

**Dokumen Tugas Besar**  
**IF3070 Dasar Intelegensi Artifisial**



**Disusun Oleh:**

Jonathan Wiguna/18222019

Muhammad Yaafi Wasesa Putra/18222052

Matthew Nicholas Gunawan/18222058

Harry Truman Suhalim/18222081

**Program Studi Sistem dan Teknologi Informasi**  
**Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung**  
**Jl. Ganesha 10, Bandung 40132**

**2024**

## Daftar Isi

Daftar Isi.....	2
Deskripsi Persoalan.....	3
Pembahasan.....	4
• Objective Function.....	4
• Penjelasan Algoritma Local Search.....	7
➢ Steepest Ascent Hill Climbing.....	8
➢ Hill-Climbing with Sideways Move.....	10
➢ Random Restart Hill-Climbing.....	12
➢ Stochastic Hill-Climbing.....	16
➢ Simulated Annealing.....	20
➢ Genetic Algorithm.....	23
• Hasil Eksperimen dan Analisis.....	26
• Steepest Ascent Hill-Climbing.....	26
• Hill-Climbing with Sideways Move.....	28
• Random Restart Hill-Climbing.....	31
• Stochastic Hill-Climbing.....	35
• Simulated Annealing.....	38
○ Genetic Algorithm.....	44
• Kesimpulan dan Saran.....	56
• Pembagian Tugas.....	57
Referensi.....	58

## Deskripsi Persoalan

*Magic cube* adalah kubus yang tersusun dari angka 1 hingga  $n^3$  (dalam kasus kita, 1 hingga  $5^3$  atau 125) tanpa pengulangan,  $n$  dalam konteks ini adalah panjang sisi kubus. Angka-angka ini harus memenuhi ketentuan berikut :

- Terdapat satu angka yang disebut "magic number". *Magic number* ini tidak harus berada dalam rentang 1 hingga  $n^3$  dan tidak termasuk 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 (dari atas ke bawah) 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, kita akan bekerja dengan *magic cube* berukuran  $5 \times 5 \times 5$ . Initial state kubus adalah susunan acak angka 1 hingga 125. Kita akan menggunakan beberapa algoritma *local search* untuk menyelesaikan permasalahan ini. Pada setiap iterasi algoritma *local search*, kita hanya boleh menukar posisi 2 angka pada kubus (2 angka yang ditukar tidak harus bersebelahan). Khusus untuk genetic algorithm, diperbolehkan menukar posisi lebih dari 2 angka sekaligus dalam satu iterasi (tetapi tetap diperbolehkan juga hanya menukar 2 angka).

## Pembahasan

- **Objective Function**

*Objective function* sendiri bisa disebut adalah sebuah patokan yang akan digunakan untuk menilai apakah suatu *state* sudah mendekati *goal state* atau belum. Di tugas ini ada banyak *objective function* yang sebenarnya bisa digunakan, akan tetapi kelompok saya memutuskan untuk menggunakan *objective function* yang kami pikir cukup intuitif yaitu dengan menghitung seberapa banyak aspek *magic cube* yang sudah terpenuhi (jumlah baris, kolom, tiang, diagonal bidang, dan diagonal ruang) dengan *goal* akhir adalah mencapai 109. *Goal* akhir berupa angka 109 karena dalam *magic cube* terdapat  $n^2$  baris,  $n^2$  kolom,  $n^2$  tiang, 4 diagonal ruang, dan  $6n$  diagonal bidang. Karena kubus yang digunakan adalah  $5 \times 5 \times 5$  sehingga kita bisa mendapat total dari semua komponen berupa  $3(5^2) + 4 + 6.5 = 75 + 4 + 30 = 109$ . Jadi, kita akan mendapatkan *objective function* apabila 109 komponen tersebut telah bernilai sama seperti *magic\_number*.

PERTIMBANGAN UNTUK MERUBAH *OBJECTIVE FUNCTION* DI TENGAH Pengerjaan : Kami awalnya berpikir bahwa *objective function* dengan *goal state* semua komponen dari sebuah *cube* mencapai *magic number* adalah salah satu yang terbaik, tetapi setelah mencoba beberapa algoritma untuk masalah optimisasi ini (terutama ketika mengetes algoritma *Simulated Annealing*), kami menemukan *objective function* yang secara objektif lebih baik yaitu menghitung deviasi dari tiap komponen *cube* dibandingkan dengan *magic number*. Fungsi ini memiliki aspek-aspek positif yang sama seperti fungsi sebelumnya tetapi, ada satu alasan utama kami memutuskan untuk memilih fungsi ini dibandingkan fungsi sebelumnya yaitu karena dengan menggunakan deviasi-deviasi ini, hampir setiap iterasi yang dilakukan algoritma akan mencapai *better state*; Misal, ketika menggunakan fungsi awal, ketika ada sebuah baris yang sebenarnya sudah mencapai *near miss*, misal ketika dijumlahkan mencapai 314, hal ini tidak dipedulikan karena masih terhitung bahwa *correct components* dari *cube* ini belum bertambah, jauh berbeda dengan *objective function* yang menggunakan deviasi dimana ketika sebuah *state* naik 1 angka (dari SUM baris 310 menjadi 311) saja akan terhitung bahwa *state*-nya lebih baik dari sebelumnya.

Selanjutnya, sebelum memutuskan untuk menggunakan *objective function* ini, kami juga memikirkan perbandingan beban komputasi antara fungsi awal dan fungsi deviasi yang akhirnya kami gunakan. Kami menyimpulkan bahwa beban komputasinya hampir sama karena di kedua fungsi, di setiap iterasi, setiap komponen akan dihitung *value*-nya perkomponen, perbedaan utamanya hanyalah di fungsi pertama akan dicek apakah *SUM* dari komponen utama tersebut sudah sama dengan *magic number* atau belum, sedangkan untuk *objective function* kedua, yang dicek adalah deviasi atau absolut pengurangan *magic number* dengan SUM komponen yang sedang diiterasi; Penambahan beban komputasi hanyalah dari penghitungan deviasi yang tidak terlalu berarti.

```
def calculate_deviation(cube):
    total_deviation = 0

    def line_deviation(line_sum):
        return abs(line_sum - MAGIC_SUM)

    for i in range(5):
        for j in range(5):
            total_deviation += line_deviation(np.sum(cube[i, j, :])) #baris
            total_deviation += line_deviation(np.sum(cube[i, :, j])) #kolom
            total_deviation += line_deviation(np.sum(cube[:, i, j])) #pilar

    for i in range(5):
        total_deviation += line_deviation(np.trace(cube[i, :, :])) #trace = buat itung diagonal
        total_deviation += line_deviation(np.trace(np.fliplr(cube[i, :, :])) #diflip left to right, buat ngitung diagonal yg
        total_deviation += line_deviation(np.trace(cube[:, i, :]))
        total_deviation += line_deviation(np.trace(np.fliplr(cube[:, i, :]))
        total_deviation += line_deviation(np.trace(cube[:, :, i]))
        total_deviation += line_deviation(np.trace(np.fliplr(cube[:, :, i]))

    total_deviation += line_deviation(np.trace(cube.diagonal(axis1=0, axis2=1))) #diag 0,0,0 ke 5,5,5
    total_deviation += line_deviation(np.trace(np.fliplr(cube).diagonal(axis1=0, axis2=1))) #diag 0,0,5 ke 5,5,0
    total_deviation += line_deviation(np.trace(cube.diagonal(axis1=0, axis2=2))) #diag 0,5,0 ke 5,0,5
    total_deviation += line_deviation(np.trace(np.flipud(cube).diagonal(axis1=0, axis2=2))) #diag 0,5,5 ke 5,0,0
```

Fungsi `calculate_deviation` menggambarkan *objective function* yang digunakan. Dimulai dengan mendeklarasikan variabel `total_deviation` yang menggambarkan total deviasi dari sebuah cube, di-assign nilai awal 0. Selanjutnya penghitungan deviasi dari tiap baris, kolom, dan pilar dari magic cube, dihitung menggunakan double loop dimana di setiap loop akan dihitung komponennya menggunakan `sum` dari library `numpy`, Kodenya bekerja dengan misalkan `np.sum(cube[i, j, :])` maka akan diiterasi axis kolom dan baris, sedangkan tanda : berarti di axis tersebut akan diambil semua komponennya, di aspek ini berarti axis ke 3 diambil semua elemennya lalu dijumlahkan dan dikurangi `magic_sum` untuk mengetahui deviasi untuk line tersebut (menggunakan fungsi `line_deviation`). Sama

halnya untuk komponen lain. Selanjutnya untuk diagonal bidang, cara menghitungnya adalah mengambil bidang horizontal, vertikal, dan depth (ada 15 bidang) bidang diambil menggunakan `cube[i, :, :]`, dst, dimana selanjutnya diagonal bidang dihitung menggunakan fungsi `np.trace` dan `np.fliplr` (untuk mendapatkan diagonal yang berseberangan di bidang yang sama). Selanjutnya diagonal ruang akan dihitung menggunakan `axis`, sebagai contoh, ketika `axis1=0` dan `axis2 = 1` berarti `axis` ke 3 akan di-ignore. Diagonal yang dihasilkan sudah terlampir di komen code. Terakhir, akan dikembalikan total nilai deviasi yang akan digunakan sebagai objective function dari sebuah state kubus.

- **Penjelasan Algoritma Local Search**

*Local search* adalah algoritma *search* dimana *path* menuju *goal* tidak relevan. Fokus dari *local search* adalah mencari *neighbor* dengan solusi terbaik, menilai solusi terbaik ini didasarkan dengan *objective function* yang dipilih. Ada 2 kondisi utama yang biasanya menggunakan *local search algorithm* yaitu untuk *scheduling* dan *optimization*. Berikut ini adalah metode-metode *local search* yang kami gunakan setelah algoritma-algoritma inisialisasi kubus ini.

```
def generate_cube():  
    numbers = list(range(1, 126))  
    random.shuffle(numbers)  
    return np.array(numbers).reshape((5, 5, 5))
```

Function `generate_cube` adalah fungsi yang melakukan *generate cube* dimana angka-angkanya sesuai dengan spesifikasi yaitu secara acak diinput ke dalam *magic cube* (dengan fungsi `reshape`) berupa angka 1 hingga 125 secara acak dengan menggunakan bantuan `shuffle`. Fungsi ini yang kemudian digunakan untuk inisialisasi untuk semua kasus *local search*.

```
def generate_neighbor(cube):  
    new_cube = np.copy(cube)  
    pos1 = tuple(random.randint(0, 4) for _ in range(3))  
    pos2 = tuple(random.randint(0, 4) for _ in range(3))  
    while pos1 == pos2:  
        pos2 = tuple(random.randint(0, 4) for _ in range(3))  
    new_cube[pos1], new_cube[pos2] = new_cube[pos2], new_cube[pos1]  
    return new_cube
```

Function `generate_neighbor` menerima *input* berupa *cube* yang telah diinisialisasi dan selanjutnya membuat konfigurasi *neighbor* dari sebuah kubus dengan melakukan penukaran 2 elemen kubus 1 kali. Penukaran ini dimulai dengan dilakukannya generate tuple dengan jumlah 3 angka yang menggambarkan posisi dari komponen cube yang akan ditukar, setelah digenerate-nya 2 posisi ini, akan dicek apakah posisinya sama atau tidak, apabila sama maka akan di-loop sampai kedua

posisi ini berbeda. Hal penting yang perlu diingat adalah sebelum menukar posisi dari 2 elemen ini, kubus awal harus di-*copy* untuk menghindari terjadinya perubahan pada kubus yang asli karena kubus asli ini sangat penting karena bertindak sebagai *initial state* dimana akan digunakan untuk nge-generate *neighbor-neighbor* lainnya(di case random restart). Setiap selesai penukaran ini, akan dilakukan *return cube* baru tersebut.

### ➤ Steepest Ascent Hill Climbing

*Steepest Ascent Hill Climbing* adalah sebuah algoritma pencarian *local search* yang berfungsi untuk meningkatkan nilai secara iteratif. Dalam algoritmanya, akan dilakukan pencarian secara terus-menerus untuk mendapat nilai *neighbor* yang lebih baik dari nilai yang disimpan sementara. Nilai sementara tersebut kemudian diganti menjadi nilai *neighbor* jika nilai *neighbor* tersebut lebih baik dan akan berhenti jika tidak ditemukan adanya *neighbor* yang lebih baik dari nilai sementara lagi. Penggunaan algoritma *local search steepest ascent hill climbing* rentan terjebak dalam *local maximum* yang membuat algoritma ini tidak menjamin tercapainya objektif yang berupa *global maximum*.

