedua, lalu sisanya dilengkapi dengan elemen-elemen dari parent yang lain tanpa duplikasi.
fill_remaining	Fungsi ini bertugas untuk mengisi bagian-bagian kosong

	dalam *child* dengan elemen-elemen dari *parent* yang lain, tanpa mengulangi elemen yang sudah ada.
genetic_algorithm	Fungsi utama ini mengimplementasikan seluruh proses algoritma genetika untuk menemukan *state* magic cube yang terbaik
mutate	Fungsi ini bertugas untuk memperkenalkan variasi pada konfigurasi *magic cube* dengan menukar posisi elemen-elemen di dalamnya. Ini bertujuan agar solusi tidak terjebak di *local optimum*, memungkinkan terbukanya ruang baru eksplorasi.

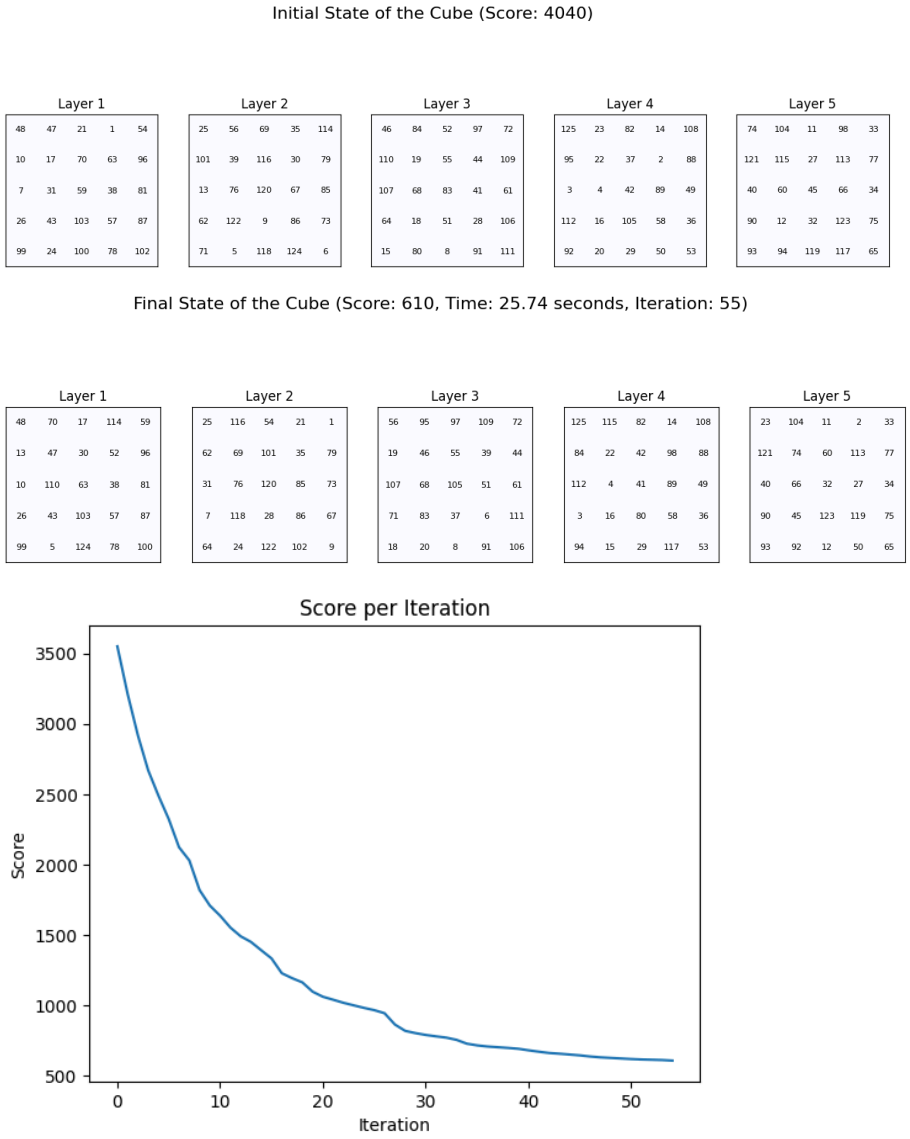
3. Hasil Eksperimen dan Analisis

Dari algoritma pencarian lokal yang telah dikerjakan, kami akan melakukan eksperimen dengan menjalankan algoritma sebanyak 3 kali

3.1 Steepest Ascent Hill-Climbing

Berikut adalah tiga kali percobaan pada algoritma *Steepest Ascent Hill-Climbing*.

a. Percobaan pertama



b. Percobaan kedua

Initial State of the Cube (Score: 4111)

Layer 1				
3	69	75	1	2
17	118	122	54	125
63	20	116	72	24
4	114	86	22	96
95	90	101	27	10

Layer 2				
87	106	85	66	73
50	31	94	40	13
68	65	92	119	46
88	123	51	47	98
37	33	42	7	109

Layer 3				
81	43	113	57	79
121	107	52	58	18
16	39	117	76	99
60	56	44	112	108
6	41	97	71	53

Layer 4				
115	8	12	77	34
14	28	89	45	19
59	5	30	93	70
35	29	105	32	64
103	49	25	82	111

Layer 5				
78	124	120	9	11
100	55	61	15	36
102	104	83	48	91
23	84	38	67	21
80	62	110	74	26

Final State of the Cube (Score: 476, Time: 20.05 seconds, Iteration: 43)

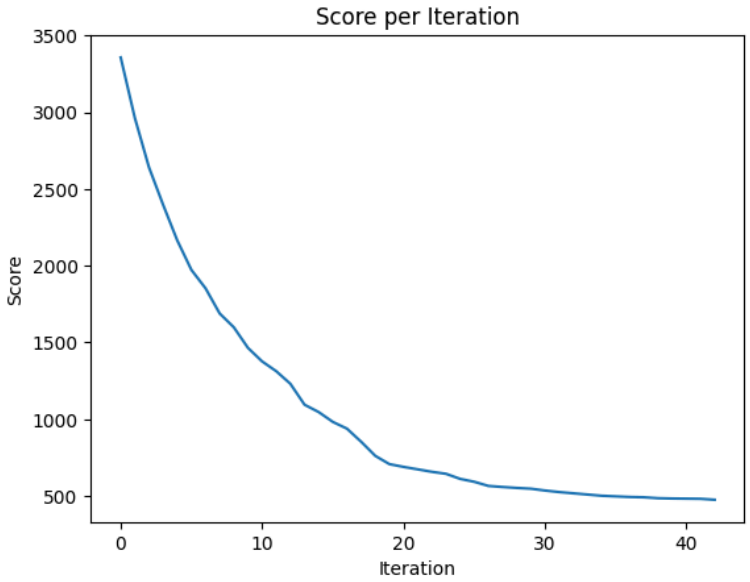
Layer 1				
17	65	75	1	54
3	118	122	2	125
63	20	72	101	24
4	114	90	98	96
95	86	27	92	10

Layer 2				
87	106	85	66	73
50	69	94	45	13
68	31	22	119	46
37	123	7	40	116
41	33	42	53	109

Layer 3				
81	43	113	57	79
121	107	52	76	19
88	14	44	58	99
56	60	47	32	64
6	39	97	71	51

Layer 4				
115	12	28	11	34
16	8	124	117	18
84	29	5	93	91
59	105	25	112	111
80	49	30	110	108

Layer 5				
78	89	120	9	77
102	55	61	15	36
100	104	83	48	70
23	35	38	67	21
103	62	74	82	26



c. Percobaan ketiga

Initial State of the Cube (Score: 3818)

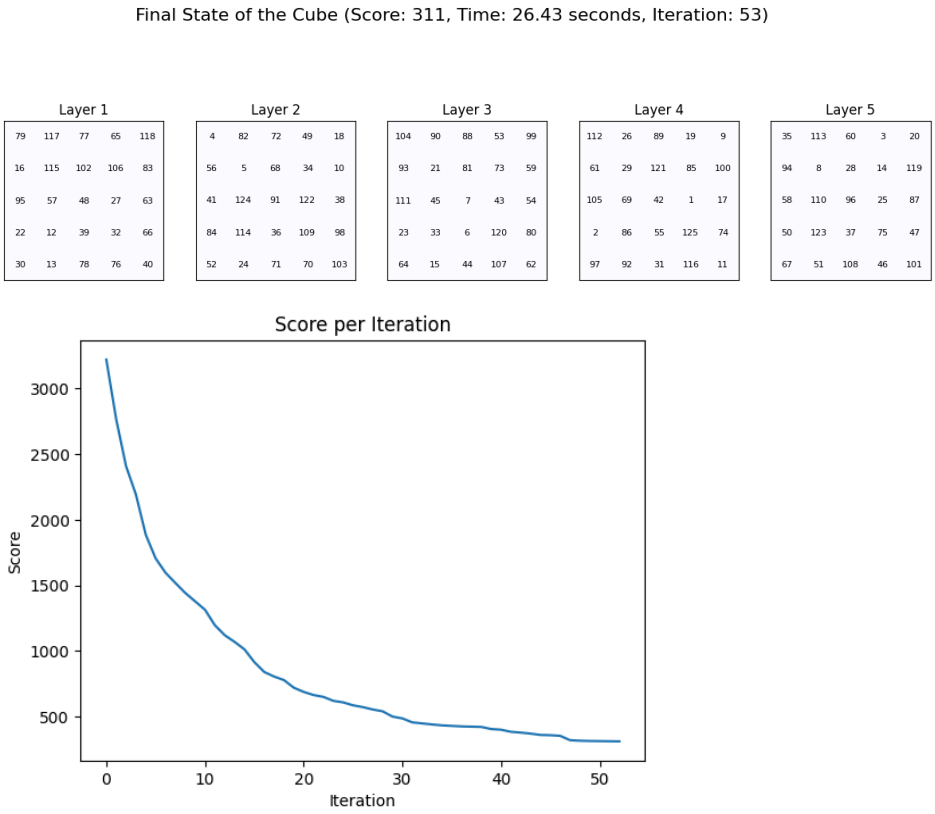
Layer 1				
82	77	106	102	118
16	115	124	117	83
95	41	48	39	63
84	13	27	40	98
22	52	71	76	32

Layer 2				
5	79	73	49	99
57	4	68	72	53
56	65	85	122	38
12	114	78	109	103
30	33	36	70	66

Layer 3				
112	90	88	10	34
42	93	18	59	91
111	21	7	43	54
24	15	92	120	80
64	23	6	107	116

Layer 4				
104	35	19	3	9
61	29	14	119	100
105	8	51	1	17
2	86	44	125	75
97	55	31	62	11

Layer 5				
26	113	60	89	20
110	69	28	121	81
58	94	123	47	87
50	96	45	74	25
67	37	108	46	101



Setelah melakukan tiga kali eksperimen algoritma Steepest Ascent Hill-Climbing, beberapa analisis dapat dilakukan untuk mengevaluasi kinerja dari algoritma ini. Hal pertama yang dapat dianalisa dari algoritma ini adalah seberapa dekat algoritma ini mendekati global optimum (skor *objective function* = 0). Didapat dari eksperimen, bahwa skor akhir yang didapatkan masih jauh dari global optimum. Hal ini disebabkan karena algoritma ini hanya mencari solusi hanya memilih solusi atau langkah terbaik dari solusi saat ini saja tanpa mempertimbangkan kemungkinan solusi yang lebih baik. Selain itu, kita ketahui juga bahwa algoritma ini sangat rentan terjebak pada lokal optimum saja.

Setelah itu, dapat dianalisis hasil akhir, durasi, dan iterasi pencarian dari tiap eksperimen. Berdasarkan hasil akhir, masih terdapat variasi hasil yang hasilnya juga tidak selalu konsisten. Variasi ini disebabkan karena proses pencarian dimulai dari kondisi yang berbeda-beda. Berdasarkan durasi proses pencarian, waktu pencariannya bervariasi antara 20 hingga 27 detik dimana tidak ada perbedaan yang terlalu lama. Iterasi proses juga bervariasi dari 40 hingga 55 iterasi. Dapat disimpulkan dari analisis tiga aspek tersebut bahwa algoritma ini sangat bergantung dengan *state* awal kubus dan banyaknya iterasi yang dilakukan. Iterasi yang lebih banyak memungkinkan algoritma ini mencapai solusi yang lebih baik, namun waktu eksekusi kemungkinan juga akan lebih lama.

Jika dibandingkan dengan algoritma pencarian lokal lain, algoritma Steepest Ascent Hill-Climbing memiliki durasi penyelesaian yang lebih cepat.

Tetapi jika dibandingkan dengan hasil akhir pencarian, algoritma ini memiliki solusi paling buruk. Hal ini disebabkan karena algoritma ini lebih memiliki kecenderungan terjebak di lokal optimum dibandingkan algoritma lain.