Berikut ini adalah code untuk fungsi *Steepest Ascent Hill-Climbing* :

```
import numpy as np
import time
from general_func import generate_cube, generate_neighbor, calculate_deviation, evaluate, plot_deviation

def steepest_ascent_hill_climbing(max_iterations=1000):
    # Inisialisasi cube
    current_cube = generate_cube()
    initial_cube = np.copy(current_cube)
    current_deviation = calculate_deviation(current_cube)

    # Untuk melakukan plot_deviation
    iterations = []
    deviations = []

    start_time = time.time() #catat waktu mulai
```

*Function* *steepest\_ascent\_hill\_climbing* menerima *parameter* berupa banyak iterasi maksimal yang ingin dilakukan. Dimulai dengan menginisialisasi *cube* dengan fungsi yang sudah dibuat pada *general\_func*. Setelah itu menyimpan *iterations* dan *deviations* untuk dilakukan plot serta mencatat waktu mulai.



```

# loop utama
iteration = 0
while iteration < max_iterations:
    iterations.append(iteration)
    deviations.append(current_deviation)

    # Inisialisasi tetangga terbaik
    best_neighbor = None
    best_neighbor_deviation = current_deviation

    # loop untuk mendapat nilai tetangga yang lebih baik
    for i in range(max_iterations):
        neighbor_cube = generate_neighbor(current_cube)
        neighbor_deviation = calculate_deviation(neighbor_cube)

        if neighbor_deviation < best_neighbor_deviation:
            best_neighbor = neighbor_cube
            best_neighbor_deviation = neighbor_deviation

    # kondisi jika telah mencapai local optimum
    if best_neighbor is None or best_neighbor_deviation >= current_deviation:
        print(f"Local optimum reached at iteration {iteration}")
        break

    # Pindah ke tetangga yang lebih baik
    current_cube = best_neighbor
    current_deviation = best_neighbor_deviation

```

Setelah itu, dilakukan sebuah *loop* utama yang menghitung jumlah iterasi yang dilakukan dan akan berhenti jika iterasi telah mencapai nilai yang ditentukan pada parameter. Kemudian diinisialisasi tetangga terbaik dan juga nilai deviasinya, lalu dilakukan *search* menggunakan fungsi yang ada pada *general\_func* untuk membuat *cube* tetangga yang dibandingkan deviasinya dengan *best\_neighbor\_deviation* (deviasi yang diambil dari *current\_deviation*) untuk mendapat nilai *cube* terbaik (deviasi paling kecil). Jika tidak ada nilai deviasi yang lebih kecil dari nilai deviasi *current\_deviation* berarti *local optimum* telah dicapai, ketika belum dicapai *local optimum* maka nilai yang didapat tersebut akan di assign pada *current\_cube* dan *current\_deviation*.

```

# Menampilkan progress setiap 10 iterasi
if iteration % 10 == 0:
    print(f"Iteration {iteration}: Current deviation = {current_deviation}")

# kondisi jika telah ditemukan solusi
if current_deviation == 0:
    print("Perfect magic cube found!")
    break

iteration += 1 # menambah iterasi

duration = time.time() - start_time

# menggunakan fungsi evaluate untuk menunjukkan cube awal, cube akhir, final obj function, dan durasi
evaluate(initial_cube, current_cube, current_deviation, duration)

# melakukan plot berdasarkan iterasi dan deviasi yang telah dikumpulkan
plot_deviation(iterations, deviations)

return current_cube, current_deviation

```

*Code* ini menampilkan hasil yang didapat setiap 10 iterasi berlangsung dan akan berhenti jika *current\_deviation* bernilai 0 yang berarti *magic cube* telah sempurna. *Code* ini juga menggunakan fungsi *evaluate* dan *plot\_deviation* untuk menampilkan kondisi awal *cube*, kondisi akhir *cube*, nilai deviasi terbaik, waktu melakukan *search*, dan *plot iterations* dan *deviations* yang nilainya telah disimpan sebelumnya.

### ➤ Hill-Climbing with Sideways Move

*Hill-Climbing with Sideways Move* adalah algoritma *hill climbing* yang mirip dengan *steepest ascent hill climbing* tetapi dapat bergerak ke *neighbor* yang bernilai sama, tidak hanya ke *neighbor* yang lebih baik. Dengan algoritma ini, kemungkinan untuk mencapai *goal state* pun semakin meningkat karena dapat menemukan solusi dalam lebih banyak kasus dan kemungkinan terjebak dalam *local maximum* menurun. Walaupun dapat menghindari *local*, algoritma ini juga memiliki kekurangan seperti lebih membutuhkan banyak langkah sehingga akan lebih lambat dalam mencapai solusi dan tetap ada kemungkinan untuk terjebak pada *local maximum*.

Berikut ini adalah code untuk fungsi *Steepest Ascent Hill-Climbing* :

```

import numpy as np
import time
from general_func import generate_cube, generate_neighbor, calculate_deviation, evaluate, plot_deviation

def hill_climbing_with_sideways(max_iterations=1000, max_sideways=100):
    # Inisialisasi cube
    current_cube = generate_cube()
    initial_cube = np.copy(current_cube)
    current_deviation = calculate_deviation(current_cube)

    # Untuk melakukan plot_deviation
    iterations = []
    deviations = []

    start_time = time.time() #catat waktu mulai

```

Berbeda dengan *steepest ascent*, *function* `hill_climbing_with_sideways` menerima 2 parameter yaitu jumlah iterasi maksimal dan juga jumlah gerakan *sideways* maksimal. *Code* ini sama seperti *steepest ascent* dimana dimulai dengan inisialisasi *cube* dari *function* yang telah dibuat di `general_func`, lalu *iterations* dan *deviation* yang dilakukan akan disimpan serta dilakukan pencatatan waktu mulai.

```

# loop utama
iteration = 0
sideways_moves = 0
while iteration < max_iterations:
    iterations.append(iteration)
    deviations.append(current_deviation)

    # Inisialisasi tetangga terbaik
    best_neighbor = None
    best_neighbor_deviation = current_deviation

    # loop untuk mendapat nilai tetangga yang lebih baik
    for i in range(max_iterations):
        neighbor_cube = generate_neighbor(current_cube)
        neighbor_deviation = calculate_deviation(neighbor_cube)

        if neighbor_deviation < best_neighbor_deviation:
            best_neighbor = neighbor_cube
            best_neighbor_deviation = neighbor_deviation
        elif neighbor_deviation == best_neighbor_deviation: #kondisi untuk bergerak sideways di mana nilai neighbor sama
            best_neighbor = neighbor_cube
            sideways_moves += 1
            if sideways_moves >= max_sideways:
                print(f"Maximum number of sideways moves reached at iteration {iteration}")
                break

    # kondisi jika telah mencapai local optimum
    if best_neighbor is None or best_neighbor_deviation >= current_deviation:
        print(f"Local optimum reached at iteration {iteration}")
        break

    # Pindah ke tetangga yang lebih baik
    current_cube = best_neighbor
    current_deviation = best_neighbor_deviation
    sideways_moves = 0 # reset counter untuk sideways ketika menemukan solusi yang lebih baik

```

Selain jumlah iterasi yang dihitung, jumlah *sideways* juga dihitung. Dibuat *loop* utama yang menjaga iterasi tidak melebihi nilai maksimal yang ditentukan, lalu diinisialisasi tetangga terbaik dan dilakukan *looping* untuk mendapat nilai deviasi tetangga yang lebih kecil atau sama dengan nilai deviasi yang disimpan sementara

pada `best_neighbor_deviation`. Untuk kondisi nilai deviasi lebih kecil dari nilai deviasi yang disimpan maka akan di *assign* nilai tetangga sebagai `best_neighbor` dan `best_neighbor_deviation` dan apabila nilai deviasi sama dengan yang disimpan, dapat di *assign* sebagai `best_neighbor` dengan kondisi gerakan *side ways* masih kurang dari `max_sideways` yang telah ditentukan. Jika tidak ada nilai deviasi yang lebih baik dari nilai deviasi `current_deviation` berarti *local optimum* telah dicapai, ketika belum dicapai *local optimum* maka nilai yang didapat tersebut akan di *assign* pada `current_cube` dan `current_deviation` dan *counter sideways* akan di *reset*.

```
# Menampilkan progress setiap 10 iterasi
if iteration % 10 == 0:
    print(f"Iteration {iteration}: Current deviation = {current_deviation}")

# kondisi jika telah ditemukan solusi
if current_deviation == 0:
    print("Perfect magic cube found!")
    break

iteration += 1 # menambah iterasi

duration = time.time() - start_time

# menggunakan fungsi evaluate untuk menunjukkan cube awal, cube akhir, final obj function, dan durasi
evaluate(initial_cube, current_cube, current_deviation, duration)

# melakukan plot berdasarkan iterasi dan deviasi yang telah dikumpulkan
plot_deviation(iterations, deviations)

return current_cube, current_deviation
```

*Code* ini menampilkan hasil yang didapat setiap 10 iterasi berlangsung dan akan berhenti jika `current_deviation` bernilai 0 yang berarti *magic cube* telah sempurna. *Code* ini juga menggunakan fungsi `evaluate` dan `plot_deviation` untuk menampilkan kondisi awal *cube*, kondisi akhir *cube*, nilai deviasi terbaik, waktu melakukan *search*, dan *plot iterations* dan *deviations* yang nilainya telah disimpan sebelumnya.

### ➤ Random Restart Hill-Climbing

*Random Restart Hill-Climbing* juga merupakan salah satu variasi dari *hill climbing* yang jika gagal, maka akan mengulang algoritma berkali-kali hingga ditemukan solusi terbaik. Cara kerjanya adalah ketika tidak tercapainya *goal state* yang dicari maka akan direset ke kondisi acak dan diulang lagi. Algoritma ini sangat efektif dalam menyelesaikan masalah yang sulit dan semakin banyak jumlah *restart* maka kemungkinan mencapai objektif pun semakin besar pula. Selain memberikan solusi yang efektif dalam menyelesaikan masalah, penggunaan *restart* juga memiliki

beberapa kekurangan karena ketika menghadapi kasus yang membutuhkan jumlah *restart* yang sangat banyak tentu akan berpengaruh pada hasil dan waktu eksekusi apabila dijalankan ulang.

Berikut ini adalah code untuk fungsi *Steepest Ascent Hill-Climbing* :

```
import numpy as np
import time
from general_func import generate_cube, generate_neighbor, calculate_deviation, evaluate, plot_deviation

def random_restart_hill_climbing(max_iterations=1000, max_restart=5):
    best_cube = None
    best_deviation = float('inf')

    # Untuk melakukan plot_deviation
    iterations = []
    deviations = []

    # Melakukan tracking terhadap iterasi tiap restart
    attempt_iterations = []

    start_time = time.time()
    restart_count = 0
    total_iterations = 0
```

Dalam algoritma `random_restart_hill_climbing` menerima 2 buah parameter berupa iterasi maksimal yang dapat dilakukan (`max_iterations`) dan *restart* maksimal yang dapat dilakukan (`max_restart`). Lalu diinisialisasi `best_cube` dan juga `best_deviation`, diinisialisasi juga `iterations` dan `deviations` untuk menyimpan nilai yang akan diplot. Kemudian dibuat `attempt_iterations` untuk melakukan tracking iterasi tiap *restart* disimpan juga waktu mulai dan dibuat *counter* untuk iterasi dan *restart*.

```

# Loop untuk restart
while restart_count <= max_restart:
    if restart_count == 0:
        print("\nInitial attempt")
    else:
        print(f"\nRestart {restart_count}")

    # Inisialisasi cube baru untuk setiap percobaan
    current_cube = generate_cube()
    if restart_count == 0:
        initial_cube = np.copy(current_cube)

    current_deviation = calculate_deviation(current_cube)
    local_iterations = 0

    # loop utama untuk setiap percobaan
    while local_iterations < max_iterations:
        iterations.append(total_iterations)
        deviations.append(current_deviation)

        # Inisialisasi tetangga terbaik
        best_neighbor = None
        best_neighbor_deviation = current_deviation

        # loop untuk mendapat nilai tetangga yang lebih baik
        for i in range(max_iterations):
            neighbor_cube = generate_neighbor(current_cube)
            neighbor_deviation = calculate_deviation(neighbor_cube)

            if neighbor_deviation < best_neighbor_deviation:
                best_neighbor = neighbor_cube
                best_neighbor_deviation = neighbor_deviation