3.2 Simulated Annealing

Berikut adalah hasil dari tiga kali percobaan algoritma simulated annealing.

a. Percobaan pertama

Initial State of the Cube (Score: 3858)

Layer 1				
83	27	39	43	71
117	37	120	124	77
25	75	56	46	64
34	101	44	125	68
114	100	4	21	108

Layer 2				
5	13	104	118	103
33	79	57	24	29
78	92	115	81	42
65	93	53	2	35
30	67	54	102	32

Layer 3				
23	6	87	94	31
73	90	86	106	12
16	61	66	84	74
62	36	14	112	58
99	48	22	97	38

Layer 4				
50	109	72	80	121
91	40	119	70	85
76	47	82	63	105
9	7	116	95	28
96	111	122	49	51

Layer 5				
19	88	15	98	17
55	52	1	110	41
69	18	60	3	26
10	123	113	20	107
89	11	59	8	45

Final State of the Cube (Score: 578, Time: 8.49s, Total Stuck: 45)

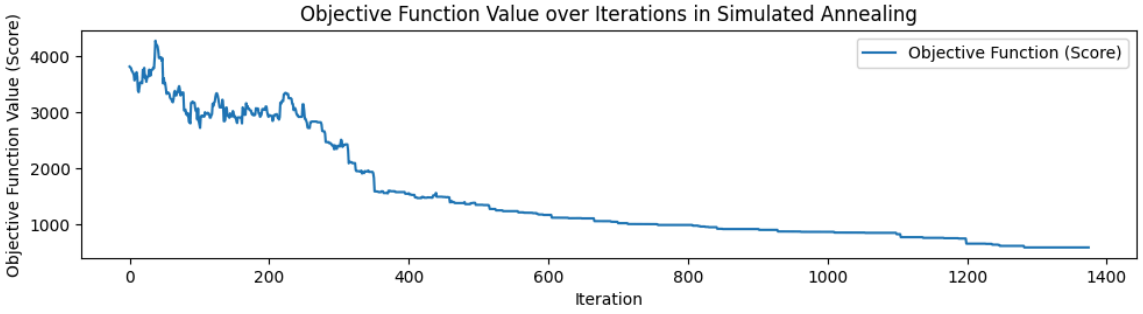
Layer 1				
51	19	39	120	66
27	43	4	124	24
83	13	28	46	84
75	34	108	68	93
114	33	97	54	125

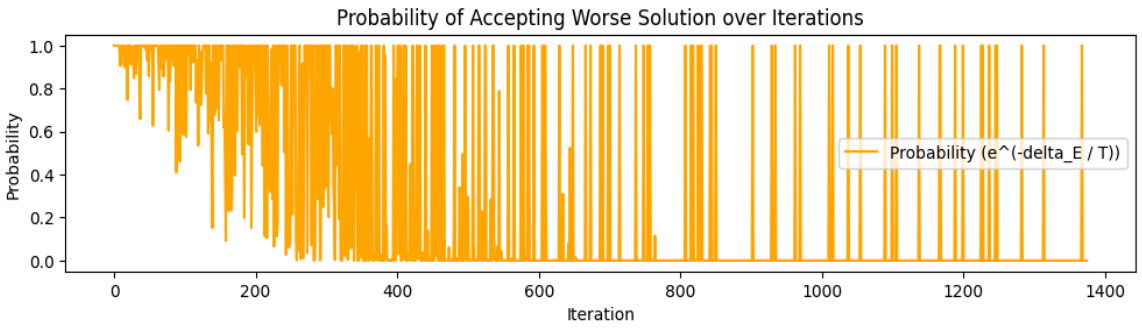
Layer 2				
23	100	121	77	29
60	116	12	104	112
92	122	86	64	87
2	47	78	22	10
99	38	9	21	32

Layer 3				
5	6	37	71	50
44	101	57	103	110
96	72	59	14	35
117	61	76	73	82
53	25	48	102	74

Layer 4				
115	81	80	62	17
55	98	15	85	90
89	106	16	18	67
30	70	119	11	42
111	26	105	45	20

Layer 5				
52	88	79	118	69
36	113	3	1	41
7	56	94	109	31
49	63	123	8	107
91	40	95	58	65





b. Percobaan kedua

Initial State of the Cube (Score: 4129)

Layer 1				
112	99	6	84	104
62	26	113	21	3
20	121	23	12	59
19	95	58	96	25
85	77	36	11	50

Layer 2				
72	82	92	80	47
117	18	10	7	39
107	105	63	111	5
94	73	97	17	34
53	22	86	119	108

Layer 3				
116	109	38	93	31
115	79	2	114	106
69	102	51	91	16
42	122	1	68	27
90	15	13	30	83

Layer 4				
103	40	28	88	98
60	61	32	44	29
110	67	87	57	54
46	45	76	8	125
120	9	100	74	52

Layer 5				
124	48	75	55	70
64	65	14	123	101
24	43	41	49	71
66	37	118	81	56
35	89	4	78	33

Final State of the Cube (Score: 620, Time: 8.67s, Total Stuck: 109)

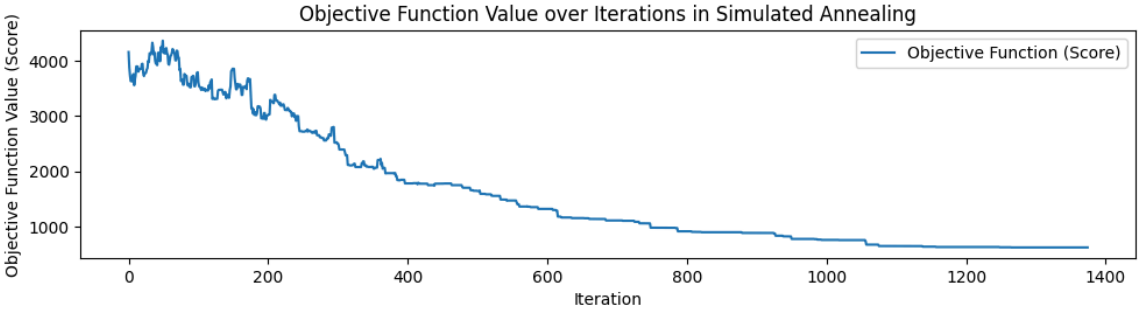
Layer 1				
105	18	111	91	63
95	2	93	3	59
26	113	77	16	32
19	82	34	29	50
92	53	96	102	108

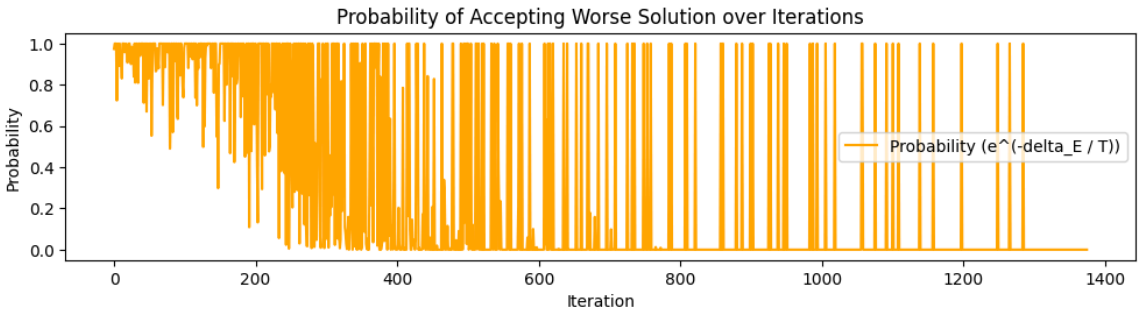
Layer 2				
62	1	7	58	104
30	72	56	118	6
115	117	44	51	109
85	64	90	52	21
17	88	22	122	73

Layer 3				
5	116	86	38	39
84	103	37	106	121
46	107	10	20	80
9	67	125	97	12
110	11	33	68	45

Layer 4				
112	98	23	99	101
60	40	57	61	49
42	87	66	119	70
94	36	114	13	74
35	25	15	4	83

Layer 5				
31	55	48	75	28
71	41	123	43	54
120	24	65	8	14
47	124	76	79	78
69	89	27	81	100





c. Percobaan ketiga

Initial State of the Cube (Score: 3430)

Layer 1				
88	77	103	9	123
66	35	18	93	92
6	41	125	49	117
75	44	120	121	21
97	89	45	64	22

Layer 2				
112	91	109	7	85
2	24	11	15	86
32	42	29	70	118
55	100	115	8	16
80	124	46	5	84

Layer 3				
61	19	57	62	38
71	37	12	105	74
34	26	69	102	104
23	58	27	116	113
14	65	119	101	122

Layer 4				
99	63	79	73	10
48	67	53	51	76
28	3	50	87	31
20	60	83	114	94
56	78	36	1	54

Layer 5				
30	95	40	72	4
90	81	111	52	47
110	82	13	43	96
108	25	17	59	68
33	98	107	106	39

Final State of the Cube (Score: 785, Time: 8.30s, Total Stuck: 103)

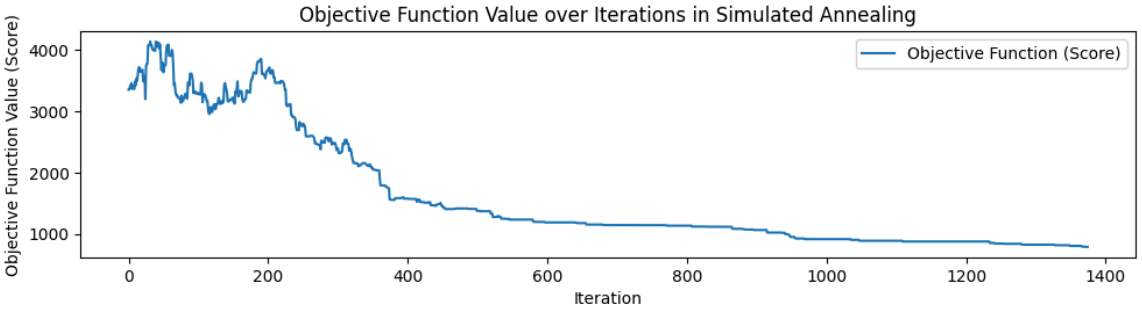
Layer 1				
77	88	26	7	15
8	112	11	119	90
97	103	39	80	50
42	94	32	72	82
21	81	100	102	22

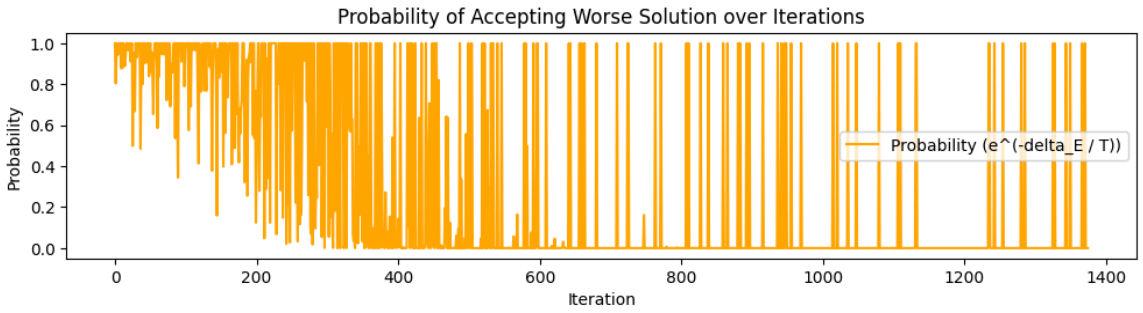
Layer 2				
109	79	105	9	71
2	104	84	73	120
61	23	46	92	4
89	111	40	55	118
24	67	6	85	5

Layer 3				
115	17	117	53	70
3	76	114	47	69
64	18	59	86	54
75	20	57	36	58
124	34	41	121	43

Layer 4				
51	83	19	12	113
110	13	16	10	68
96	29	87	27	123
65	45	116	49	52
56	35	91	107	101

Layer 5				
30	99	95	62	38
28	48	25	63	74
37	125	98	31	44
78	14	122	66	1
108	60	33	106	93





Setelah melakukan tiga kali percobaan dengan algoritma Simulated Annealing, beberapa analisis dapat dilakukan untuk mengevaluasi kinerja dari algoritma ini. Hal pertama yang dapat dianalisa adalah seberapa dekat algoritma ini mendekati global optimum (nilai *objective function* = 0). Dari hasil percobaan, kita dapat melihat nilai akhir yang didapatkan masih jauh dari global optimum. Hal ini bisa disebabkan salah satunya adalah karena ruang pencarian kurang banyak dan juga tidak jarangnya *stuck* di lokal optimum atau lokal optima.

Setelah itu, dapat dianalisis juga, nilai *objective function*, waktu, dan total *stuck* di lokal optima. Berdasarkan hasil akhir, nilai *objective function* masih sangat bervariasi paling sering berkisaran di 500-1000. Tetapi untuk waktu yang diperlukan algoritma dalam melakukan pencarian tidak terlalu lama walau dengan iterasi yang sangat banyak, yaitu berkisaran antara 8-10 detik. Pada visualisasi yang kedua, yaitu probabilitas menerima solusi yang lebih buruk untuk setiap iterasi dapat ditarik kesimpulan bahwa semakin banyak iterasinya atau semakin suhu awal mendekati sama dengan 0, maka algoritma akan lebih jarang untuk mencoba kemungkinan nilai *objective function* baru yang lebih besar (menjauhi 0) dari pada nilai *objective function* lama yang lebih bagus. Hal ini bisa disebabkan karena salah satunya adalah karena fungsi perhitungan nilai *objective function* tidak memberikan cukup variasi nilai, hal ini bisa menyebabkan hasil perhitungan $e^{\frac{\Delta E}{T}}$ sering bernilai 0, sehingga ini juga bisa menyebabkan penerimaan hasil solusi yang lebih buruk tetap tinggi.

3.3 Algoritma Genetic

Sesuai dengan spesifikasi yang diminta sebelumnya, saya membuat percobaan dengan 2 parameter yang dapat diubah yaitu jumlah populasi dan banyak iterasi. Berikut ini adalah variasinya:

- Banyak iterasi:3, 15, dan 125.
- Jumlah populasi: 50, 500, 2000

Dari variasi ini, dihasilkan sebanyak 6 konfigurasi dengan total sebanyak 18 percobaan yang terdiri atas:

1. Jumlah populasi kontrol, banyak iterasi 3 (3 kali)
2. Jumlah populasi kontrol, banyak iterasi 15 (3 kali)

3. Jumlah populasi kontrol, banyak iterasi 125 (3 kali)
4. Banyak iterasi kontrol, jumlah populasi 50 (3 kali)
5. Banyak iterasi kontrol, jumlah populasi 500 (3 kali)
6. Banyak iterasi kontrol, jumlah populasi 2000 (3 kali)

Untuk informasi tambahan, jumlah populasi kontrol yang saya gunakan adalah 1000 populasi, dan banyak iterasi kontrol yang saya gunakan adalah 50 kali.