```

Dilakukan *looping* untuk *restart*, dimana akan diinisialisasi *cube* baru yang akan digunakan di tiap *restart* dan diinisialisasi juga *current\_deviation* dan *counter* untuk iterasi yang dilakukan. Lalu dibuat *loop* utama yang memasukkan nilai iterasi dan deviasi ke dalam *iterations* dan *deviations*, dilakukan juga inisialisasi untuk *best\_neighbor* dan *best\_neighbor\_deviation* yang kemudian dilakukan *looping* untuk membandingkan nilai yang disimpan sementara dengan nilai tetangga untuk mencari deviasi terkecil.

```

# kondisi jika telah mencapai local optimum
if best_neighbor is None or best_neighbor_deviation >= current_deviation:
    if restart_count == 0:
        print(f"Local optimum reached at iteration {local_iterations} (initial attempt)")
    else:
        print(f"Local optimum reached at iteration {local_iterations} (restart {restart_count})")
    break

# Pindah ke tetangga yang lebih baik
current_cube = best_neighbor
current_deviation = best_neighbor_deviation

# Mengubah nilai terbaik jika ditemukan deviasi lebih rendah
if current_deviation < best_deviation:
    best_cube = np.copy(current_cube)
    best_deviation = current_deviation

# Menampilkan progress setiap 10 iterasi
if local_iterations % 10 == 0:
    print(f"Iteration {local_iterations}: Current deviation = {current_deviation}")

# kondisi jika telah ditemukan solusi
if current_deviation == 0:
    print("Perfect magic cube found!")
    attempt_iterations.append(local_iterations + 1)
    duration = time.time() - start_time

# Memberikan output dari search yang telah dilakukan
print("\nSearch Summary:")
print(f"Number of restarts: {restart_count}") # Tidak menghitung initial attempt
print("Initial attempt:", attempt_iterations[0], "iterations")
if len(attempt_iterations) > 1:
    print("Iterations per restart:", attempt_iterations[1:])
    print(f"Average iterations per restart: {sum(attempt_iterations[1:]) / len(attempt_iterations[1:]):.2f}")

```

Dibuat juga kondisi jika tidak ada nilai deviasi tetangga yang lebih kecil (tercapainya *local optimum*) dan jika belum tercapainya *local optimum* nilai yang didapat di *assign* menjadi *current\_cube* dan *current\_deviation*. Ketika ditemukan *current\_deviation* yang lebih kecil dari *best\_deviation* maka nilai *current\_deviation* tersebut akan di *assign* menjadi *best\_deviation* dan *cube* tersebut akan menjadi *best\_cube*. *Code* ini menampilkan hasil yang didapat setiap 10 iterasi berlangsung dan akan berhenti jika *current\_deviation* bernilai 0 atau *magic cube* telah sempurna, ditampilkan pula *search summary* yang menampilkan jumlah restart, jumlah iterasi, dan rata-rata iterasi tiap *restart*.

```

        # menggunakan fungsi evaluate untuk menunjukkan cube awal, cube akhir, final obj function, dan durasi
        evaluate(initial_cube, best_cube, best_deviation, duration)

        # melakukan plot berdasarkan iterasi dan deviasi yang telah dikumpulkan
        plot_deviation(iterations, deviations)

        return best_cube, best_deviation

    local_iterations += 1
    total_iterations += 1

    attempt_iterations.append(local_iterations)
    restart_count += 1

duration = time.time() - start_time

# Memberikan output dari search yang telah dilakukan
print("\nSearch Summary:")
print(f"Number of restarts: {restart_count - 1}")
print("Initial attempt:", attempt_iterations[0], "iterations")
print("Iterations per restart:", attempt_iterations[1:])
if len(attempt_iterations) > 1:
    print(f"Average iterations per restart: {sum(attempt_iterations[1:]) / len(attempt_iterations[1:]):.2f}")

# menggunakan fungsi evaluate untuk menunjukkan cube awal, cube akhir, final obj function, dan durasi
evaluate(initial_cube, best_cube, best_deviation, duration)

# melakukan plot berdasarkan iterasi dan deviasi yang telah dikumpulkan
plot_deviation(iterations, deviations)

return best_cube, best_deviation

```

*Code* ini juga menggunakan fungsi `evaluate` dan `plot_deviation` yang dibuat pada `general_func` untuk menampilkan kondisi awal *cube*, kondisi akhir *cube*, deviasi terbaik, durasi *search*, dan *plot* dari *iterations* dan *deviations* ketika menemukan *magic cube sempurna*. Disimpan juga `attempt_iterations` yang dilakukan lalu diberikan *output* yang sama dengan *magic cube* yang telah sempurna ketika *loop* berhenti karena jumlah *restart* telah mencapai nilai maksimal yang ditentukan.

## ➤ Stochastic Hill-Climbing

*Stochastic Hill-Climbing* adalah salah satu dari jenis *hill climbing* yang dapat memilih *neighbor* dengan acak dan berpindah ke *neighbor* tersebut berdasarkan nilainya. Jenis *hill climbing* ini berbeda dengan *steepest ascent hill climbing* karena algoritma ini tidak mencari yang terbaik tetapi mencari yang lebih baik secara acak. Titik henti dari algoritma ini adalah jika mencapai jumlah iterasi maksimal atau telah ditemukannya solusi. Algoritma ini akan lebih cepat setiap iterasinya karena tidak perlu melakukan pengecekan semua *neighbor* serta memiliki kapabilitas dalam menghindari *local maximum*. Kekurangan dari pemakaian algoritma *stochastic hill climbing* adalah bisa saja algoritma ini melewati jalur yang lebih baik dan tidak



konsisten akibat pemilihan yang bersifat acak yang dapat mengakibatkan perbedaan hasil tiap eksekusinya.

Berikut ini adalah code untuk fungsi Stochastic Hill-Climbing :

```
import numpy as np
import time
import random
from general_func import generate_cube, generate_neighbor, calculate_deviation, evaluate, plot_deviation

def stochastic_hill_climbing(max_iterations=1000):
    # Inisialisasi cube
    current_cube = generate_cube()
    initial_cube = np.copy(current_cube)
    current_deviation = calculate_deviation(current_cube)

    # Untuk melakukan plot_deviation
    iterations = []
    deviations = []

    start_time = time.time() #catat waktu mulai
```

Dalam algoritma `stochastic_hill_climbing` mirip seperti *steepest ascent* yang menerima parameter berupa iterasi maksimal yaitu `max_iterations`. Dilanjutkan dengan inisialisasi `cube` dengan fungsi dari `general_func` dan mencatat `iterations`, `deviations`, dan waktu mulai.

```

# loop utama
iteration = 0
while iteration < max_iterations:
    iterations.append(iteration)
    deviations.append(current_deviation)

    # Inisialisasi tetangga terbaik
    better_neighbors = []

    # loop untuk mendapat nilai tetangga yang lebih baik
    for i in range(max_iterations):
        neighbor_cube = generate_neighbor(current_cube)
        neighbor_deviation = calculate_deviation(neighbor_cube)

        if neighbor_deviation < current_deviation:
            better_neighbors.append((neighbor_cube, neighbor_deviation))

    # kondisi jika telah mencapai local optimum
    if not better_neighbors:
        print(f"local optimum reached at iteration {iteration}")
        break

    # Memilih secara acak tetangga yang lebih baik
    chosen_neighbor, chosen_deviation = random.choice(better_neighbors)

    current_cube = chosen_neighbor
    current_deviation = chosen_deviation

```

Lalu dilakukan *loop* utama yang menjaga agar iterasi tidak melebihi batas maksimal yang ditentukan. *Loop* ini akan mencatat tiap iterasi dan deviasinya, dan menginisialisasi tetangga yang lebih baik, di mana setiap melakukan pencarian apabila ada tetangga yang lebih baik akan dimasukkan ke dalam `better_neighbors`. Jika tidak ditemukan tetangga yang lebih baik maka *local optimum* telah dicapai, setelah itu akan dipilih secara acak tetangga yang akan di *assign* sebagai `current_cube` dan `current_deviation`.

```

# Menampilkan progress setiap 10 iterasi
if iteration % 10 == 0:
    print(f"Iteration {iteration}: Current deviation = {current_deviation}")

# kondisi jika telah ditemukan solusi
if current_deviation == 0:
    print("Perfect magic cube found!")
    break

iteration += 1 # menambah iterasi

duration = time.time() - start_time

# menggunakan fungsi evaluate untuk menunjukkan cube awal, cube akhir, final obj function, dan durasi
evaluate(initial_cube, current_cube, current_deviation, duration)

# melakukan plot berdasarkan iterasi dan deviasi yang telah dikumpulkan
plot_deviation(iterations, deviations)

return current_cube, current_deviation

```

*Code* ini menampilkan hasil yang didapat setiap 10 iterasi berlangsung dan akan berhenti jika `current_deviation` bernilai 0 yang berarti *magic cube* telah sempurna. *Code* ini juga menggunakan fungsi `evaluate` dan `plot_deviation` untuk menampilkan kondisi awal *cube*, kondisi akhir *cube*, nilai deviasi terbaik, waktu melakukan *search*, dan *plot iterations* dan *deviations* yang nilainya telah disimpan sebelumnya.

## ➤ Simulated Annealing

ealing

### Simulated Annealing

```
function SimulatedAnnealing(problem)
  current=initial state of problem
  t=1 //initialize time
  loop
    T = coolDown(t) //converts time in Temp.
    if T=0 then return current
    next = random successor of current
     $\delta E$  = next.VALUE - current.VALUE
    if  $\delta E > 0$  then current=next
    else current=next with some prob ( $e^{\frac{\delta E}{T}}$ )
    t=t+1
```

$$e^{\frac{\delta E}{T}}$$

9:01 / 10:02



Fungsi *Simulated Annealing* ini mengambil konsep suhu; fungsi ini akan memiliki 3 *phase* yaitu “High Temperature” dimana fungsi akan mengecek berbagai macam *state* dari masalah dengan tingkat *random* yang tinggi, lalu *phase* selanjutnya “Cooling Down” dimana fungsi sudah mulai tidak terlalu sering memilih *state* yang tidak lebih baik dari *state* yang dimilikinya saat ini (ideally di tahap ini akan mulai menuju ke “gunung” tertinggi global maksimum), *phase* terakhir, “Cold”, *Temperature* di fungsi sudah sangat dekat ke 0 dan *state* yang lebih buruk dari *state* saat ini sangat kecil kemungkinannya untuk dipilih.

Kunci utama dari konsep ini adalah fungsi IF ELSE di gambar di atas dengan penjelasan bahwa ketika *state neighbor*nya tidak lebih baik dari *state* yang saat ini tetap masih ada potensi untuk dipilih, dengan probabilitasnya adalah  $e^{(\text{delta energi} / T)}$ , delta energi di sini adalah aspek dari *objective function* yaitu total deviasi dari *state* kubus sekarang dibandingkan *state neighbor*. Di saat ketika *state neighbor* lebih buruk maka delta e akan bernilai negatif dan juga sebaliknya. Selanjutnya e disini

adalah euler yang bernilai 2.7 sekian dan terakhir T adalah variabel yang menggambarkan temperatur. Ketika berada di tahap awal dimana temperatur masih bernilai tinggi maka nilai deviasi tidak terlalu besar pengaruhnya dikarenakan nilai deviasi tersebut akan dibagi dengan T sehingga probabilitas angka yang digenerate secara *random* dengan *range* 0-1 memiliki kemungkinan yang cukup besar untuk memenuhi kondisi IF sehingga *state* akan berpindah ke *state* yang lebih buruk (aspek random dari *simulated annealing*). Seiring berjalannya waktu temperatur akan diturunkan sedikit demi sedikit sehingga nilai T tersebut makin lama akan semakin tidak begitu mempengaruhi perhitungan probabilitas tersebut dan delta E akan memiliki *major effect* dalam menentukan nilai probabilitas, dikarenakan hal ini peluang untuk dipilihnya *state* yang lebih buruk akan semakin menurun dan diharapkan setelah beberapa iterasi ketika *state* mencapai temperatur 0 akan ditemukan global maksimum.