1. Jumlah populasi kontrol (1000), banyak iterasi 3 (3 kali)
   1.1. Percobaan pertama
      1.1.1. State awal

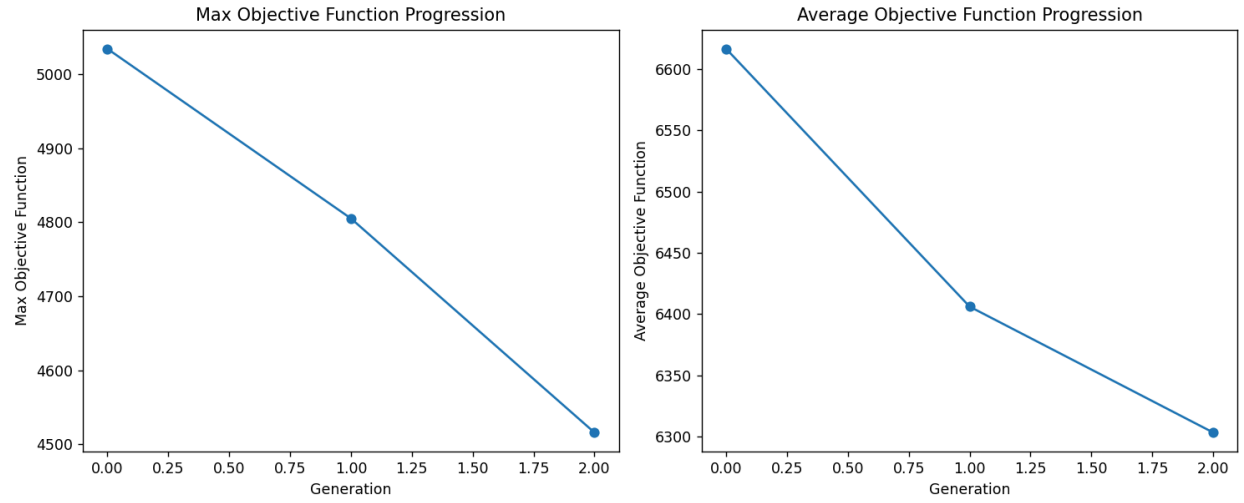
```
State kubus awal:
[[[ 19  95  44  30 109]
  [ 88  46  90  45  55]
  [ 97  83  49 114  25]
  [108 113  37 111  78]
  [ 70   6  35 102  59]]

 [[  8  85  79 118  82]
  [ 18  52  33  96 101]
  [123 117  32  69 120]
  [110  93 105  47  16]
  [ 94  60  84  72  40]]

 [[ 10  38  48  67 107]
  [  5  41 124  71  51]
  [  1  63  66  12  22]
  [ 76  20  50  23  77]
  [  7  62  31   3 122]]

 [[ 65  98  21  73 106]
  [  2  29  36  91  99]
  [ 75  74  81  64  58]
  [ 43  61  68  42  28]
  [ 24  34  87  27 112]]

 [[104 119   9 125  26]
  [ 80  53  11  15  54]
  [121  17 115  89  56]
  [ 86  57 103   4  14]
  [ 13  92  39 100 116]]]
```

      1.1.2. State akhir

```
State Kubus akhir (terbaik):
[[[  8  49  65  56  80]
  [ 26  89  13  50  67]
  [101  19  59 100  27]
  [ 68  84  81  20  52]
  [120  87  73  25  51]]

 [[ 99 123  31  28  36]
  [108  58   9  64  98]
  [ 30   4 109  97  94]
  [ 71  79  43   3   7]
  [ 21  60 124  91  16]]

 [[ 57  75  46 122  61]
  [106  55  62  39  37]
  [ 86  17 121  76  70]
  [ 29 114 116  66   6]
  [104  12  44  95 107]]

 [[ 33 125  14 111  93]
  [ 10  77  92  24  35]
  [115  23  45   5 117]
  [ 74  48  11  72  96]
  [118  85 103  38  82]]

 [[ 41  54  69 119  18]
  [ 34 105  90   2  63]
  [102  40   1 110  83]
  [113  42  47  88  22]
  [ 15 112  78  32  53]]]
```

1.1.3. Nilai objective function akhir yang dicapai = 4516.0

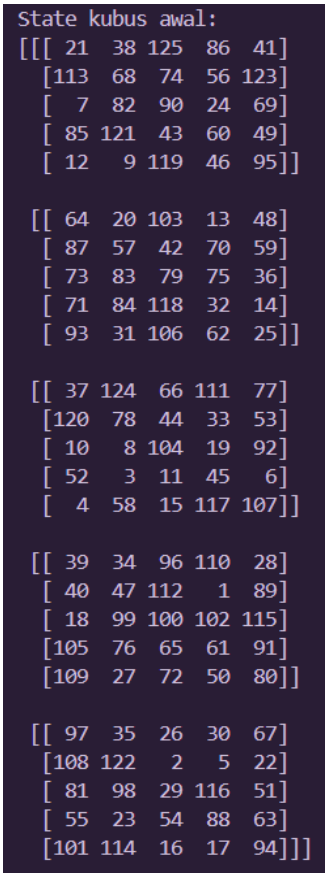
1.1.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



1.1.5. Durasi proses pencarian = 5.85 detik

1.2. Percobaan kedua

1.2.1. State awal

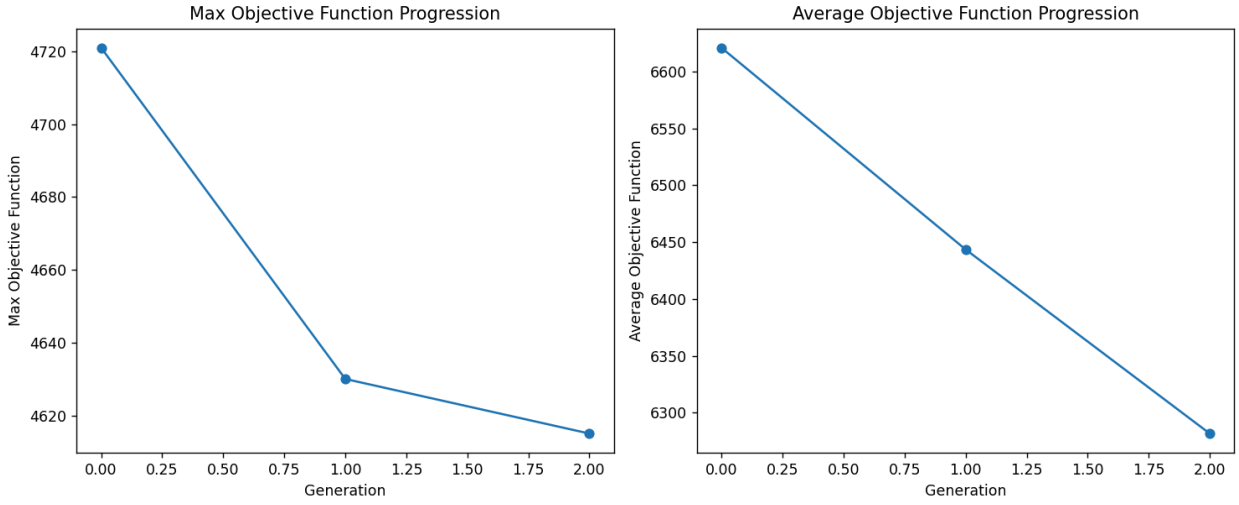


1.2.2. State akhir

```
State Kubus akhir (terbaik):
[[[105  72  76  98  97]
  [119 108  38   8  66]
  [ 16  42 104  24 107]
  [ 22  85  49  15  47]
  [ 48 110  50 125  10]]

 [[ 37  55  63  26 112]
  [ 62  64  91  23  11]
  [101 111  18  41  56]
  [ 33   9  57  40  89]
  [ 13 124  54  94  86]]

 [[ 73  12  87  80  58]
  [  7  21  79  20 123]
  [ 96  95  71  60   6]
  [ 90 113   5  75  65]
  [ 46  81 109  92  45]]

 [[ 32  88 106  30  67]
  [ 74  43  25  82 116]
  [ 84  70 115  28  52]
  [ 53  51  19  99 120]
  [  3   2 102  93  31]]

 [[114  39 103  69 100]
  [ 59  34 117  27  29]
  [ 35 121   4 122  17]
  [ 68  77  78  14 118]
  [ 36  61   1  44  83]]]
```

1.2.3. Nilai objective function akhir yang dicapai = 4615.0

1.2.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



1.2.5. Durasi proses pencarian = 5.96 detik

1.3. Percobaan ketiga

1.3.1. State awal

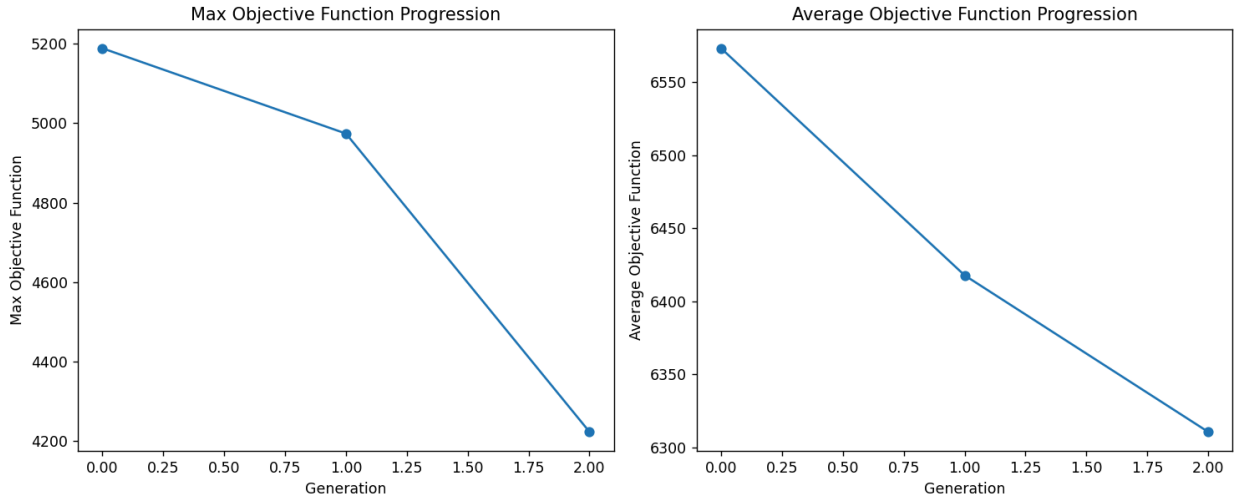
```
State kubus awal:
[[[ 61 122 108  34  29]
  [123 109  19  51  35]
  [ 65  86 110  56  20]
  [ 62 102  87  95  21]
  [ 89  72  52  71  63]]

 [[ 70  67  92  53   7]
  [ 76  66  85  28  94]
  [  3   1  74 113   9]
  [ 79 104  54  90  50]
  [120  68   8 119 125]]

 [[ 23  39  32 124   2]
  [ 11  42 107  22  33]
  [ 15  78 101 105  96]
  [118  45  46  16  27]
  [ 81 114  93   5  73]]

 [[ 38  43  36  75  97]
  [ 69 106  57  82  31]
  [ 59 115  10  83  49]
  [  6   4  48  58  37]
  [ 88 111 121  30  91]]

 [[ 80 117  47  64  84]
  [100 116  55  40  14]
  [ 18  12 103  41  60]
  [ 77  25  44  26  24]
  [ 13  98  99 112  17]]]
```

1.3.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 28  90 118  23 121]
  [ 71  53  47  20 110]
  [ 97   4  51  78  40]
  [ 38  83   9 102  33]
  [ 55   3  63  88  41]]

 [[107  54  70  17  46]
  [ 35  16  84  65  37]
  [ 58  73  31 109   2]
  [ 22  57 113  25  80]
  [111  36   6  43 124]]

 [[ 75 123  39  42 105]
  [ 30 119  64 112  12]
  [ 98  67  19  79  66]
  [ 50   8 117  99  89]
  [100  68  44 108  24]]

 [[ 26 125  92 104  52]
  [ 93 115  27  49   1]
  [ 86  10  96  13  60]
  [101  48  62  32  69]
  [ 77   5  74  76 114]]

 [[ 14  61  72 103  45]
  [ 21 120 116  56  29]
  [ 85  59  82   7  95]
  [ 87  34  11  91  81]
  [106  18  15 122  94]]]
```

1.3.3. Nilai objective function akhir yang dicapai = 4222.0

1.3.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



1.3.5. Durasi proses pencarian = 5.89 detik

2. Jumlah populasi kontrol, banyak iterasi 15 (3 kali)

2.1. Percobaan pertama

2.1.1. State awal

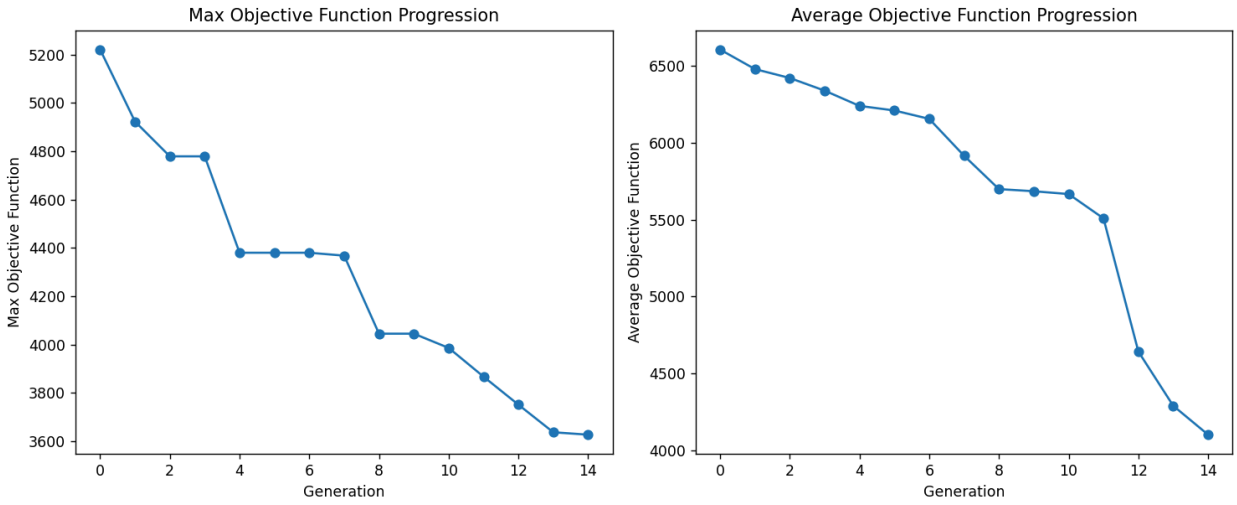
```
State kubus awal:
[[[ 71 119 118  14  27]
  [ 73 111  98  10 102]
  [ 23  96  77  91  64]
  [ 79 123  94  19  44]
  [108  35  69 113   7]]

 [[ 76  30   3 107  50]
  [ 72  70  85  13  74]
  [112  86  75  12  24]
  [110  99 104  36  68]
  [ 25  83  16 121  22]]

 [[115  11  61  15  95]
  [ 56  45  29  97  82]
  [ 90  49 105  62 114]
  [  6  42  65 120  84]
  [ 39  37  89   9 117]]

 [[  8  33  31  81  54]
  [103  93 116   5  38]
  [ 34  88  48  60  20]
  [ 51  87  67  59  32]
  [ 92   2  53  26  63]]

 [[125 122  55  41  28]
  [109  18  66  58   4]
  [ 40  57  52  80 124]
  [101 106 100  17  47]
  [ 43  46  21   1  78]]]
```

2.1.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 42  33  23  37 102]
  [109  68  91  25  14]
  [ 61  97  18  81 124]
  [  7  98  85  72  32]
  [ 66   5  80 121  35]]

 [[123  31 114  71  15]
  [ 49  78  44  57   1]
  [ 20  62  99  12 106]
  [ 51  54  70 100  55]
  [ 96  79  46   8 113]]

 [[ 63  11  30  84 107]
  [ 39  40 116  22  94]
  [122  13  65  34  16]
  [ 90  92  77 110 125]
  [ 26 117   3  87  24]]

 [[ 50  56  73  93  74]
  [ 29  43  53  19 104]
  [108  41 120  58  21]
  [ 10  64  59 111 119]
  [ 89 118   2  95  36]]

 [[ 83 105   6   4 101]
  [ 75  17 115  28  82]
  [ 67  27  48 103  47]
  [ 76 112  45  69  38]
  [  9  52  86  88  60]]]
```

2.1.3. Nilai objective function akhir yang dicapai = 3627.0

2.1.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



2.1.5. Durasi proses pencarian = 17.85 detik

2.2. Percobaan kedua

2.2.1. State awal

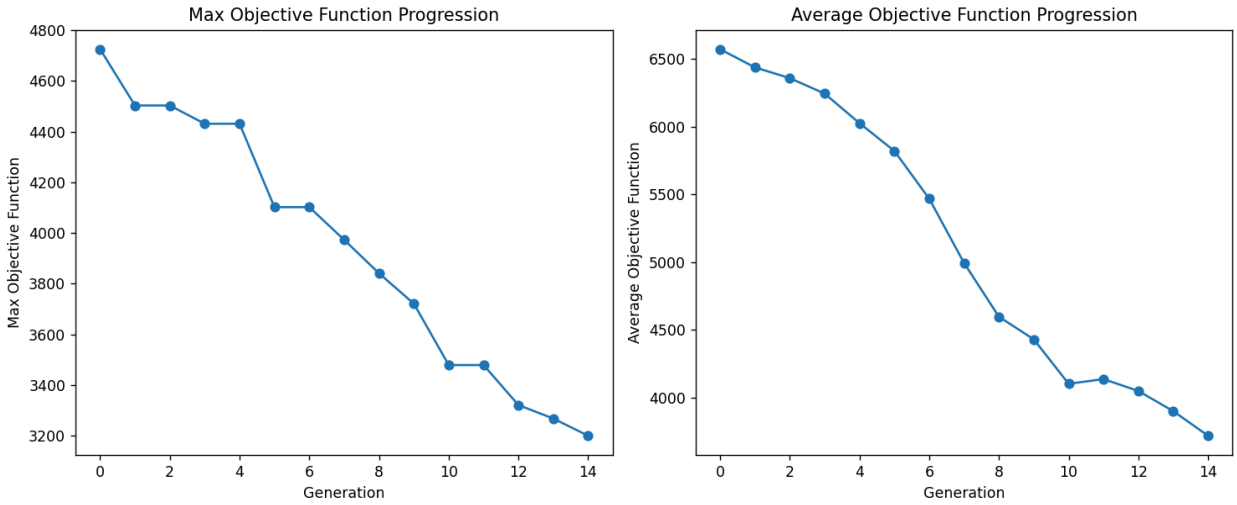
```
State kubus awal:
[[[ 15  18   6 106  26]
  [ 36  42  14 117   8]
  [ 75  19  11  10  22]
  [ 20 124  90 125  37]
  [ 84 118  62  44  91]]

 [[ 87  25  29  81 113]
  [ 78  80  74  96  12]
  [107 120 119  70 121]
  [ 65  82  64  46  32]
  [ 51  43 123  99 115]]

 [[ 57  39  67 112  95]
  [ 53   1   5  88  61]
  [ 71 103  76  72  93]
  [ 92 114  41  24  77]
  [110  30 101  60  50]]

 [[109  31  79  59  54]
  [104  47 116  48 105]
  [100  83  35  21   9]
  [ 86  94  23  49  98]
  [ 33  17   3  34  40]]

 [[102  38  45   4  52]
  [ 27  66  89 111  73]
  [ 56  16  68 108   7]
  [ 13  58  28  69  97]
  [ 55  63   2  85 122]]]
```

2.2.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 84  58  55  79  53]
  [ 27  63  91 106  38]
  [ 94  44  13  36  65]
  [  5  64 121  34 100]
  [105  75  18  69 109]]

 [[ 62 102  31  52  43]
  [ 15  76  66 104 118]
  [ 56   9  72 114  30]
  [ 60  51  99   2  78]
  [ 95 117  49  32  37]]

 [[ 89   7  28 113  82]
  [108  48  39  67  33]
  [ 17  46  93  90  24]
  [ 71  80  29   1 101]
  [ 22  81 122  41  83]]

 [[ 20  96  11  74  85]
  [ 98  23 124  40   3]
  [ 86 103  97  25  12]
  [116  87  16  59 125]
  [  8  70  73 123  61]]

 [[ 88  42   4 112  21]
  [115  92  10  54 120]
  [ 26  57  77  14  45]
  [ 47 110  50  19 119]
  [ 68  35 111 107   6]]]
```

2.2.3. Nilai objective function akhir yang dicapai = 3200.0

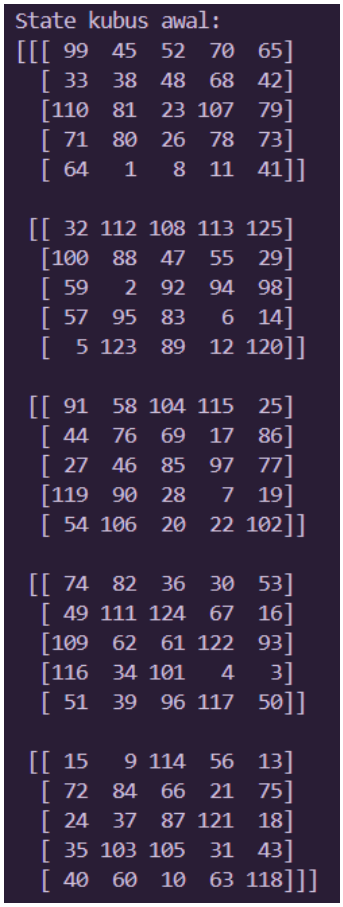
2.2.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



2.2.5. Durasi proses pencarian = 17.70 detik

2.3. Percobaan ketiga

2.3.1. State awal

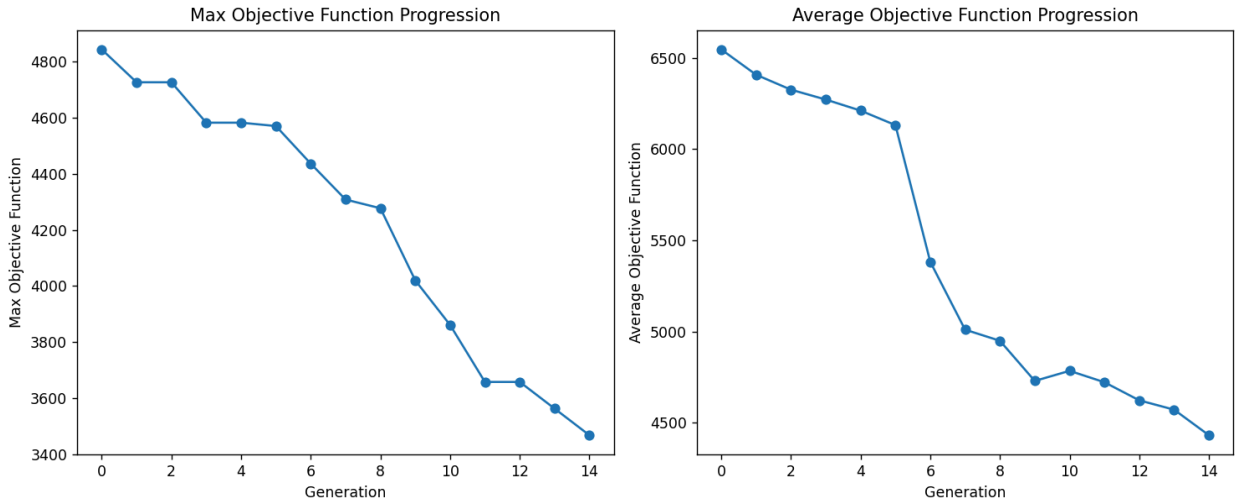


2.3.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 51 109  37  94   7]
  [117  55 125   1  33]
  [  4  76  79  92  74]
  [ 38  68  65  35  96]
  [111  44  28  84  98]]

 [[ 24  42 119 100  22]
  [118  85  27  62  18]
  [ 66 108  83  25   3]
  [  6  31 103  71 124]
  [ 73  11  39  43 120]]

 [[ 30  64 115  14 105]
  [ 29  90  40 116  20]
  [ 70  41  88  59  72]
  [ 32  15 123  80  57]
  [113  82   9   8  46]]

 [[ 81  63  23  78 121]
  [102  93  19  17  56]
  [ 95 112 114  75  77]
  [ 48  50  60  69 110]
  [  2  16 106  58   5]]

 [[ 34 107  26  87  12]
  [ 36  99  52  97  61]
  [ 13  49  89  47 104]
  [122  67 101  10  53]
  [ 91  54  45  21  86]]]
```

2.3.3. Nilai objective function akhir yang dicapai = 3468.0

2.3.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



2.3.5. Durasi proses pencarian = 41.05 detik

3. Jumlah populasi kontrol, banyak iterasi 125 (3 kali)

3.1. Percobaan pertama

3.1.1. State awal

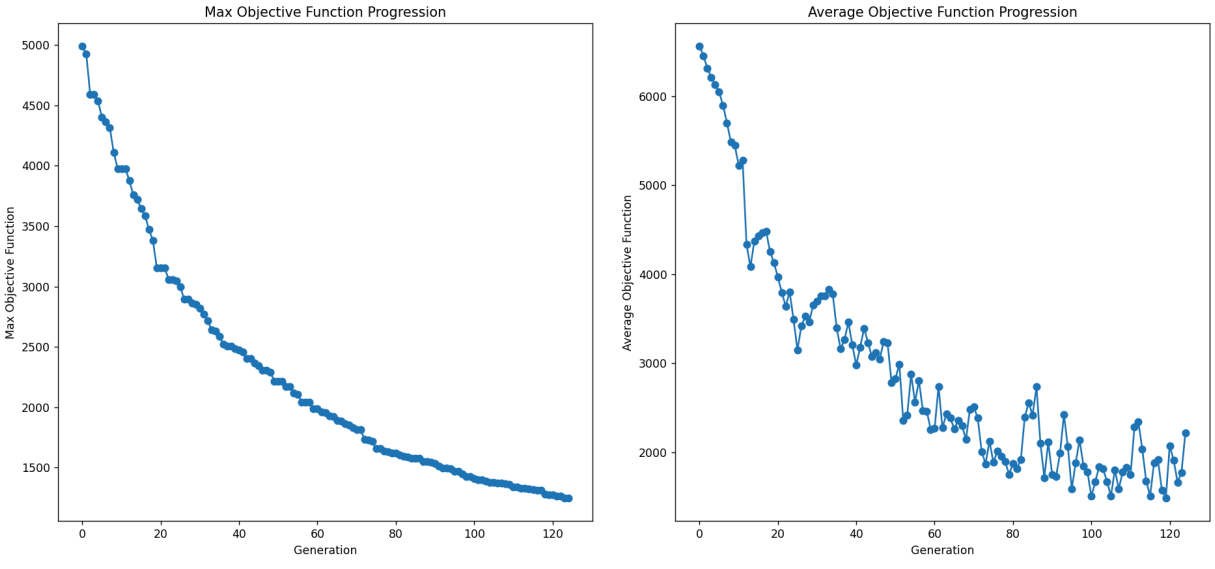
```
State kubus awal:
[[[ 24  20  39  61  54]
  [  5  51  50  71 103]
  [ 60  22 116  85  87]
  [ 64 117  83  97  19]
  [ 88  32  99 114  43]]

 [[102  66  59 119  90]
  [ 68  11  17 107  13]
  [ 30  55   9  42  48]
  [ 58  36  81  84  25]
  [109 101 113  72  69]]

 [[ 98   7  76 104 118]
  [108 124 120  23  15]
  [  3  62  75  86  79]
  [  1 110 123  44  38]
  [ 34  27  70  92  35]]

 [[  2  63  29 115 121]
  [ 93  40  73  45  28]
  [111  53   6  96  12]
  [ 77  14  16 125  95]
  [ 82  46  57  18  21]]

 [[ 10  65  49  94  41]
  [ 26  89 122  56  31]
  [ 37 100  80 105 106]
  [ 47   8 112  67  74]
  [ 33  78   4  91  52]]]
```

3.1.2. State akhir

```
State Kubus akhir (terbaik):
[[[  4 115  98  10  89]
  [ 28  47 104  18  83]
  [ 96  30  67  99  16]
  [ 85  34  42 118  45]
  [103  86   7  61  81]]

 [[ 43 119   6  26 111]
  [108  65  48  35  20]
  [ 58  49  41 125  59]
  [ 22  73 123  76  37]
  [ 82   9  97  53  84]]

 [[ 17 116  24 102  68]
  [ 87  79  93  60   1]
  [ 80  72  52  64  50]
  [ 23  27 122  51  78]
  [105  21  25  39 117]]

 [[ 40 101  19  95  54]
  [ 90   8 113  38  66]
  [ 56  77  62  92  33]
  [  5  36 106  91  55]
  [121  94  15   2 107]]

 [[ 69  71  12  75  46]
  [ 70  14  44  74 110]
  [ 63 112 100  29  11]
  [ 31 120 109  13  32]
  [ 88   3  57 124 114]]]
```