Berikut code untuk fungsi simulated annealing :

```
MAGIC_SUM = 315
MAX_ITERATIONS = 50000
INITIAL_TEMPERATURE = 100 #ini yg krusial buat diganti2
COOLING_RATE = 0.9999 #ini yg krusial buat diganti2

def simulated_annealing():
    initial_cube = generate_cube() #harus disimpen buat ditampilkan di akhir
    current_cube = np.copy(initial_cube)
    current_deviation = calculate_deviation(current_cube)
    best_cube = np.copy(current_cube)
    best_deviation = current_deviation

    temperature = INITIAL_TEMPERATURE
    start_time = time.time()

    #buat ngeplot
    deviations = []
    iterations = []
    entropies = []
    stuck_count = 0
```

initial temperature di 100 dan cooling rate di 0.9999, kedua komponen ini akan diganti-ganti untuk melihat bagaimana efeknya ke plot di bagian selanjutnya.

```

for iteration in range(MAX_ITERATIONS):
    neighbor_cube = generate_neighbor(current_cube)
    neighbor_deviation = calculate_deviation(neighbor_cube)

    delta_e = current_deviation - neighbor_deviation

    entropy = math.exp(delta_e / temperature)
    entropies.append(entropy)

    if neighbor_deviation < current_deviation or random.random() < entropy:
        current_cube = neighbor_cube # ^ klo deviationnya worse masi ada kemungkinan untuk pindah ke neighbornya
        current_deviation = neighbor_deviation

        if current_deviation < best_deviation:
            best_cube = np.copy(current_cube)
            best_deviation = current_deviation
    else:
        stuck_count += 1

```

Akan dilakukan iterasi sampai menyampai max iterasi, hal-hal terkait neighbor dan deviation sama saja seperti algoritma lain, yang paling penting adalah terkait delta\_e dan entropy. Fungsinya sendiri sudah dijelaskan di paragraf di atas. Frekuensi 'stuck' di local optima akan dihitung menggunakan variabel stuck\_count dimana akan bertambah ketika deviasi dari cube neighbor tidak lebih baik atau angka yang digenerate secara random tidak melebihi entropi.

```

#simpen
deviations.append(current_deviation)
iterations.append(iteration)

# temperature dikurangi sesuai cooling rate
temperature *= COOLING_RATE

if best_deviation == 0: #klo deviation uda 0 yang berarti perfect magic cube
    print("Perfect magic cube found!")
    break

# setiap 1000 iterasi print deviasi
if iteration % 1000 == 0:
    print(f"Iteration {iteration}: Current deviation = {current_deviation}")

# plot
duration = time.time() - start_time
evaluate(initial_cube, best_cube, best_deviation, duration)
print(f"'stuck' in local optimum: {stuck_count}")
plot_deviation(iterations, deviations, entropies)

simulated_annealing()

```

Terakhir, akan dilakukan plotting terhadap nilai-nilai yang ada. Disini, fungsi plot\_deviation menerima 3 parameter, tidak seperti algoritma lainnya dikarenakan di

bagian ini diminta untuk menampilkan juga Plot  $e^{\frac{\Delta E}{T}}$  terhadap banyak iterasi yang telah dilewati.

*Challenge* utama dari menggunakan algoritma ini agar menjadi salah satu yang terbaik adalah menentukan *cooling rate* dan *initial temperature* yang paling ideal untuk dipakai, dikarenakan apabila *cooling rate* terlalu cepat/ *initial temperature* terlalu rendah akan menyebabkan peluang terjebak di lokal maksimum lebih besar dan apabila *cooling rate* terlalu lambat/ *initial temperature* terlalu tinggi maka algoritma akan berjalan mirip dengan *random walk algorithm* dengan waktu yang cukup lama sehingga waktu komputasi akan meningkat secara cukup signifikan.

### ➤ Genetic Algorithm

Algoritma terakhir, *genetic algorithm* mengambil konsep evolusi makhluk hidup. 3 aksi utama dari algoritma ini adalah *selection*, *crossover*, dan *mutation*; 3 komponen itu akan menggantikan konsep pencarian *neighbor* di 5 algoritma *local search* sebelumnya. Perbedaan mayor lainnya adalah di *genetic algorithm* ada banyak *initial state* yang akan digunakan periterasi-nya, sedangkan *local search* sebelumnya hanya memiliki 1 *initial state*. Tujuan dimulainya menggunakan banyak *initial state* ini adalah agar 3 aksi utama *genetic algorithm* yang sebelumnya disebutkan dapat dijalankan, pertama, dari banyaknya *initial state* tersebut akan dilakukan seleksi *cube* dari banyaknya *cube* tersebut dan diambil *cube* dengan *state* terbaik, dalam konteks pengerjaan ini yang saya pilih adalah *cube* dengan nilai total deviasi terkecil. Selanjutnya akan dipilih *potential parents*, pemilihan ini ada berbagai macam cara seperti *Roulette Wheel Selection*, *Stochastic Universal Sampling* (Mirip sekali dengan *roulette* tapi ada 2 panah sehingga pasangan *parents* terpilih dalam 1 putaran), *Tournament Selection*, *Rank Selection*, dan *Random Selection*. Di algoritma kali ini saya menggunakan *Tournament Selection* karena menurut saya paling mudah untuk dinalar. Setelah ditemukannya *parents* yang berpasangan, akan dilakukan *crossover* dimana aspek-aspek terbaik dari tiap *cube* akan disatukan menjadi 1 *cube* baru, terakhir, dari tiap anak yang tercipta ada kemungkinan untuk terjadinya mutasi dengan tujuan apabila *crossover* tidak memunculkan anak yang *desired* maka diharapkan mutasi tersebut dapat membantu *cube* terkait mencapai *state* yang lebih baik dengan cara mempertahankan aspek *randomness* yang mungkin saja bisa membantu mencapai

*goal state*. Algoritma akan digunakan sampai beberapa generasi terbuat dan akhirnya tercipta generasi terakhir yang akan dilakukan seleksi untuk mendapatkan *cube* terbaik dari generasi-generasi yang sudah dijalankan.

Berikut merupakan kode dari fungsi genetic algorithm :

```
8 def crossover(parent1, parent2):
9     point = np.random.randint(1, 5)
10    child = np.copy(parent1)
11    child[:, :, point:] = parent2[:, :, point:]
12    return child
13
14 def mutate(cube):
15     i, j, k, i2, j2, k2 = np.random.randint(0, 5, 6)
16     cube[i, j, k], cube[i2, j2, k2] = cube[i2, j2, k2], cube[i, j, k]
17     return cube
```

fungsi crossover melakukan perkawinan silang antara dua parent kubus 5x5x5 untuk menghasilkan kubus baru. cara kerjanya adalah dengan memilih titik potong secara acak antara 1 dan 4, kemudian mengambil bagian kubus dari parent1 dari awal sampai titik potong, dan mengambil sisa bagian dari parent2 setelah titik potong tersebut.

fungsi mutate melakukan mutasi pada sebuah kubus dengan cara menukar posisi dua angka yang dipilih secara acak. fungsi ini memilih secara acak dia set koordinat (i,j,k) dan (i2,j2,k2) dalam kubus 5x5x5, kemudian menukar nilai yang berada pada kedua posisi tersebut.



```

def genetic_algorithm(population_size, max_iterations):
    cube_initial = generate_cube()
    current_deviation = calculate_deviation(cube_initial)

    def generate_initial_population(population_size):
        population = []
        for _ in range(population_size):
            cube = generate_cube()
            population.append(cube)
        return population

    population = generate_initial_population(population_size)
    best_scores = []
    avg_scores = []

    start_time = time.time()

    for iteration in range(max_iterations):
        population_scores = [calculate_deviation(cube) for cube in population]
        best_scores.append(min(population_scores))
        avg_scores.append(np.mean(population_scores))

        if (iteration + 1) % 50 == 0:
            print(f"Iteration {iteration + 1}: Current Objective Function Value = {best_scores[-1]}")

        selected = [population[i] for i in np.argsort(population_scores)[:population_size // 2]]

        new_population = []
        for _ in range(population_size // 2):
            idx1, idx2 = np.random.choice(len(selected), 2, replace=False)

            new_population = []
            for _ in range(population_size // 2):
                idx1, idx2 = np.random.choice(len(selected), 2, replace=False)
                parent1, parent2 = selected[idx1], selected[idx2]

                for attempt in range(5):
                    child = crossover(parent1, parent2)
                    if not np.array_equal(child, parent1) and not np.array_equal(child, parent2): # agar child tidak sama dengan parent 1 & 2
                        break
                else:
                    child = mutate(child)

                if np.random.rand() < 0.2:
                    child = mutate(child)
                # cek apakah child sam dengan kedua parent
                new_population.append(child)

            population = selected + new_population

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

        final_population_scores = [calculate_deviation(cube) for cube in population]
        best_idx = np.argmin(final_population_scores)
        cube_final = population[best_idx]
        final_score = final_population_scores[best_idx]

        print("\n" + "="*50)
        print("HASIL AKHIR")
        print("="*50)

        evaluate(cube_initial, cube_final, final_score, duration)

    return cube_initial, cube_final, best_scores, avg_scores, duration, final_score

```

Fungsi `genetic_algorithm` merupakan fungsi yang menjalankan algoritma genetik untuk mencari solusi optimal atau mendekati optimal dari magic cube dengan iterasi yang ditentukan oleh parameter `max_iteration`. Dimulai dengan inisialisasi populasi awal yang di generate secara acak, kemudian menjalankan sejumlah iterasi seleksi dan reproduksi. Pada setiap iterasi, nilai fitness (deviasi) setiap individu dihitung untuk menentukan kualitas solusi. setengah dari populasi dengan nilai fitness terbaik akan dipertahankan melalui proses seleksi. populasi baru kemudian dibentuk melalui proses

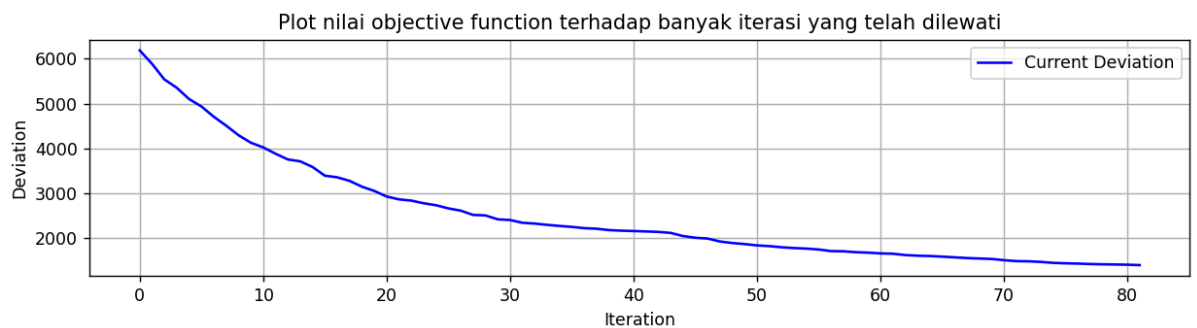
crossover antara individu yang terpilih, dengan probabilitas 20%. Proses ini berlanjut sampai jumlah iterasi maksimum yang ditentukan.

## • Hasil Eksperimen dan Analisis

### • Steepest Ascent Hill-Climbing

Berikut ini adalah hasil *plot* dari *steepest ascent hill-climbing* beserta hasil yang didapat dari *search* yang dilakukan. Eksperimen yang dilakukan adalah mengubah parameter berupa jumlah maksimal iterasi yang dapat dilakukan.

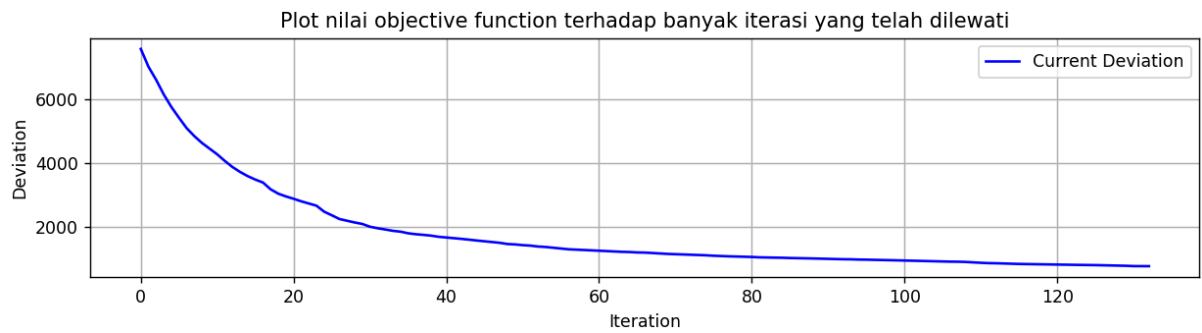
A.  $\text{max\_iterations} = 100$



Final Obj Func: 1404  
Duration: 3.693 s

<p>Initial State:</p> <pre>[[[122 65 114 81 2]  [ 72 79 99 59 14]  [ 16 60 6 124 20]  [ 95 43 32 63 113]  [ 87 26 19 21 29]]  [[ 34 46 33 68 30]  [ 73 37 39 112 44]  [ 12 56 83 80 123]  [ 85 67 69 82 106]  [ 42 120 31 28 88]]  [[108 35 86 109 92]  [ 47 125 104 45 71]  [110 70 18 4 107]  [ 89 116 55 25 101]  [ 27 5 13 74 11]]  [[102 15 103 23 64]  [ 1 9 118 57 117]  [ 62 53 17 121 7]  [ 54 93 111 76 3]  [ 77 91 75 100 24]]  [[ 98 36 41 119 66]  [ 58 49 51 50 10]  [ 78 61 96 84 48]  [ 22 90 105 8 97]  [ 94 115 40 52 38]]]</pre>	<p>Final State:</p> <pre>[[[ 55 65 112 2 81]  [104 103 35 59 14]  [ 28 77 46 124 41]  [ 71 43 16 68 113]  [ 84 21 98 51 66]]  [[ 34 122 33 74 48]  [ 83 37 36 114 44]  [ 94 45 73 10 121]  [ 95 62 63 82 12]  [ 11 78 106 32 88]]  [[ 5 20 86 109 92]  [ 47 116 97 56 3]  [ 79 54 67 4 107]  [123 89 60 25 22]  [ 27 53 18 111 101]]  [[115 9 110 23 64]  [ 1 15 85 57 117]  [ 76 120 26 91 7]  [ 17 70 69 118 42]  [108 102 13 80 24]]  [[ 93 99 6 119 29]  [ 75 58 61 30 72]  [ 39 19 96 87 50]  [ 8 52 105 31 125]  [100 90 40 49 38]]]</pre>
---	--