3.1.3. Nilai objective function akhir yang dicapai = 1246.0

3.1.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



3.1.5. Durasi proses pencarian = 311.33 detik

3.2. Percobaan kedua

3.2.1. State awal

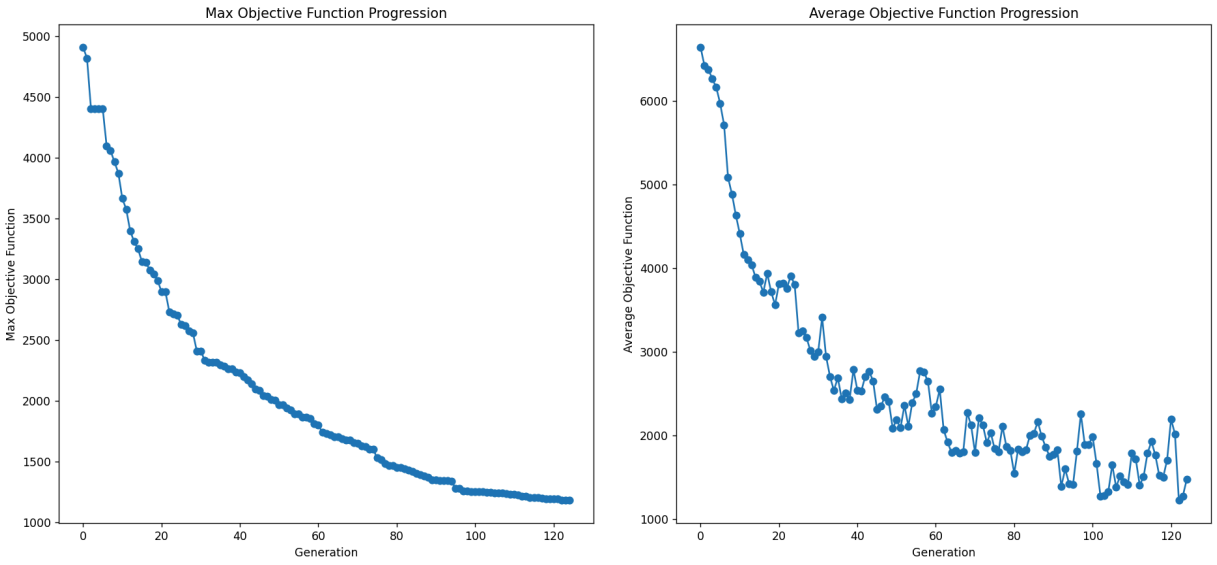
```
State kubus awal:
[[[ 54 121  51  91  84]
  [ 66   6  74  37 104]
  [110  11  64  98  20]
  [ 97  85  44  31 119]
  [ 10  34  92 113  88]]

 [[ 35  86  27   7  79]
  [ 58  46 118 108  61]
  [ 55 125  73  75  59]
  [  2   9   3  52 120]
  [ 95  87   1  71  23]]

 [[ 29  13  57  24  89]
  [ 60  62  72  81  18]
  [ 41   8  33  14  22]
  [123  83  30  12  99]
  [  4  15  78  48  93]]

 [[ 63  32 116  49  90]
  [ 45  50  47 106  36]
  [ 26  96 105 107  43]
  [ 42 100  16 112 103]
  [ 65 117  69  76 115]]

 [[ 40  21  94  82 102]
  [ 77  56 111 124  19]
  [ 39  80  53  70  68]
  [ 67 101  28 109   5]
  [ 38 114 122  17  25]]]
```

3.2.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 78  68  84  45  71]
  [  1  72 121   7 124]
  [112  15   6  91  28]
  [ 14  61  97 109  38]
  [106 102   5  64  52]]

 [[ 58 101  34  25 110]
  [122  18  98  39  42]
  [  3  35 107 100  26]
  [ 94  59  66  36  43]
  [ 37  95  10 117  96]]

 [[ 65   2  56 118  75]
  [ 53  87 108  12  79]
  [ 27  62  88  32 105]
  [ 83 123  51  33  19]
  [ 86  46  11 120  40]]

 [[125  47  82  77  31]
  [ 13  23  89  80 111]
  [ 29  85  30  70  99]
  [ 81  92   4  74  22]
  [ 69  76 104  16  49]]

 [[116  21  63 103   8]
  [ 48  17  60  55 114]
  [ 54 113 115  44   9]
  [ 57  93  24  20 119]
  [ 41  73  50  90  67]]]
```

3.2.3. Nilai objective function akhir yang dicapai = 1183.0

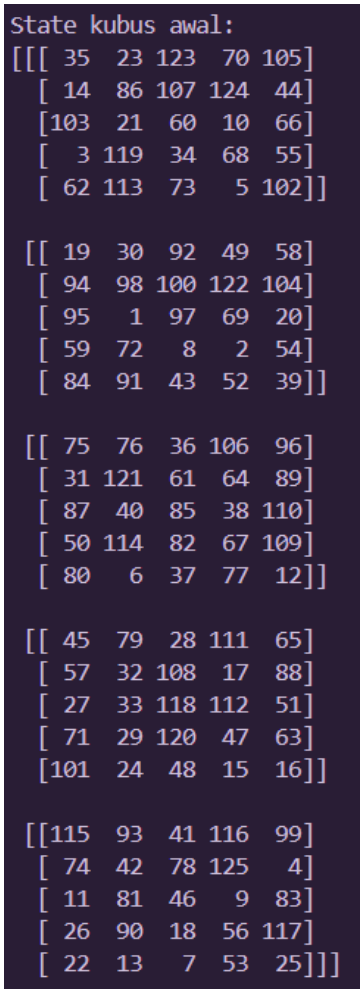
3.2.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



3.2.5. Durasi proses pencarian = 315.92 detik

3.3. Percobaan ketiga

3.3.1. State awal

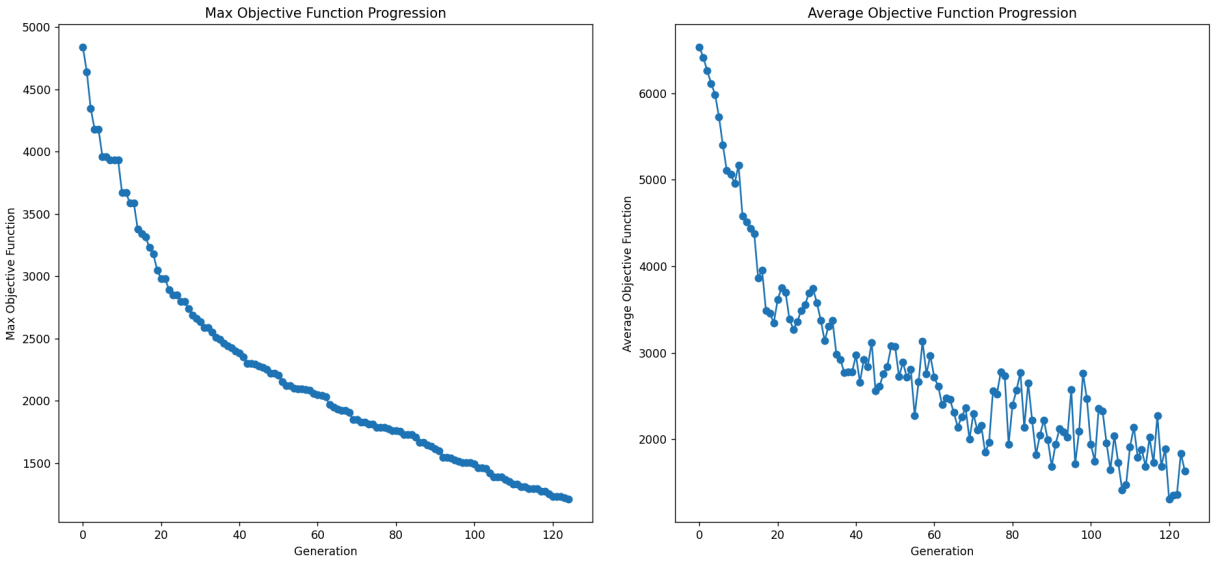


3.3.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 59  97  78   1  42]
  [ 96  70  53  54  43]
  [ 46   8  77  99 111]
  [ 32 105  63  47  68]
  [ 82  40  30 113  52]]

 [[ 80  90  26  15  95]
  [ 75   6 110  23 101]
  [ 91  10  81  56  76]
  [ 14  84  86 118  12]
  [ 55 124   4 104  31]]

 [[ 17  65  64 107  58]
  [ 72  57  98   5  87]
  [108   3 103  33  69]
  [ 92 117  21  48  37]
  [ 28  74  25 125  67]]

 [[119 109  11  49  44]
  [ 60   2  85  24 123]
  [ 34  88  50 106   7]
  [ 19  89  45 102  83]
  [ 79  27 120  38  62]]

 [[ 73 100  66  29  51]
  [115  61   9  22 121]
  [ 36  18 114 112  35]
  [ 16  41 116  39  93]
  [ 71  94  13 122  20]]]
```

3.3.3. Nilai objective function akhir yang dicapai = 1209.0

3.3.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



3.3.5. Durasi proses pencarian = 190.61 detik

4. Banyak iterasi kontrol (50), jumlah populasi 50 (3 kali)

4.1. Percobaan pertama

4.1.1. State awal

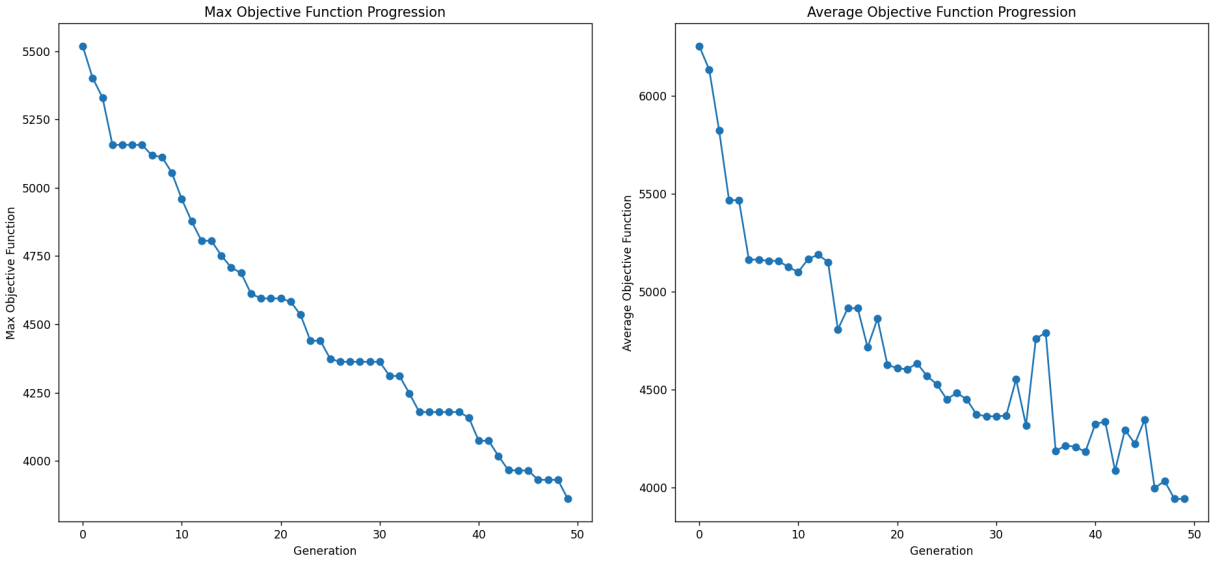
```
State kubus awal:
[[[ 50  57  43  31  79]
  [  8  16  85  25   6]
  [ 55 119  58 106  35]
  [ 80 118  65  10 110]
  [112  95  56  83  41]]

 [[104  53 123  59  48]
  [ 73 103  61  96  26]
  [ 21   7  74  11  71]
  [  3 120   5  70  27]
  [ 12  75   1 122  34]]

 [[115  28  23  67  18]
  [107  32  14 114  24]
  [ 77  29  94  66  19]
  [113  40 117  37   2]
  [ 38 109  82  86  49]]

 [[ 17 105  42  47 111]
  [ 13 108  91  22  78]
  [ 97  68 101  84  92]
  [ 54  36  60  45  64]
  [ 72  15  51 116   9]]

 [[ 63  89  30  44 102]
  [ 33 125  20  99  87]
  [ 39 100  81  98  93]
  [  4  76 124 121  88]
  [ 62  90  69  52  46]]]
```

4.1.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 15  27  55 125  87]
  [ 52  90  51   4  23]
  [  5  24  94  95  63]
  [119 117  67  45  32]
  [109  56  20  35 113]]

 [[ 83  84 118  99 107]
  [104  89  14   8  75]
  [ 86  22  64  97  71]
  [  6  44  33  78  12]
  [ 39  47  92  36  81]]

 [[ 19  38  40  60 101]
  [  2 106 100  18  73]
  [ 93  65  91  88  58]
  [102  46 124 111  25]
  [ 69  68  10  16 122]]

 [[120  98  77  31  49]
  [ 96  42  70  50  72]
  [ 30  76  11 108 110]
  [114  41 112  61  13]
  [  7  54  37 103  66]]

 [[ 29 121  26  59  74]
  [115  82   1  34  28]
  [ 21  17 123 116  80]
  [ 62  57  43  48   9]
  [ 85   3  79  53 105]]]
```

4.1.3. Nilai objective function akhir yang dicapai = 3861.0

4.1.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



4.1.5. Durasi proses pencarian = 5.23 detik

4.2. Percobaan kedua

4.2.1. State awal

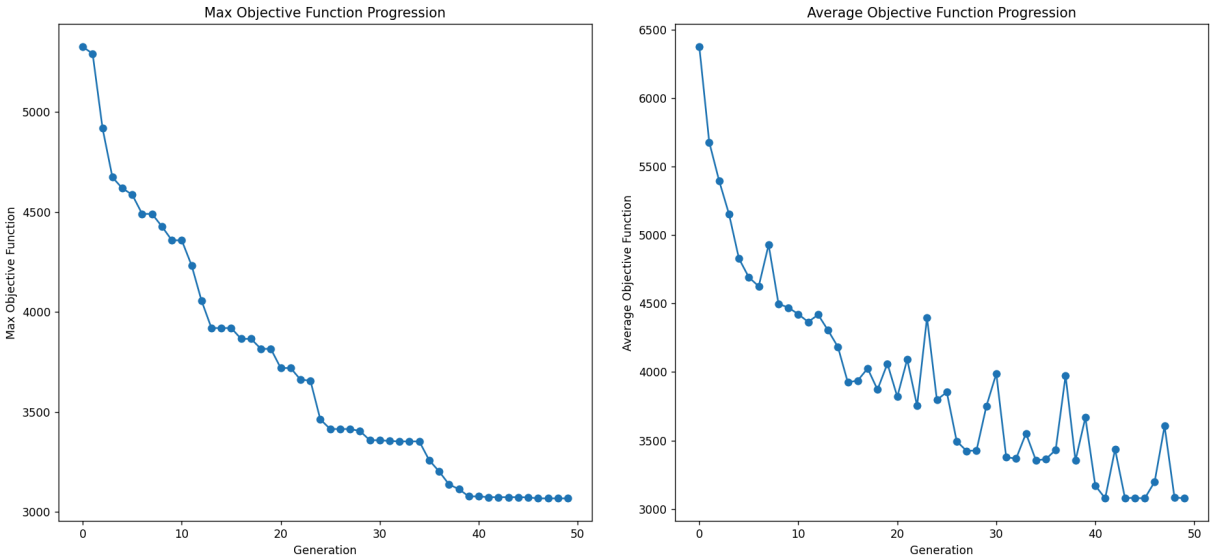
```
State kubus awal:
[[[ 65  45   4  68  63]
  [ 89 100  80  75  90]
  [125  48 122  46  55]
  [ 98  34  78  56  67]
  [ 77  51   1 119  23]]

 [[103  84  81   2 121]
  [110  32  97  94 117]
  [ 44 102  64  70  13]
  [ 52  92 124  27   9]
  [ 35  87  88  86  43]]

 [[ 49  39  83 111  59]
  [120  69 104  30  24]
  [ 37  50  61  36  76]
  [ 25  66  11 116  19]
  [ 54 109 112  82  15]]

 [[ 62   8  79  85  58]
  [101  47  71  74  72]
  [115 108  20 105 107]
  [ 31 123  14  41  12]
  [  3  22   6  96  10]]

 [[ 53 113  28  60  42]
  [106  91 118  16  73]
  [  5  57  33  29  95]
  [ 18  26  99  38  17]
  [114   7  21  93  40]]]
```

4.2.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 69 122  50 105   3]
  [ 19 114  38  72  39]
  [124  14  40   4 110]
  [ 55   8  76 103  83]
  [ 84  96 106  16  73]]

 [[ 95   1 113  93   7]
  [ 21 100  90  78  56]
  [ 51  27  86  37 108]
  [ 52  46  12  59  79]
  [ 61  97  31  58  42]]

 [[116  47 118  98  11]
  [107  36  17  75  18]
  [ 13  92  57   6 120]
  [ 33  74  91   5  49]
  [ 29  43  28 109 111]]

 [[  9  81 101  66  67]
  [115  45  71  89  10]
  [ 99  60  54  26  64]
  [112 102  94  15  48]
  [ 25  24   2 119 123]]

 [[ 53  63  80  70  35]
  [121  65  23  68  82]
  [ 20  77 104  44  88]
  [ 34  22  62 125  87]
  [ 85  41  32 117  30]]]
```

4.2.3. Nilai objective function akhir yang dicapai = 3068.0

4.2.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



4.2.5. Durasi proses pencarian = 5.36 detik

4.3. Percobaan ketiga

4.3.1. State awal

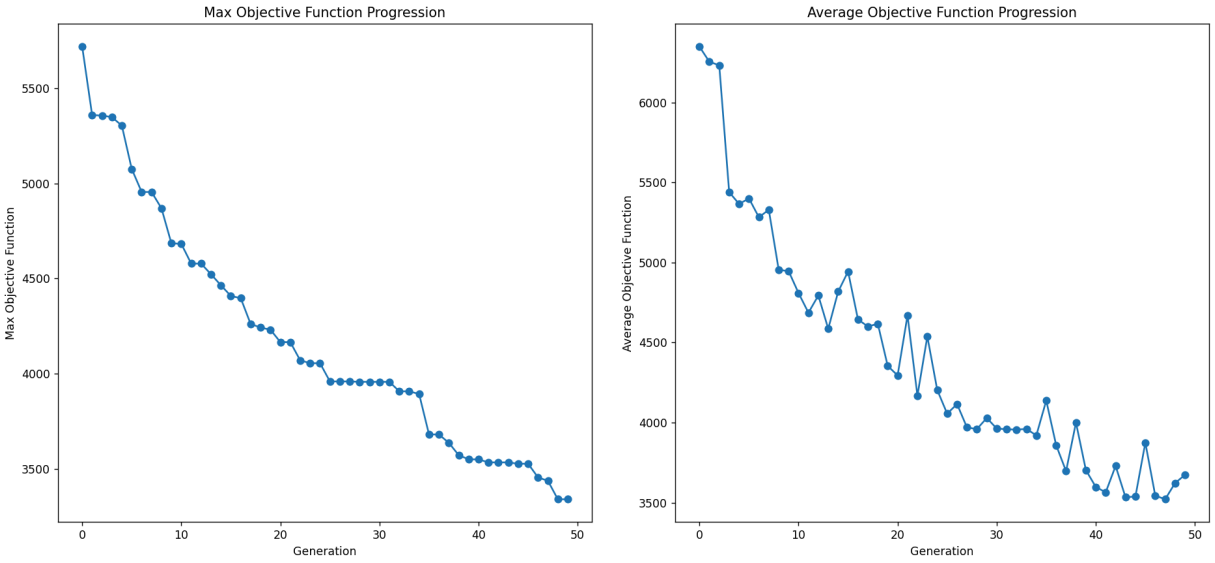
```
State kubus awal:
[[[ 34   8  92  29  78]
  [106 109  18 108 112]
  [ 98  21  28 102  36]
  [ 99  13  94 119  43]
  [ 39  41  82   2  15]]

 [[ 91  51  47  45  70]
  [120  76  74  85  59]
  [  4   7  38  68 107]
  [115   9  65 101  69]
  [122  83  54  84  22]]

 [[123 104 114  96  89]
  [ 86   1  46  10  67]
  [ 55 118 105  63  77]
  [100  17  81  50 117]
  [  3 121  72  95 103]]

 [[ 31  57  73  71  30]
  [ 24 111  23 116  66]
  [ 52  80  42  40  87]
  [110  33  12  11  58]
  [113  93  32  37  26]]

 [[ 88  60  97 125  14]
  [  5  19  25  56  61]
  [ 20  64  35  48  44]
  [ 53  16  27  90 124]
  [  6  75  62  79  49]]]
```

4.3.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 95  44  10 105  84]
  [ 34  67  22  45 106]
  [ 11 120  99 101  77]
  [122  54   2  25  33]
  [ 75  24 112  37  12]]

 [[ 74  13 115  41  55]
  [109  52  85  29 107]
  [ 58 117  48   6  43]
  [ 46  86  27 104  61]
  [ 36  91  40 111  42]]

 [[ 87  72   5  96  82]
  [121  62  79   1  47]
  [ 20  94 113  35  81]
  [ 38  68  17 118  31]
  [ 53  59 100  93  18]]

 [[ 16  90  71 125  26]
  [ 50  15 123 110  65]
  [ 70  98  76  32  28]
  [ 92  49  69  51  64]
  [108  57   3  89 102]]

 [[ 88  80  56   9  97]
  [114  19   7 103  14]
  [ 30  83 124  21  66]
  [ 63  60   8  73 116]
  [ 39  23 119  78   4]]]
```

4.3.3. Nilai objective function akhir yang dicapai = 3339.0

4.3.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



4.3.5. Durasi proses pencarian = 6.11 detik

5. Banyak iterasi kontrol (50), jumlah populasi 500 (3 kali)

5.1. Percobaan pertama

5.1.1. State awal

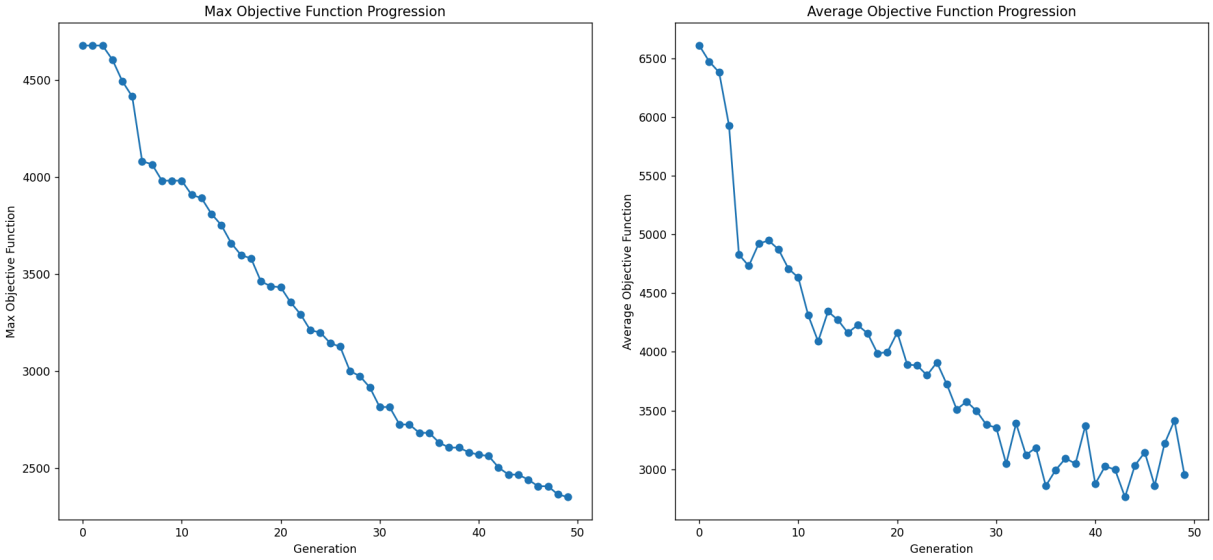
```
State kubus awal:
[[[105 123  58  20  61]
  [ 84  16  96  71  79]
  [  2   9  37  95  81]
  [ 24 111  35 120  64]
  [ 54  78 118  41   1]]

 [[ 27   4  70  26 115]
  [ 47  51  48  11 100]
  [ 77  36  34  15 108]
  [ 39  82  75  63  66]
  [ 12  80  53 102  28]]

 [[101  94  19  14  49]
  [ 22 121  89  67 124]
  [ 87  30  68  72  69]
  [ 76 110  62  29  13]
  [ 99  40 107 104 109]]

 [[ 93  88  83  38  52]
  [ 92 119  25  10   8]
  [125   3  91 106  43]
  [ 42  45  50 122  59]
  [ 33  98  46  17  57]]

 [[ 60  56  97  65  85]
  [ 32 117   6 113  73]
  [ 18 116 103   7  44]
  [ 90 112  74  23  31]
  [ 86   5  21  55 114]]]
```

5.1.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 27  43 123 110   9]
  [100  54  14  39 120]
  [ 46  61 119  17  73]
  [ 94  86  23  69  18]
  [ 47  76  11  91 104]]

 [[ 81  49  28  40  85]
  [ 34  64  63  77  31]
  [ 67 109  10  96  57]
  [ 99  66 117   7  20]
  [ 51  32  82  89 115]]

 [[ 80  84 102  35  26]
  [ 50  52  22 121  78]
  [ 88  15  70 124  24]
  [ 95  90  45  19  98]
  [ 13  93  83   3  68]]

 [[ 74 125  12  62  55]
  [ 30  21 114  48  36]
  [101  65  79 105  58]
  [  1 113  38  53 107]
  [122   4  71  41  60]]

 [[ 37 111  72  42  56]
  [ 92  33 108  59 112]
  [ 25  29 103  97   5]
  [116  75   6 106   8]
  [ 44  87  16   2 118]]]
```

5.1.3. Nilai objective function akhir yang dicapai = 2351.0

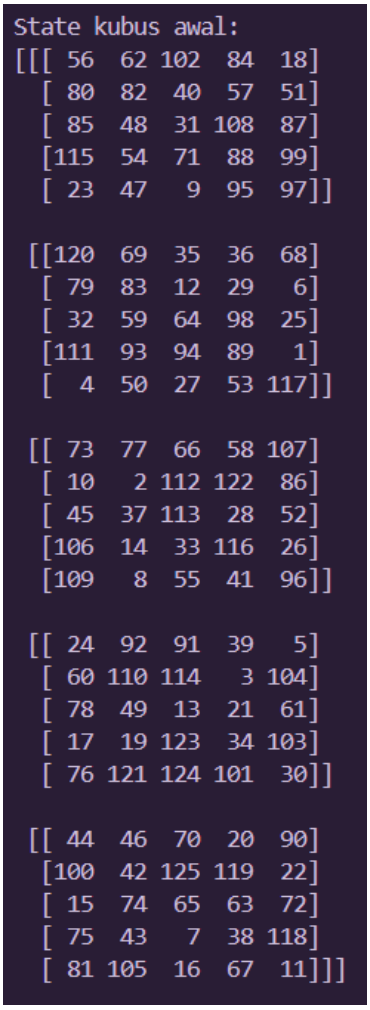
5.1.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