B. `max_iterations = 1000`



Final Obj Func: 787  
Duration: 44.487 s

Initial State:	Final State:
[[ 62 25 56 83 37]	[[ 31 108 32 83 55]
[ 46 100 124 111 75]	[ 48 92 38 59 80]
[ 89 113 16 105 47]	[ 89 74 97 4 52]
[ 64 9 85 22 29]	[ 64 27 22 85 117]
[ 58 28 5 79 17]]	[ 77 15 125 86 11]]
[[ 94 36 106 53 68]	[[ 94 39 120 45 18]
[ 18 57 49 120 65]	[ 20 63 54 109 58]
[ 88 71 101 66 110]	[121 9 71 66 46]
[ 21 26 82 2 107]	[ 68 103 10 19 110]
[ 14 93 24 92 81]]	[ 14 93 56 76 81]]
[[117 122 60 118 50]	[[ 33 1 118 114 50]
[ 34 96 7 61 67]	[124 96 12 5 78]
[ 74 31 80 77 44]	[ 87 36 75 17 101]
[ 4 116 70 19 54]	[ 6 116 70 67 57]
[ 33 6 27 114 20]]	[ 69 65 40 113 28]]
[[ 98 78 43 102 72]	[[106 51 44 24 90]
[ 73 41 90 11 8]	[ 25 30 112 119 29]
[121 52 3 104 69]	[ 16 115 8 105 79]
[ 95 32 97 51 10]	[ 95 26 111 62 21]
[ 38 115 39 108 91]]	[ 73 107 41 3 91]]
[[ 30 1 40 59 99]	[[ 47 122 2 49 100]
[103 35 86 119 12]	[ 99 34 98 23 61]
[ 84 76 63 123 45]	[ 7 88 60 123 37]
[ 87 42 112 23 13]	[ 84 42 102 72 13]
[125 15 55 48 109]]]	[ 82 35 53 43 104]]]

Pembahasan : berdasarkan *plot* yang telah dilakukan dapat dilihat bahwa untuk 20 iterasi pertama terjadi perubahan signifikan terhadap nilai deviasi dan kemudian mulai mendatar pada iterasi 40 hingga seterusnya, hal ini berlaku untuk nilai maksimal iterasi 100 maupun 1000.

Kemudian terkait lamanya durasi pencarian itu bergantung pada maksimal iterasi yang ditentukan karena *looping* mencari tetangga yang terbaik itu berdasarkan nilai `max_iterations`. Semakin besar nilai `max_iterations` maka akan semakin besar pula waktu pencarian. Namun, besar kecilnya `max_iterations` juga mempengaruhi hasil yang didapat bisa dilihat hasilnya akan lebih maksimal ketika nilai

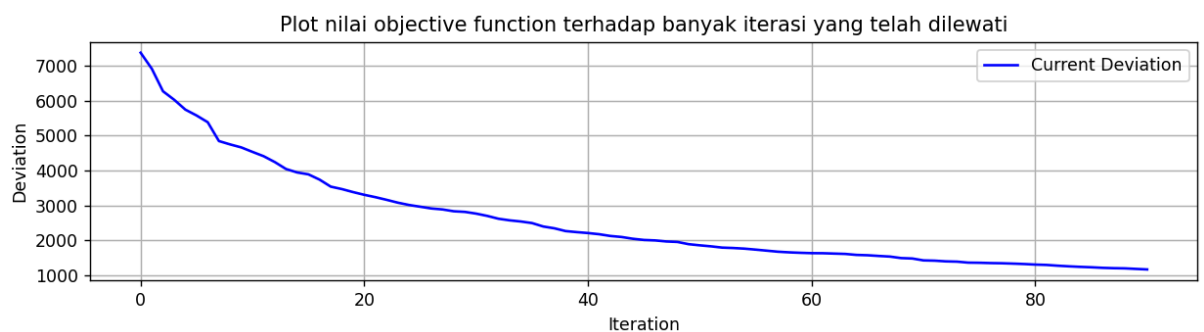
max\_iterationss lebih besar dan dapat kita simpulkan bahwa semakin besar nilai max\_iterations maka akan semakin kecil nilai deviasi yang berarti semakin mendekati *magic cube* yang sempurna. Walaupun begitu, algoritma *steepest ascent* sangat mudah untuk terjebak dalam *local optimum*.

Apabila dibandingkan dengan *local search* lainnya, bisa dikatakan bahwa *steepest ascent hill-climbing* bukanlah yang paling optimal jika ingin mencari global optima. Hal ini karena, berdasarkan eksperimen walaupun telah ditetapkan iterasi maksimal sebesar 1000 tetap sulit dalam mencapai global optima dilihat dari grafik walaupun telah diatur maksimal iterasi yang dilakukan adalah 1000 tetap berhenti di iterasi 126 dan nilai deviasi masih di 700an yang tergolong jauh dari global optima.

- **Hill-Climbing with Sideways Move**

Berikut ini adalah hasil *plot* dari *hill-climbing with sideways move* beserta hasil yang didapat dari *search* yang dilakukan. Eksperimen yang dilakukan adalah mengubah parameter berupa jumlah maksimal iterasi yang dapat dilakukan.

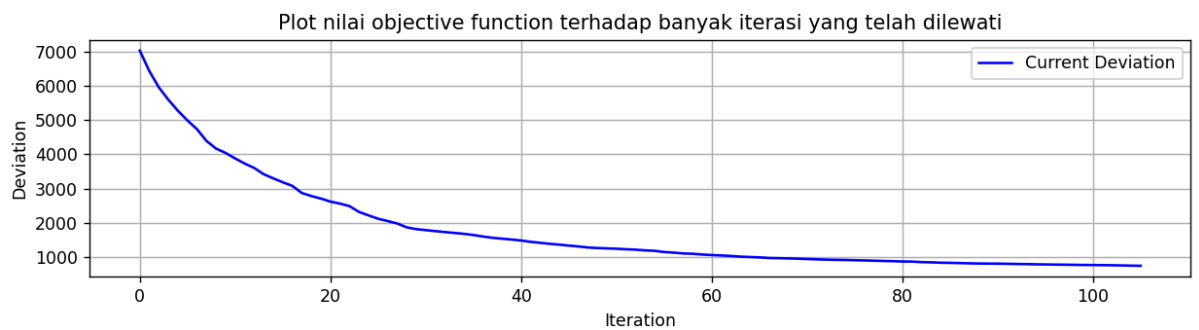
A. max\_iterations = 100



Final Obj Func: 1171  
Duration: 3.947 s

Initial State:	Final State:
[[[ 17 44 24 8 92]	[[[ 17 16 46 121 104]
[ 86 23 107 4 36]	[ 86 88 107 8 36]
[ 89 82 53 112 118]	[ 91 34 11 57 118]
[ 45 58 65 72 6]	[ 51 113 32 115 1]
[ 43 52 117 55 59]]	[ 43 65 117 14 80]]
[[ 91 96 70 50 32]	[[ 92 96 48 26 52]
[104 31 28 14 73]	[112 29 28 119 31]
[ 33 115 20 48 67]	[ 2 82 102 70 67]
[101 1 100 22 87]	[ 93 15 79 22 103]
[ 37 97 113 68 56]]	[ 33 97 58 68 56]]
[[ 3 69 83 110 79]	[[ 12 98 89 54 60]
[119 120 71 105 106]	[ 37 120 30 7 122]
[ 74 46 85 47 5]	[ 94 55 85 73 5]
[ 90 54 122 66 102]	[ 47 35 106 66 63]
[ 2 15 116 57 64]]	[124 4 6 116 64]]
[[ 49 61 18 26 21]	[[111 50 125 10 21]
[ 13 29 30 10 62]	[ 13 45 75 108 62]
[123 109 51 39 125]	[123 18 24 39 114]
[ 93 78 38 99 94]	[ 19 78 59 87 74]
[ 80 114 108 7 124]]	[ 72 109 20 83 42]]
[[ 98 111 19 60 121]	[[ 84 49 3 101 77]
[ 40 95 42 34 11]	[ 69 38 95 53 61]
[ 25 88 81 41 9]	[ 25 110 81 90 9]
[ 35 76 77 12 16]	[105 76 40 23 71]
[ 84 75 63 27 103]]]	[ 44 41 100 27 99]]]

B. max\_iterations = 1000



Final Obj Func: 746  
Duration: 43.311 s

Initial State:	Final State:
[[[ 46 114 24 14 23]	[[[ 50 114 22 105 24]
[ 13 8 4 101 31]	[ 8 13 68 101 125]
[ 67 11 105 36 66]	[106 32 107 5 65]
[ 79 18 108 76 102]	[ 89 21 75 88 54]
[ 73 26 33 20 3]]	[ 61 124 43 20 47]]
[[ 63 64 82 17 109]	[[111 17 45 64 77]
[113 120 41 2 88]	[ 34 115 85 3 79]
[ 5 70 58 39 35]	[ 19 70 57 83 86]
[ 71 100 111 50 99]	[ 29 16 112 60 84]
[121 72 65 57 115]]	[122 95 15 103 1]]
[[ 40 54 68 21 60]	[[ 37 35 100 96 51]
[ 98 53 38 69 49]	[108 49 36 69 53]
[ 95 19 83 93 123]	[ 72 14 74 78 73]
[ 86 77 55 1 44]	[ 94 117 55 31 18]
[ 85 94 119 42 103]]	[ 4 99 52 41 120]]
[[124 16 122 110 75]	[[ 48 56 119 26 66]
[104 106 6 125 12]	[104 67 9 121 12]
[ 56 96 48 52 9]	[110 97 38 58 11]
[ 51 28 97 87 117]	[ 44 82 33 46 109]
[ 92 25 107 7 89]]	[ 10 6 118 63 116]]
[[ 32 118 10 27 80]	[[ 71 93 28 25 98]
[ 62 78 61 15 22]	[ 62 76 123 23 27]
[ 59 112 37 91 116]	[ 7 102 39 91 80]
[ 74 29 43 90 81]	[ 59 42 40 92 81]
[ 34 47 84 45 30]]]	[113 2 90 87 30]]]

Pembahasan : Berdasarkan eksperimen yang dilakukan dapat diketahui bahwa *plot* yang dihasilkan mirip dengan *steepest ascent hill-climbing* yang hanya memiliki perubahan signifikan selama 20 iterasi pertama.

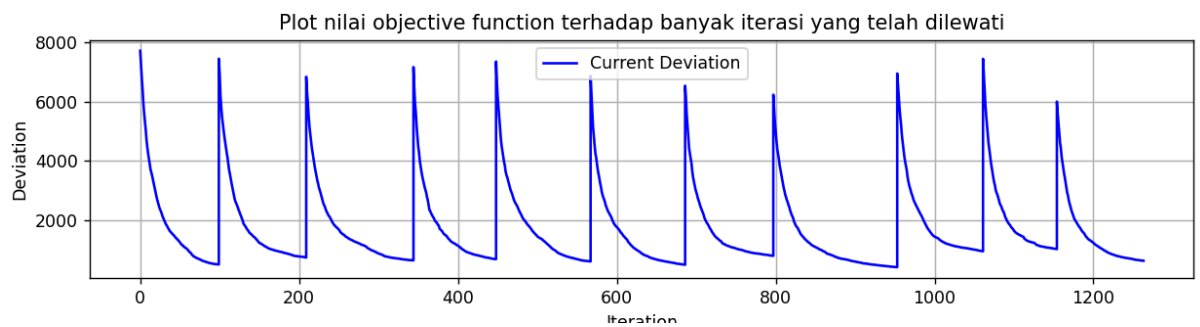
Terkait hubungan `max_iterations` dengan hasil yang didapat juga sama dimana semakin besar `max_iterations` akan menyebabkan semakin lama durasi dan juga semakin kecil nilai deviasi yang berarti semakin mendekati *magic cube* sempurna. Tetapi ada satu hal yang perlu dicatat yaitu karena ruang tetangga pencarian yang sangat besar yaitu 125! dan yang membedakan antara *steepest ascent* dan *sideways move* hanya diperbolehkannya untuk bergerak ke nilai yang sama sehingga hal ini membuat hasil yang didapat bisa dibilang hampir sama.

Apabila dibandingkan dengan algoritma *local search* lainnya tentu algoritma ini juga bukan yang paling optimal untuk menyelesaikan permasalahan *magic cube*. Hal ini karena hasil yang didapat juga belum maksimal, di mana walaupun telah ditetapkan iterasi maksimal sebesar 1000 tetap sulit untuk mencapai global optima. Algoritma ini juga cenderung masih terjebak dalam local optima yang membuatnya kurang optimal dalam penyelesaian masalah ini.

- **Random Restart Hill-Climbing**

Berikut ini adalah hasil *plot* dari *random restart hill-climbing* beserta hasil yang didapat dari *search* yang dilakukan. Eksperimen yang dilakukan adalah mengubah parameter berupa jumlah maksimal iterasi yang dapat dilakukan dan jumlah *restart* yang dilakukan.

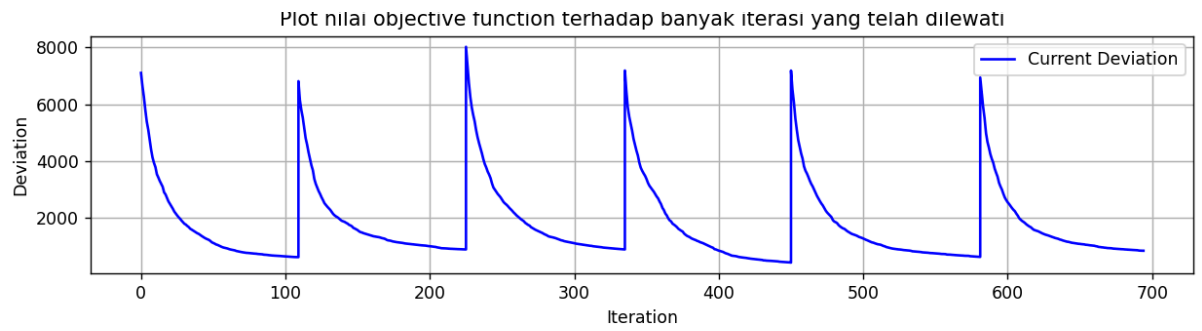
A. `max_iterations=1000, max_restart=10`



**Final Obj Func: 408**  
**Duration: 495.686 s**

<p>Initial State:</p> <pre>[[[ 30 104 109 49 105]   [ 76 121 111 116 92]   [ 31 2 38 107 66]   [ 3 123 70 108 83]   [112 58 114 96 16]]  [[ 26 102 87 51 90]  [124 80 115 8 46]  [ 52 33 59 55 4]  [ 68 14 91 98 101]  [ 47 62 48 63 10]]  [[ 17 74 110 77 36]  [ 22 9 65 6 120]  [ 23 34 45 119 18]  [ 21 97 75 85 29]  [ 7 122 82 44 40]]  [[ 99 89 56 32 103]  [ 20 125 78 106 61]  [ 12 37 71 94 1]  [100 50 79 93 95]  [ 54 19 13 42 88]]  [[113 57 60 118 117]  [ 67 28 35 24 53]  [ 84 15 69 81 11]  [ 73 5 39 41 72]  [ 25 64 27 86 43]]]</pre>	<p>Final State:</p> <pre>[[[ 41 20 124 85 45]   [ 39 125 18 57 65]   [ 50 73 87 76 29]   [110 77 2 6 120]   [ 75 15 84 91 59]]  [[ 28 114 12 94 66]  [113 62 40 74 31]  [ 9 98 88 108 11]  [103 23 72 24 90]  [ 60 19 105 13 117]]  [[121 26 36 16 115]  [104 27 49 17 116]  [ 37 109 56 58 55]  [ 8 81 93 107 25]  [ 46 71 83 112 4]]  [[ 30 54 122 78 32]  [ 10 79 89 101 38]  [123 33 52 5 102]  [ 70 35 44 86 80]  [ 82 111 7 47 68]]  [[ 97 96 14 42 61]  [ 48 21 119 63 64]  [ 95 1 34 67 118]  [ 22 100 106 92 3]  [ 53 99 43 51 69]]]</pre>
---	---

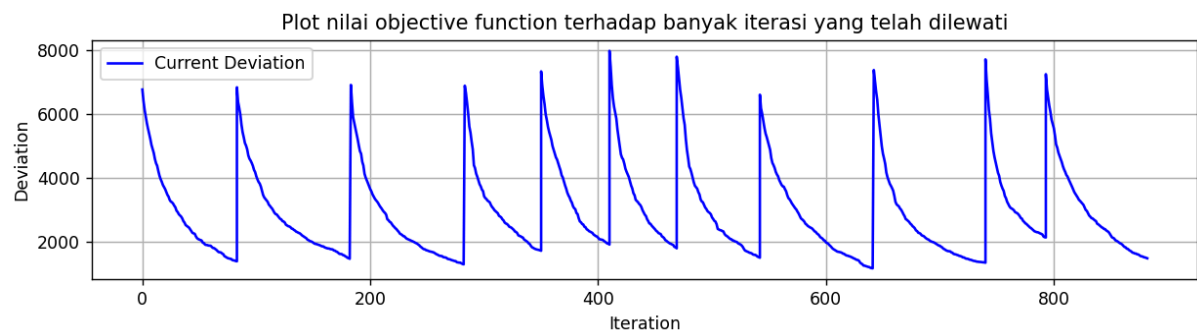
B. max\_iterations=1000, max\_restart=5



Final Obj Func: 437  
Duration: 222.764 s

Initial State:	Final State:
[[ 99 17 106 125 45]	[[ 33 115 55 52 61]
[ 81 40 39 55 119]	[106 7 41 45 120]
[ 36 111 117 124 75]	[109 6 90 108 2]
[ 59 79 67 24 21]	[ 51 98 65 80 28]
[ 46 3 101 63 107]]	[ 15 102 60 29 105]]
 [[113 102 62 50 73]	 [[124 25 36 47 83]
[103 115 54 87 11]	[ 5 57 99 101 54]
[ 92 93 70 35 104]	[ 20 93 12 50 125]
[ 23 19 64 9 42]	[ 94 59 49 91 22]
[ 61 16 112 57 27]]	[ 72 75 118 27 24]]
 [[ 8 72 110 66 33]	 [[ 76 14 68 77 82]
[ 96 56 43 89 14]	[ 73 121 53 26 46]
[ 2 37 29 71 80]	[ 63 35 88 70 58]
[ 1 48 28 51 74]	[ 32 48 74 30 123]
[ 12 22 10 91 31]]	[ 71 97 37 110 3]]
 [[105 95 77 85 123]	 [[ 56 117 42 100 1]
[ 25 44 97 4 116]	[122 18 4 87 84]
[120 78 69 86 7]	[ 10 78 111 9 107]
[ 32 100 122 114 83]	[ 95 79 66 34 40]
[ 94 98 68 15 41]]	[ 38 23 92 85 86]]
 [[ 53 82 65 6 13]	 [[ 21 44 114 39 89]
[ 26 58 52 18 60]	[ 16 112 116 62 11]
[ 47 109 76 49 30]	[113 104 13 69 17]
[121 20 84 90 88]	[ 43 31 64 81 96]
[ 34 5 38 118 108]]	[119 19 8 67 103]]]

C. max\_iterations=100, max\_restart=10

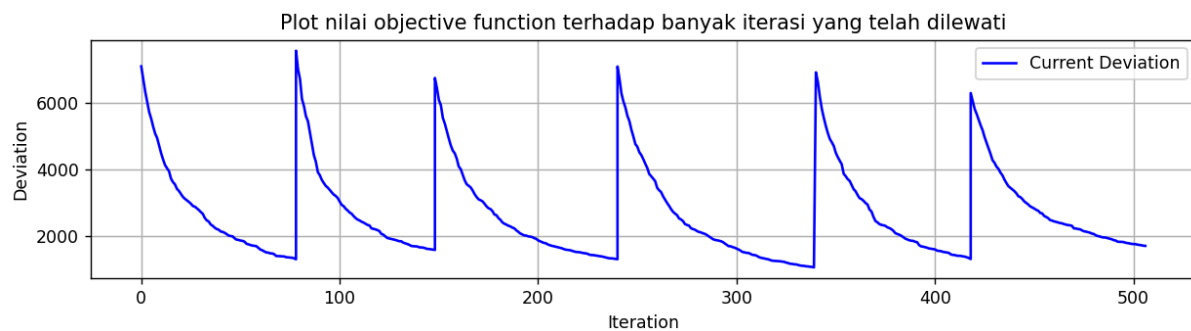




Final Obj Func: 1131  
Duration: 29.336 s

Initial State:	Final State:
[[[ 96 79 41 15 68]	[[[ 85 33 37 88 94]
[ 45 2 23 72 40]	[118 67 48 30 52]
[ 26 60 58 95 32]	[ 46 86 15 104 55]
[ 82 30 4 85 77]	[ 3 110 103 87 34]
[ 92 42 27 18 51]]	[ 63 16 120 24 77]]
 [[ 74 17 123 115 28]	 [[ 73 26 113 62 41]
[ 63 94 87 103 35]	[ 17 66 117 68 40]
[ 33 49 50 38 105]	[ 58 44 64 27 119]
[ 61 5 86 114 57]	[ 75 51 6 82 98]
[104 29 34 71 25]]	[ 91 121 8 76 29]]
 [[ 83 55 76 9 112]	 [[ 71 95 50 2 108]
[ 89 8 66 64 11]	[ 28 35 19 100 125]
[ 46 47 1 22 65]	[ 99 80 78 7 57]
[ 62 113 99 20 48]	[106 14 90 111 5]
[ 12 117 69 59 13]]	[ 12 114 79 96 21]]
 [[ 6 10 119 16 110]	 [[ 47 53 112 92 9]
[ 93 98 120 73 24]	[ 32 70 102 69 45]
[102 97 109 107 21]	[ 93 11 23 116 72]
[ 36 118 75 14 19]	[ 56 124 18 36 74]
[ 3 7 106 81 101]]	[ 89 38 65 1 122]]
 [[108 88 122 116 90]	 [[ 39 109 25 84 60]
[121 56 78 80 37]	[107 81 31 43 49]
[ 84 53 70 44 100]	[ 20 105 123 54 22]
[124 43 67 111 54]	[ 83 13 101 4 97]
[ 39 91 125 52 31]]]	[ 59 10 42 115 61]]]

D. max\_iterations=100, max\_restart=5



Final Obj Func: 1044  
Duration: 21.257 s

Initial State:	Final State:
[[[ 97 10 48 60 20]	[[[ 78 11 94 6 69]
[ 63 45 78 118 90]	[ 85 90 39 113 4]
[ 35 21 92 54 110]	[ 89 7 12 122 96]
[124 37 88 93 112]	[ 37 106 75 43 59]
[ 28 121 85 49 84]]	[ 15 95 104 28 92]]
[[ 83 57 123 122 81]	[[ 82 84 9 97 44]
[ 69 106 31 103 11]	[ 1 54 77 111 76]
[ 27 47 71 36 33]	[123 42 68 62 17]
[ 15 80 116 107 3]	[112 71 41 31 60]
[ 38 79 4 43 24]]	[ 22 67 119 21 83]]
[[ 39 98 16 9 67]	[[103 88 26 86 18]
[ 7 108 56 102 40]	[ 3 101 72 19 121]
[ 2 72 115 29 100]	[ 63 81 53 61 64]
[ 94 34 14 13 46]	[ 2 51 114 45 110]
[105 75 52 17 119]]	[124 8 52 109 13]]
[[125 87 117 42 99]	[[ 30 93 116 20 55]
[ 68 6 82 50 64]	[118 24 27 66 80]
[ 18 109 65 58 53]	[ 29 102 108 38 36]
[ 1 114 23 111 77]	[ 91 48 49 74 58]
[ 96 61 51 12 113]]	[ 57 46 14 117 79]]
[[ 89 70 104 44 86]	[[ 16 40 70 115 107]
[120 73 101 25 95]	[125 56 105 5 33]
[ 55 26 41 74 62]	[ 10 87 73 34 99]
[ 66 91 5 76 32]	[ 65 32 35 120 23]
[ 30 22 59 8 19]]]	[ 98 100 25 47 50]]]

Pembahasan : Berdasarkan 4 eksperimen yang telah dilakukan, dapat dilihat *plot* yang terbentuk semuanya adalah *plot* yang sama dengan algoritma *hill-climbing* sebelumnya, yang membedakan hanyalah *plot* yang terbentuk sesuai dengan jumlah *restart* yang dilakukan.