5.1.5. Durasi proses pencarian = 26.91 detik

5.2. Percobaan kedua

5.2.1. State awal



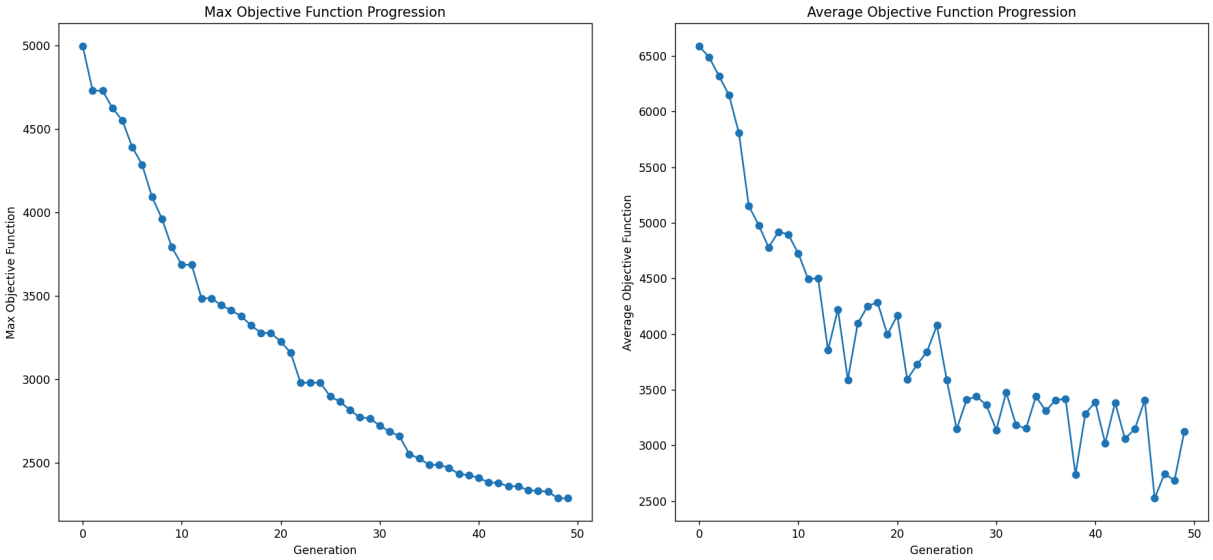
```
State kubus awal:
[[[ 56  62 102  84  18]
  [ 80  82  40  57  51]
  [ 85  48  31 108  87]
  [115  54  71  88  99]
  [ 23  47   9  95  97]]

 [[120  69  35  36  68]
  [ 79  83  12  29   6]
  [ 32  59  64  98  25]
  [111  93  94  89   1]
  [  4  50  27  53 117]]

 [[ 73  77  66  58 107]
  [ 10   2 112 122  86]
  [ 45  37 113  28  52]
  [106  14  33 116  26]
  [109   8  55  41  96]]

 [[ 24  92  91  39   5]
  [ 60 110 114   3 104]
  [ 78  49  13  21  61]
  [ 17  19 123  34 103]
  [ 76 121 124 101  30]]

 [[ 44  46  70  20  90]
  [100  42 125 119  22]
  [ 15  74  65  63  72]
  [ 75  43   7  38 118]
  [ 81 105  16  67  11]]]
```

5.2.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 62  67 120  23  49]
  [ 42 114  13 122  55]
  [108  53  41  37  83]
  [ 26  24 105  28  97]
  [ 78  60  16 107  44]]

 [[ 18 113  59  89  40]
  [102  72   2  99  17]
  [117   1  27  98  61]
  [ 33 100 124   4  84]
  [ 46  48 116  77 123]]

 [[ 70  86  15 112  20]
  [ 10  45  68 121  91]
  [ 66  63  51  35  38]
  [ 36  65 118  32  80]
  [125  54  52  25  96]]

 [[ 57  82  11 110  58]
  [ 88 115   9  79  71]
  [ 81  21  75  64  69]
  [ 39  34 111  29 106]
  [ 47  50 119  30  12]]

 [[ 56   6 104  93   3]
  [ 22  85  31 103  43]
  [ 94  14  90   8  87]
  [ 19 109   5 101  73]
  [ 74  92  76   7  95]]]
```

5.2.3. Nilai objective function akhir yang dicapai = 2288.0

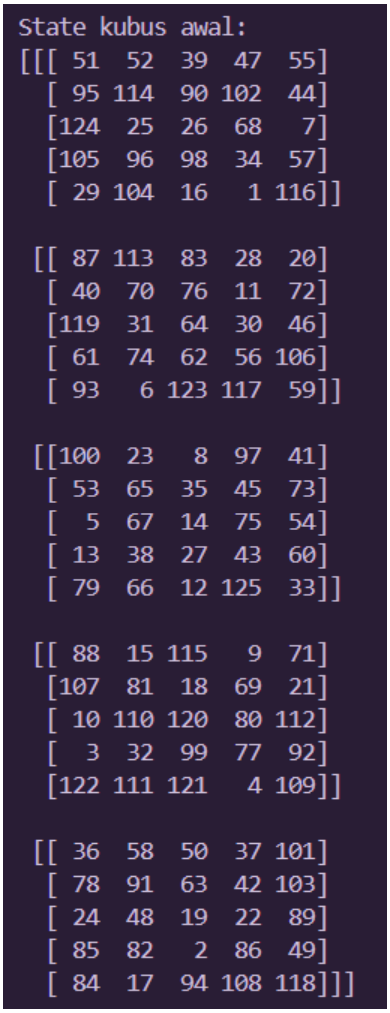
5.2.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



5.2.5. Durasi proses pencarian = 27.22 detik

5.3. Percobaan ketiga

5.3.1. State awal



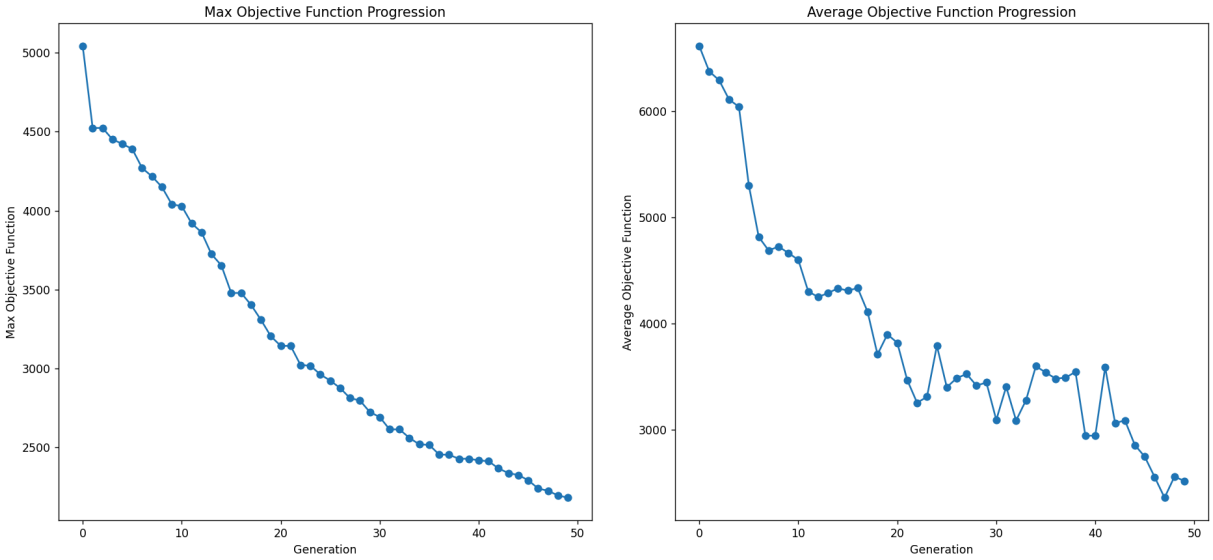
```
State kubus awal:
[[[ 51  52  39  47  55]
  [ 95 114  90 102  44]
  [124  25  26  68   7]
  [105  96  98  34  57]
  [ 29 104  16   1 116]]

 [[ 87 113  83  28  20]
  [ 40  70  76  11  72]
  [119  31  64  30  46]
  [ 61  74  62  56 106]
  [ 93   6 123 117  59]]

 [[100  23   8  97  41]
  [ 53  65  35  45  73]
  [  5  67  14  75  54]
  [ 13  38  27  43  60]
  [ 79  66  12 125  33]]

 [[ 88  15 115   9  71]
  [107  81  18  69  21]
  [ 10 110 120  80 112]
  [  3  32  99  77  92]
  [122 111 121   4 109]]

 [[ 36  58  50  37 101]
  [ 78  91  63  42 103]
  [ 24  48  19  22  89]
  [ 85  82   2  86  49]
  [ 84  17  94 108 118]]]
```

5.3.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 18  11 122  49 106]
  [ 20 112   4  96  60]
  [109  84   2  30  41]
  [ 79   6 102 110  43]
  [ 90 115  73  27  99]]

 [[100  12  19  68 120]
  [ 17  85  59  83  51]
  [ 54  65 124  48  38]
  [105  33   9 104  61]
  [ 34 114  98  29  40]]

 [[ 56   7 101  88  74]
  [ 80  31 116  58  28]
  [ 64 108  25  63  44]
  [ 92 125  16  26  45]
  [ 24  42  53  71 117]]

 [[  3  35 111  69  91]
  [ 76  72 113  21   8]
  [ 95  89  32  50  66]
  [ 97  23  22 119  82]
  [ 52  93  36  55  67]]

 [[ 14  70  81 107  78]
  [  1 121  47   5  62]
  [118  37  39  15 123]
  [ 86  10  94 103  46]
  [ 77  87  57  75  13]]]
```

5.3.3. Nilai objective function akhir yang dicapai = 2182.0

5.3.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



5.3.5. Durasi proses pencarian = 27.12 detik

6. Banyak iterasi kontrol (50), jumlah populasi 2000 (3 kali)

6.1. Percobaan pertama

6.1.1. State awal

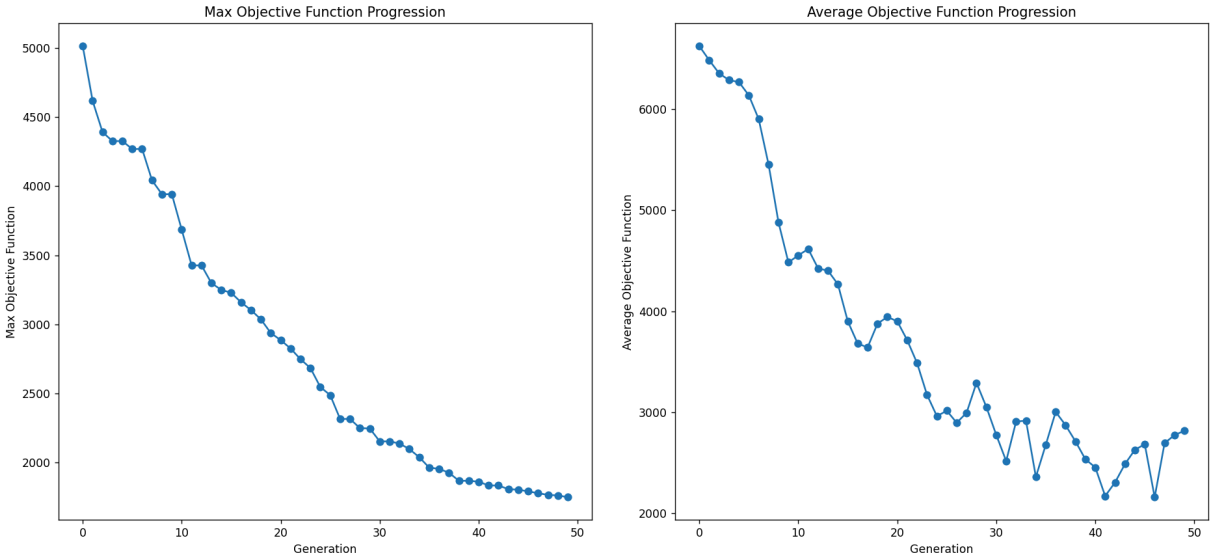
```
State kubus awal:
[[[ 68  12  85   6 125]
  [111 120  89  97 112]
  [  9 115 102  19  70]
  [ 25  40 124  86  84]
  [106  58  77 122  43]]

 [[ 98 114  27  46  95]
  [  2  30 105  59 109]
  [ 78  63  61 119  93]
  [ 10  29 108 103 104]
  [ 71  26  35  60  54]]

 [[ 14  62  28  18  57]
  [ 15  47  90  31  49]
  [ 94  32   4  96 121]
  [113  55  13  66  36]
  [ 56 101 100  42  22]]

 [[ 44  24  41   1   3]
  [ 45  53  87  69  39]
  [ 23  20  99  64  21]
  [110  73 107  82  75]
  [ 67  11   7  50  72]]

 [[ 38  33  76  16  51]
  [ 80 116  74  88  65]
  [118 117 123  52  48]
  [ 91   5  17   8  83]
  [ 81  34  79  92  37]]]
```

6.1.2. State akhir

```
State Kubus akhir (terbaik):
[[[104  18  72  67  53]
  [ 46 124  26 102  17]
  [ 65   1  33  90 120]
  [ 93 101  89  41  49]
  [ 29  70  98  15  60]]

 [[111  11  95  42  94]
  [  6  55  97  88  35]
  [ 66  87  75  19  68]
  [ 83  24  21  64 114]
  [ 40 125  28 109   3]]

 [[122  34  54  61  69]
  [  7  32  37  80 103]
  [113  73  74  48  27]
  [ 44 105 115  78   2]
  [ 30  76  36  45 107]]

 [[119  14   4  63 108]
  [ 86 121  20  77  22]
  [ 96  43  59  82  50]
  [ 12  56 118   9 117]
  [  5  85 116 100   8]]

 [[ 25  47  62 110  81]
  [ 84  23  92  16  71]
  [ 58  57 106  91  10]
  [112  79  39  38  51]
  [ 31 123  13  52  99]]]
```

6.1.3. Nilai objective function akhir yang dicapai = 1751.0

6.1.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



6.1.5. Durasi proses pencarian = 100.78 detik

6.2. Percobaan kedua

6.2.1. State awal

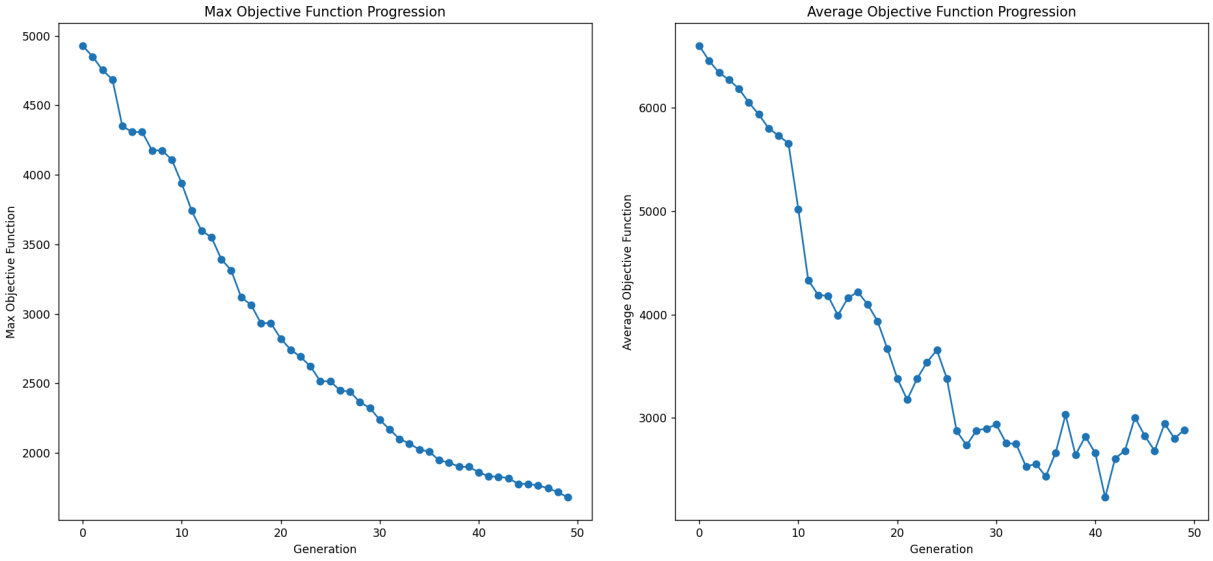
```
State kubus awal:
[[[116  25  99  12  42]
  [ 90  30  24   5  59]
  [ 18  33  28  79  34]
  [ 17  11   4  72  85]
  [ 23  56  93  45  70]]

 [[ 78   8  10  82  21]
  [ 20   2  76 120 101]
  [ 29  73 104  14 124]
  [108  95  19 118 103]
  [ 68  27  37  57 121]]

 [[102 123  94  77  62]
  [ 64 106 119 107 122]
  [  7  43  89  15  61]
  [ 50   1  60  40  65]
  [110 113   9 114  31]]

 [[ 52  81  87  38  71]
  [ 88  91  35  74  86]
  [125 111  44  58  47]
  [ 49 100 112  16   6]
  [ 46  80  84  41  63]]

 [[ 75  98  97  32  67]
  [ 55  26 109 117  51]
  [ 69 105  96  13   3]
  [ 92  53 115  22  66]
  [ 48  83  36  54  39]]]
```