Jika dilihat dari segi hasil *search* yang dilakukan berdasarkan parameter *max\_restart*. Pada *max\_iterations* 1000 ketika menggunakan 10 *max\_restart* menghasilkan hasil yang lebih baik (nilai deviasi lebih kecil) sedangkan pada *max\_iterations* 100 terjadi sebaliknya. Hal ini membuktikan bahwa banyak *restart* tidak menjamin hasil yang didapat lebih baik dari yang memiliki jumlah *restart* yang lebih sedikit hanya saja akan meningkatkan kemungkinan untuk mendekati global optima.

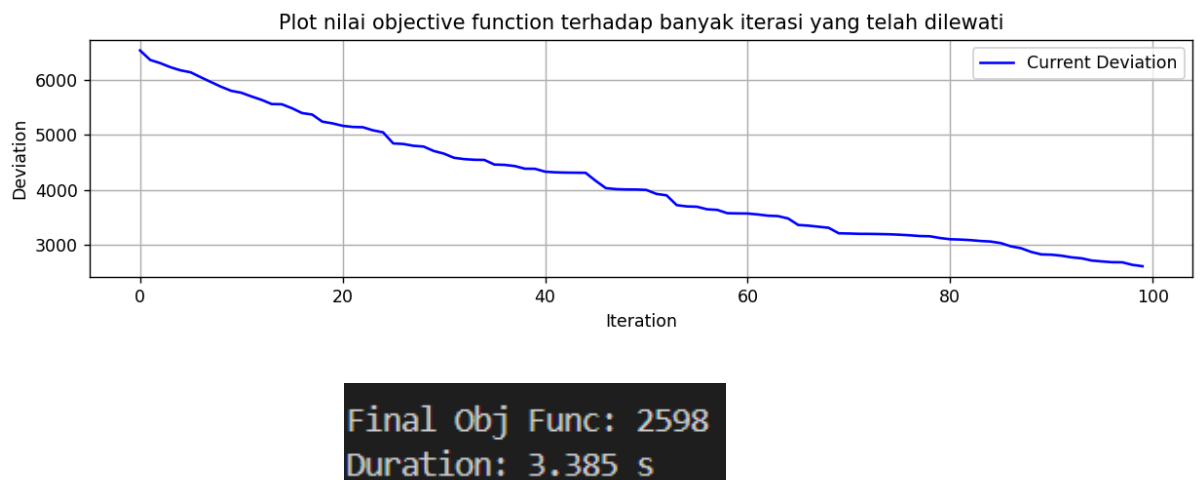
Kemudian dari segi durasi dapat dilihat bahwa semakin besar *max\_restart* dan *max\_iterations* maka akan semakin besar pula durasi yang dibutuhkan untuk melakukan *search*. Kita juga dapat menyimpulkan bahwa sama seperti dengan algoritma *hill-climbing* sebelumnya di mana semakin besar *max\_iterations* maka akan semakin kecil pula nilai deviasi yang berarti semakin mendekati *magic cube* sempurna.

Apabila dibandingkan dengan algoritma *local search* lainnya dapat dibilang bahwa *random restart hill-climbing* akan lebih unggul dibanding *hill-climbing* lainnya karena melakukan sejumlah *restart* yang memberikan kemungkinan untuk semakin mendekati global optima. Meskipun demikian, perlu dicatat bahwa semakin banyak jumlah *restart* yang dilakukan maka akan memakan waktu yang semakin besar sehingga menjadi kurang efisien dalam pencarian apalagi menggunakan *max\_iterations* yang cukup besar seperti 1000. Namun, Jika dibandingkan dengan algoritma lain selain *hill-climbing*, hasil yang didapat algoritma *random restart hill-climbing* kurang optimal di mana pada eksperimen dengan waktu yang hampir mencapai 500 sekon hanya mencapai nilai deviasi sebesar 408. Jadi, dapat disimpulkan jika penggunaan algoritma *random restart hill-climbing* akan memberikan kemungkinan yang lebih besar untuk mencapai global optima tetapi kurang efisien.

- **Stochastic Hill-Climbing**

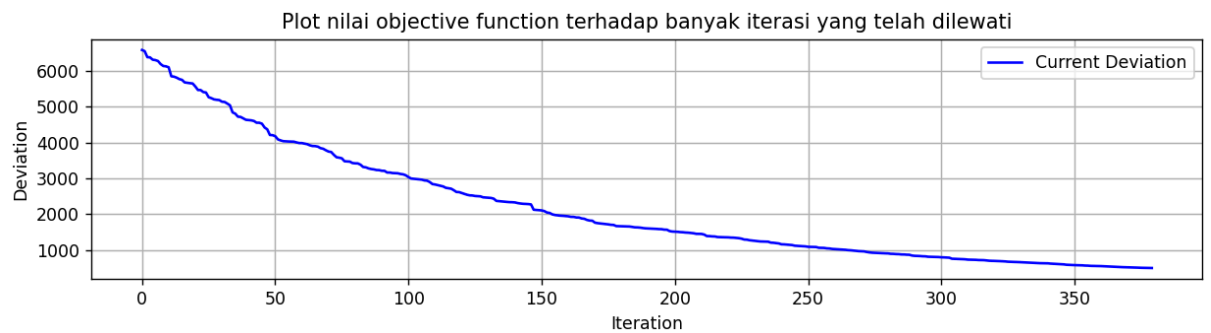
Berikut ini adalah hasil *plot* dari *stochastic hill-climbing* beserta hasil yang didapat dari *search* yang dilakukan. Eksperimen yang dilakukan adalah mengubah parameter berupa jumlah maksimal iterasi yang dapat dilakukan.

A. *max\_iterations* = 100



Initial State:	Final State:
[[[ 19 33 45 31 95]	[[[ 66 88 74 56 61]
[ 117 4 23 28 70]	[ 112 82 27 28 59]
[ 24 121 74 106 102]	[ 24 124 17 12 123]
[ 98 39 79 47 17]	[ 98 21 79 81 4]
[ 25 3 81 104 62]]	[ 37 3 111 104 62]]
[[ 18 30 71 50 69]	[[110 30 13 86 46]
[ 59 20 118 57 34]	[103 10 118 63 19]
[ 2 103 61 12 6]	[ 2 106 47 41 108]
[ 42 65 27 56 80]	[ 7 107 58 57 109]
[122 73 84 77 41]]	[ 89 48 100 77 70]]
[[ 53 101 32 8 94]	[[ 16 101 119 8 94]
[120 64 10 124 119]	[ 78 76 20 9 72]
[ 14 55 97 89 26]	[105 40 85 96 26]
[ 66 11 96 63 86]	[ 90 11 54 75 91]
[ 82 109 85 67 40]]	[ 51 102 22 67 55]]
[[108 51 116 115 110]	[[ 95 15 71 115 65]
[ 13 68 43 49 38]	[ 33 121 43 52 38]
[ 92 1 48 22 78]	[117 1 50 60 97]
[ 21 87 99 72 76]	[ 5 87 99 39 80]
[123 91 93 36 16]]	[ 64 84 93 42 18]]
[[ 83 46 37 75 7]	[[ 45 69 36 53 34]
[100 29 107 105 15]	[ 49 29 92 44 116]
[ 52 35 125 111 90]	[ 68 35 125 120 6]
[ 54 5 112 60 9]	[113 122 31 23 32]
[113 58 44 88 114]]]	[ 73 83 14 25 114]]]

B. max\_iterations = 1000



Final Obj Func: 493  
Duration: 162.609 s

Initial State:	Final State:
[[[ 97 17 76 116 64]	[[[ 79 22 23 90 100]
[ 94 86 74 91 108]	[105 66 47 8 88]
[101 82 60 65 66]	[ 42 108 51 86 30]
[ 50 58 49 33 53]	[ 46 106 64 59 37]
[ 84 42 122 89 77]]	[ 48 12 123 77 60]]
 [[ 47 40 62 85 81]	 [[ 43 107 83 26 56]
[ 44 2 79 35 88]	[124 24 70 65 32]
[ 23 61 113 25 120]	[ 28 68 115 89 15]
[123 83 29 7 121]	[111 61 27 20 98]
[ 34 78 59 92 95]]	[ 9 55 18 118 112]]
 [[109 67 1 103 48]	 [[125 40 10 50 91]
[ 31 68 30 115 19]	[ 13 39 74 113 75]
[119 80 104 39 43]	[ 35 119 69 58 34]
[ 22 9 73 112 72]	[ 38 3 121 57 95]
[ 27 107 21 46 8]]	[104 114 41 31 25]]
 [[ 10 124 111 98 114]	 [[ 14 117 103 67 16]
[ 26 5 14 54 71]	[ 72 102 2 71 76]
[106 24 96 18 20]	[ 99 17 73 4 120]
[ 55 38 100 11 41]	[ 21 45 93 92 63]
[102 70 16 45 4]]	[109 36 49 81 33]]
 [[ 37 52 105 57 36]	 [[ 54 29 96 82 52]
[ 3 75 28 69 32]	[ 1 85 122 62 44]
[ 87 13 12 125 15]	[110 5 6 78 116]
[ 6 117 90 56 63]	[ 97 101 11 87 19]
[ 93 118 110 51 99]]]	[ 53 94 80 7 84]]]

Pembahasan : Berdasarkan eksperimen yang telah dilakukan dapat dilihat bentuk *plot* yang dihasilkan memang serupa dengan *hill-climbing* lainnya tetapi perlu dicatat bahwa *plot* yang dihasilkan memiliki perbedaan dalam jumlah iterasi, jumlah iterasi yang dibutuhkan algoritma ini jauh lebih besar sehingga tidak ada perubahan deviasi yang cukup signifikan seperti algoritma *hill-climbing* lainnya.

Jika kita meninjau dari hasil yang didapat, hasil akan bagus apabila kita menggunakan `max_iterations` dengan nilai yang lebih tinggi karena seperti yang kita ketahui algoritma *stochastic hill-climbing* akan mencari secara *random* nilai yang lebih baik jadi tidak langsung ke nilai yang terbaik jadi akan memakan iterasi yang cukup lama hingga tidak ada nilai yang lebih baik lagi. Dapat dilihat ketika kita melakukan eksperimen untuk yg `max_iterations` bernilai 100, algoritmanya berhenti karena telah mencapai iterasi maksimal.

Jika meninjaunya dari segi waktu, waktu yang dibutuhkan untuk algoritma ini akan lebih besar dari algoritma *steepest ascent* yang disebabkan oleh pemilihan secara acak yang menyebabkan iterasi yang dibutuhkan semakin banyak. Apabila semakin banyak iterasi maka akan memakan waktu yang semakin banyak pula. Selain itu,

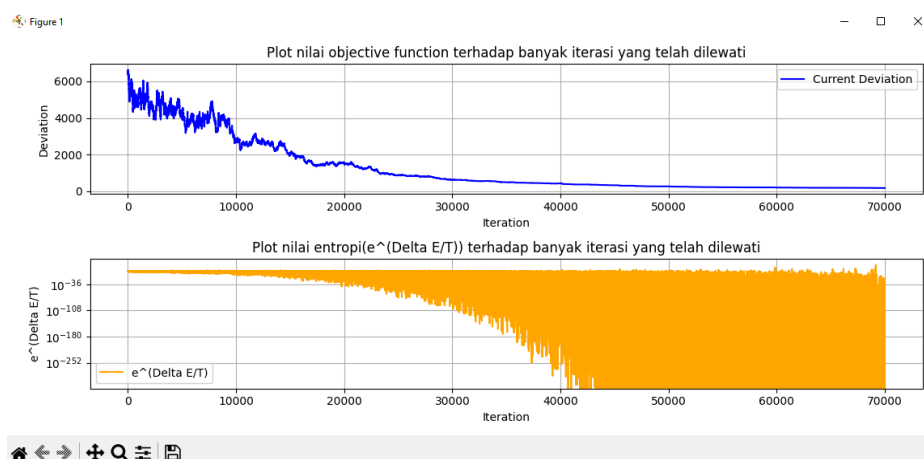
`max_iterations` yang ditentukan juga akan berpengaruh terhadap lama waktu pencarian.