6.2.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 10  32  67 117  86]
  [ 96 121   1   7 116]
  [ 38  51 123  57  36]
  [103  17 118  35  39]
  [ 80  90   9 109  34]]

 [[ 11  98  46 111  27]
  [ 79  93  87  48   2]
  [ 88  26  44 100  61]
  [ 65  85  29  60 122]
  [ 77   8 119   4 105]]

 [[ 97 101  23  91   3]
  [114  20  16 115  70]
  [ 62  81  49  52  66]
  [ 18  73 112  56  55]
  [ 19  37 113  13 110]]

 [[120  24  54 106  31]
  [ 25  22 102  43 124]
  [ 63 108  64  89  33]
  [ 15  78  75  42  94]
  [ 92  83  30  45  28]]

 [[ 84  58 107   5  69]
  [ 21  72 104  74  68]
  [ 40  12  41 125  95]
  [ 53  99  14  59   6]
  [ 82  76  50  47  71]]]
```

6.2.3. Nilai objective function akhir yang dicapai = 1679.0

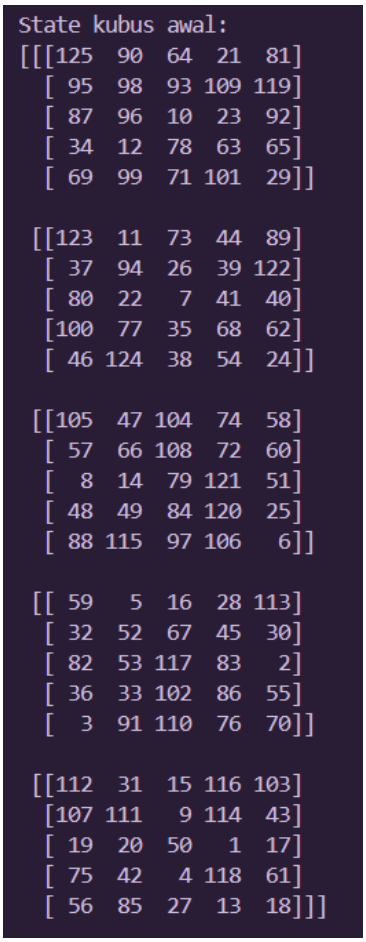
6.2.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



6.2.5. Durasi proses pencarian = 100.29 detik

6.3. Percobaan ketiga

6.3.1. State awal

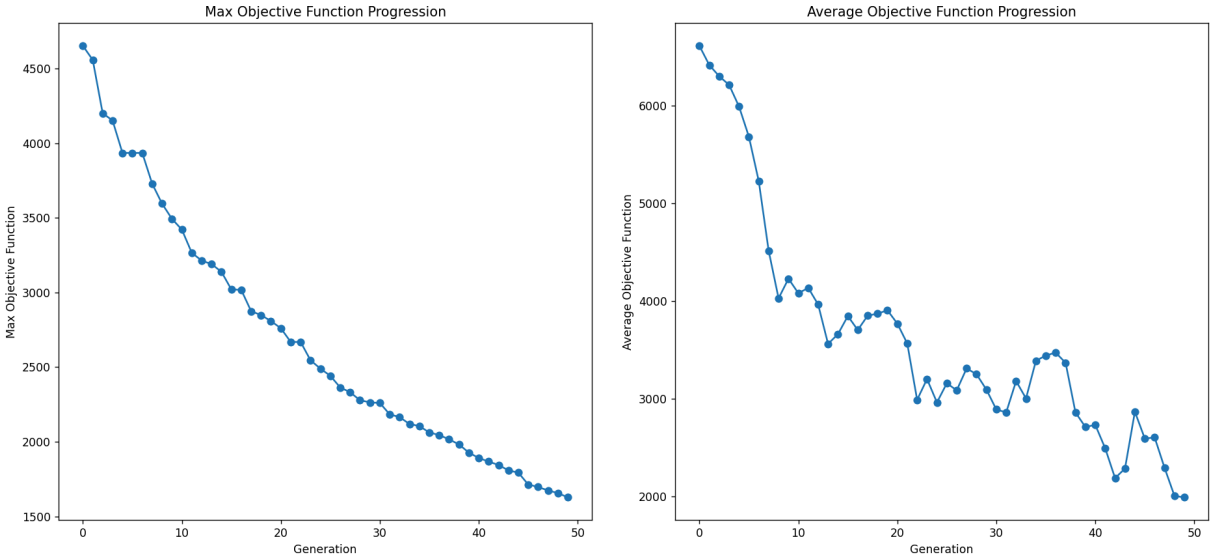


6.3.2. State akhir

```
State Kubus akhir (terbaik):
[[[ 46  85  42  62  89]
  [  4  81 122  53  12]
  [ 78 102  25  14  77]
  [ 47  36  60 123  79]
  [118  13  86  69  44]]

 [[ 26 117  65  76 111]
  [ 31 120  93  17  15]
  [ 97  40  23 103  52]
  [ 55  32  98  66  41]
  [106  11  45  58 100]]

 [[ 54  22  61 101  73]
  [ 83  88 110  10  19]
  [ 18  84  82   5 121]
  [ 27  74  38 105  92]
  [119  35  21  96   2]]

 [[  1  43  99  63 108]
  [ 94  80  68   9  49]
  [ 72   7  75  56 115]
  [112  64  30  95  34]
  [ 51 124  33 107   6]]

 [[ 37 116   3 109  50]
  [104  91  39   8  90]
  [ 70  20 125  59  16]
  [ 29  71  87  24 114]
  [ 57  28  67 113  48]]]
```

6.3.3. Nilai objective function akhir yang dicapai = 1627.0

6.3.4. Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati



6.3.5. Durasi proses pencarian = 101.56 detik

Berdasarkan hasil pengamatan saya dari eksperimen diatas, saya bisa langsung mendapatkan dua wawasan penting. Pertama, dari jumlah iterasi dan populasi untuk setiap eksperimen, bisa dilihat bahwa nilai *objective function* akhir yang dicapai semakin mendekati *global optimum* ketika 2 variabel tersebut jumlahnya ditingkatkan. Kedua, peningkatan jumlah iterasi dan populasi ini juga menyebabkan durasi proses pencarian semakin bertambah. Hal ini disebabkan karena beban komputasi yang harus diemban oleh komputer semakin banyak ketika 2 variabel tersebut meningkat.

Lalu, dari setiap 3 percobaan untuk semua variasi, hasil yang didapatkan tidak memiliki selisih yang terlalu signifikan. Hal ini menunjukkan bahwa algoritma memiliki tingkat konsistensi yang baik. Baiknya konsistensi ini menandakan bahwa algoritma bekerja dengan stabil dalam lingkungan eksperimen. Namun, konsistensi ini tidaklah selalu terjadi di setiap proses. Dalam eksperimen dengan jumlah populasi kontrol (1000) dan banyak iterasi 125, durasi proses pencarian menunjukkan variasi yang mencolok. Pada percobaan pertama dan kedua, durasi yang dihasilkan adalah 311.33 detik dan 315.92 detik. Namun, pada percobaan ketiga, durasi berubah drastis menjadi 190.61 detik. Dari kejadian ini, dapat disimpulkan bahwa konsistensi eksperimen dapat terganggu oleh faktor eksternal, terutama performa komputer dalam menjalankan komputasi.

Pada eksperimen di algoritma ini, nilai *objective function* akhir yang paling mendekati *state* sempurna terletak pada eksperimen dengan jumlah populasi kontrol (1000) dan banyak iterasi 125 di percobaan kedua, dengan nilai sebesar 1183. Hasil ini tercapai berkat jumlah iterasi yang tinggi, yang memberi algoritma lebih banyak waktu dan kesempatan untuk mencari solusi yang lebih mendalam dan terus menyempurnakannya. Dengan 125 iterasi, algoritma dapat menemukan solusi yang lebih optimal, menghasilkan nilai fungsi objektif akhir yang lebih rendah dan mendekati *state* sempurna. Hal yang sama tidak terjadi pada percobaan dengan iterasi rendah, dimana nilai *objective function* akhir yang paling jauh dicapai pada eksperimen dengan jumlah populasi kontrol (1000) dan banyak iterasi 3, yaitu sebesar 4615.0 pada percobaan kedua.

Pengaruh populasi juga terlihat jelas melalui data eksperimen ini. Dengan iterasi tetap di 50, populasi sebesar 50, nilai fungsi objektif dihasilkan sekitar 3068 pada percobaan kedua, sementara populasi yang lebih besar, seperti 2000, mampu menurunkannya hingga 1627. Hal ini menunjukkan bahwa populasi yang lebih besar membantu algoritma mengeksplorasi ruang pencarian lebih luas dan menghindari jebakan di optima lokal. Namun, penambahan populasi atau iterasi berlebihan tidak selalu memberikan peningkatan yang signifikan pada hasil akhir dan justru meningkatkan waktu komputasi. Oleh karena itu, perlu ada keseimbangan dalam jumlah iterasi dan populasi untuk mendapatkan hasil optimal tanpa membebani komputasi secara berlebihan.

KESIMPULAN DAN SARAN

Setiap algoritma memiliki hasil nilai *objective function* yang berbeda-beda antara Steepest Ascent Hill Climbing, Simulated Annealing, dan juga algoritma Genetic. Berdasarkan hasil skor dari percobaan yang kami lakukan, algoritma Steepest Ascent Hill Climbing memiliki hasil pencarian yang terbaik, diikuti dengan Simulated Annealing, lalu algoritma Genetic di posisi akhir dengan hasil terburuk dari ketiganya. Perbedaan hasil ini pasti dikarenakan dasar teori pencarian yang berbeda. Kemungkinan lainnya karena fungsi *magic cube* yang digunakan pada algoritma Steepest Ascent Hill Climbing dan Simulated Annealing sama, yang berbeda hanyalah algoritma Genetic. Fungsi *magic cube* ini menentukan fungsi pencarian nilai *objective function* sehingga nantinya juga akan mempengaruhi hasil *random initial state cube*.

Setiap algoritma *local search* memiliki sifat yang berbeda-beda dan unik. Untuk algoritma Steepest Ascent Hill Climbing, disetiap iterasinya hanya mencari nilai terbaik dari *initial staten*ya. Sedikit berbeda dengan algoritma Simulated Annealing, algoritma ini memungkinkan untuk mencari nilai *state* yang lebih buruk dari *initial state* nya sehingga memungkinkan ruang pencarian yang lebih luas. Untuk algoritma Genetic, ruang pencarian diperluas dalam bentuk populasi yang terus meningkat kualitasnya setiap generasi, namun sangat tergantung dari jumlah populasi dan generasi yang ada.

Perbandingan waktu memang bervariasi tapi tidak terlalu signifikan perbedaan waktunya. Untuk algoritma Genetic memiliki durasi proses pencarian tercepat, yaitu selama 5-6 detik, disusul oleh algoritma Simulated Annealing dengan durasi selama 8-9 detik, lalu algoritma Steepest Ascent Hill Climbing dengan durasi proses pencarian yang paling lama yaitu antara 20-27 detik.

Setiap algoritma memiliki nilai rentan konsistennya masing-masing. Untuk algoritma Steepest Ascent Hill Climbing memiliki nilai hasil yang konsisten dengan perbedaan sekitar 300-600an. Untuk algoritma Simulated Annealing memiliki nilai hasil yang konsisten dengan perbedaan sekitar 500-1000an. Untuk algoritma Genetic memiliki nilai hasil yang konsisten dengan sekitar 100-400.

PEMBAGIAN TUGAS SETIAP KELOMPOK

Nama anggota	**NIM**	**Tugas**
Samuel Franciscus Togar Hasurungan	18222131	- Implementasi algoritma - genetic.py - gencube.py (Magic Cube untuk genetic) - Laporan: - Pemilihan Objective Function - Penjelasan implementasi algoritma Genetic - Hasil eksperimen dan analisis algoritma genetic - README.md
Salsabilla Azzahra	18222139	- Implementasi proses algoritma simulated annealing - Laporan : - Penjelasan implementasi algoritma simulated annealing
Muhammad Reffy Haykal	18222103	- Implementasi Algoritma - Magiccube.py (Digunakan di Steepest Ascent Hill-Climbing dan Simulated Annealing) - SteepestHillClimb.py - visual.py - Laporan: - Penjelasan implementasi algoritma Magic Cube - Penjelasan implementasi algoritma Steepest Ascent Hill-Climbing
Hanan Fitra Salam	18222133	- Implementasi algoritma simulated annealing bagian plotting - Laporan : - Hasil eksperimen simulated annealing - Kesimpulan dan saran

REFERENSI

1. Features of the magic cube - vierkant
2. Perfect Magic Cubes (trump.de)
3. Magic cube - Wikipedia
4. Slide kuliah IF3070 Dasar Intelegensi Artifisial
5. Gambar 2