Apabila kita bandingkan dengan algoritma *local search* lainnya, penggunaan algoritma *stochastic hill-climbing* mendapatkan hasil bergantung pada pemilihan tetangga yang lebih baik secara acak. Hal ini membuat algoritma ini bukan yang paling optimal dalam mencari objektif berupa *magic cube* yang sempurna. Walaupun jika dibandingkan dengan eksperimen *steepest ascent* dan *sideways move* mendapatkan hasil yang lebih baik meskipun waktu yang lebih lama, tetap saja algoritma ini tidak selamanya dapat menghasilkan nilai yang lebih baik karena bergantung pada *random* yang dilakukan. Jika dibandingkan dengan algoritma selain *steepest ascent hill-climbing* dan *hill-climbing with sideways move* dapat dilihat bahwa hasil yang dihasilkan *stochastic hill-climbing* tetap kurang maksimal yaitu masih di angka 493 yang masih cukup jauh dalam mencapai *magic cube* sempurna.

## ● Simulated Annealing

Terlampir adalah hasil plottingan dari simulated annealing, akan dilakukan beberapa eksperimen berbeda dengan cara mengganti initial temperature dan cooling rate untuk memvisualisasikan perbedaan yang dihasilkan ketika 2 variabel ini dirubah.

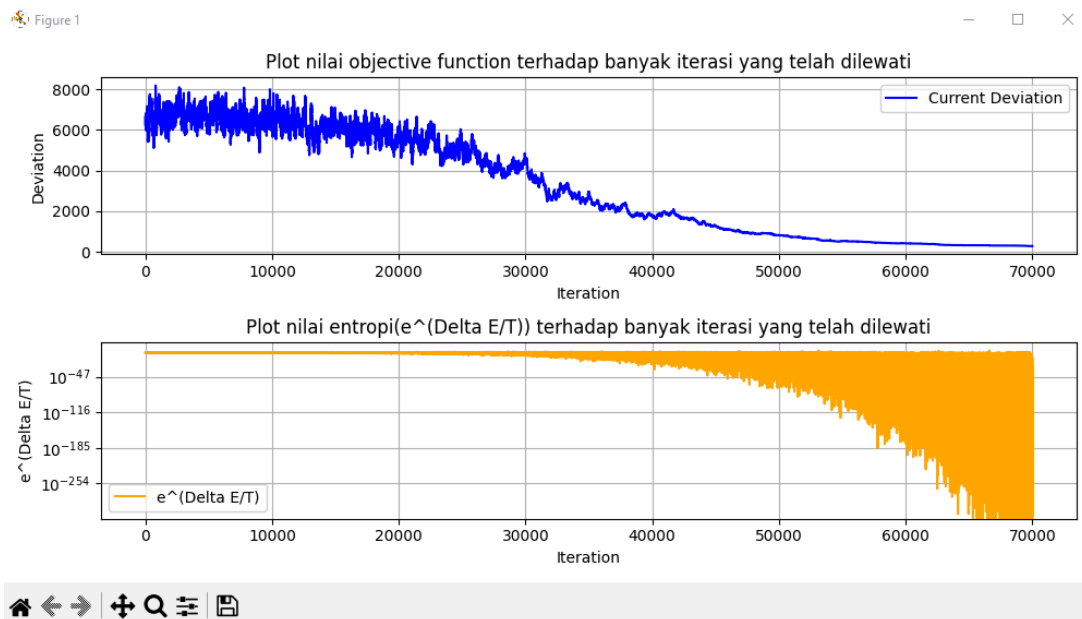
A. Initial Temperature = 100 , Cooling rate = 0.9999



```
Final Obj Func: 173
Duration: 38.947 s
'stuck' in local optimum: 4256
```

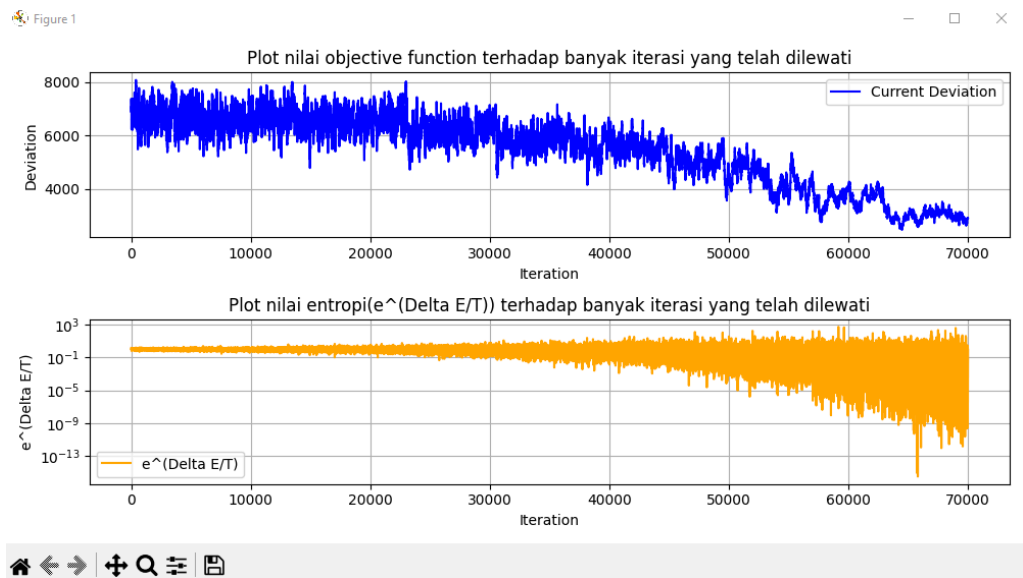
Initial State:	Final State:
[ 27 48 103 80 44]	[ 89 58 67 25 77]
[ 50 53 17 36 104]	[ 98 30 73 13 101]
[ 62 125 84 90 78]	[ 47 115 69 78 3]
[ 41 107 85 83 79]	[ 1 88 28 96 102]
[ 65 112 86 40 21]	[ 80 20 79 104 31]
[ 105 58 99 42 110]	[ 62 85 51 74 43]
[ 113 10 16 109 35]	[ 72 55 86 82 22]
[ 115 89 4 15 108]	[ 36 57 48 84 93]
[ 88 119 34 87 9]	[ 100 95 50 32 38]
[ 55 76 61 47 91]	[ 44 26 81 46 118]
[ 2 56 38 8 45]	[ 41 65 4 91 114]
[ 33 81 46 116 121]	[ 122 121 29 10 33]
[ 26 120 123 1 12]	[ 14 71 56 75 99]
[ 100 117 77 37 54]	[ 27 23 112 90 63]
[ 20 124 52 60 68]	[ 111 35 113 49 7]
[ 101 49 32 64 57]	[ 18 53 68 117 59]
[ 39 67 28 122 118]	[ 6 87 9 94 119]
[ 13 51 31 3 95]	[ 105 61 92 45 12]
[ 111 43 24 22 25]	[ 123 5 106 37 42]
[ 23 114 29 7 63]	[ 64 109 40 19 83]
[ 75 19 102 14 73]	[ 107 54 125 8 21]
[ 5 6 93 72 98]	[ 17 24 120 116 39]
[ 69 71 18 82 106]	[ 110 11 52 34 108]
[ 94 96 66 70 74]	[ 66 103 15 60 70]
[ 30 97 11 92 59]	[ 16 124 2 97 76]

B. Initial Temperature = 1000 , Cooling rate = 0.9999



Initial State:	Final State:
[[ 72 83 116 43 101]	[[ 30 36 9 122 118]
[ 74 52 36 25 17]	[ 55 123 101 2 33]
[ 53 111 24 104 90]	[112 7 47 113 38]
[ 37 27 113 31 77]	[ 50 88 74 44 58]
[ 71 1 86 33 20]]	[ 63 61 85 37 68]]
[[ 85 109 92 49 120]	[[ 77 124 86 16 11]
[ 78 26 48 9 30]	[ 89 71 41 42 69]
[ 47 29 3 68 123]	[ 35 4 75 87 115]
[ 45 118 87 114 57]	[ 8 81 80 57 90]
[121 98 32 46 18]]	[107 32 27 114 34]]
[[ 60 39 99 97 55]	[[ 25 119 91 67 12]
[124 16 8 96 122]	[ 29 13 96 72 105]
[106 94 11 80 21]	[103 109 56 45 6]
[ 42 65 117 61 13]	[ 31 51 53 104 76]
[ 81 56 119 62 63]]	[125 23 22 28 117]]
[[ 23 115 5 82 67]	[[111 20 46 18 121]
[107 102 108 54 44]	[ 54 62 14 84 100]
[ 38 4 70 10 125]	[ 24 98 78 49 64]
[ 34 58 50 2 105]	[106 17 102 66 26]
[ 35 88 103 110 6]]	[ 19 116 73 99 3]]
[[ 66 95 28 15 22]	[[ 70 15 83 94 52]
[ 93 51 76 73 112]	[ 92 43 59 110 10]
[ 59 7 84 64 14]	[ 40 97 60 21 95]
[ 91 40 19 79 75]	[120 79 5 48 65]
[ 41 100 89 69 12]]]	[ 1 82 108 39 93]]]

C. Initial Temperature = 1000 , Cooling rate = 0.99995

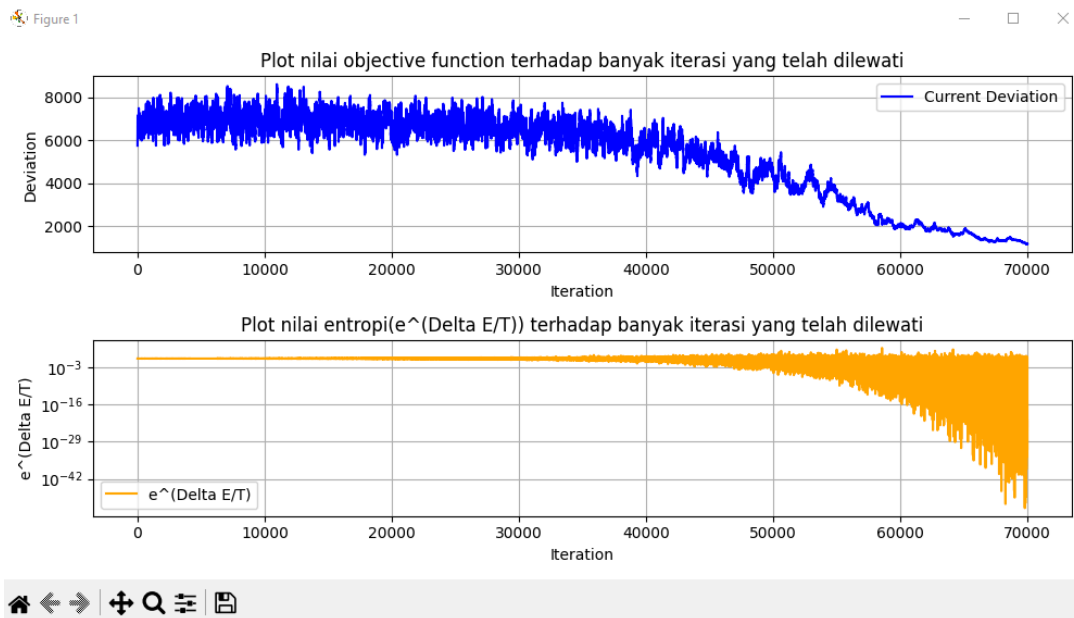


```
Final Obj Func: 2476
Duration: 39.317 s
'stuck' in local optimum: 26237
```



Initial State:	Final State:
[[[ 49 38 118 44 98]	[[[ 62 85 30 77 82]
[123 5 32 18 78]	[ 71 97 35 15 65]
[122 15 3 71 89]	[ 22 9 63 64 113]
[ 83 86 2 25 85]	[107 26 123 49 16]
[ 14 76 117 57 84]]	[ 73 67 55 86 18]]
[[115 101 26 105 99]	[[ 60 23 125 45 42]
[ 47 8 39 79 111]	[ 12 48 72 57 53]
[ 80 20 112 97 95]	[100 87 46 92 47]
[119 55 104 68 125]	[ 21 80 54 104 94]
[ 67 23 24 109 50]]	[105 81 5 40 118]]
[[ 56 124 31 27 65]	[[ 13 102 59 61 78]
[ 74 45 106 63 75]	[ 44 95 75 91 39]
[ 22 82 90 53 121]	[119 90 79 88 17]
[ 36 33 30 11 87]	[120 11 32 68 106]
[107 120 37 9 66]]	[ 36 33 69 4 124]]
[[ 48 73 72 100 34]	[[ 70 25 41 111 83]
[ 13 114 28 93 61]	[ 99 37 27 19 112]
[ 42 29 58 6 102]	[101 31 114 14 28]
[116 70 64 1 91]	[ 24 93 29 56 115]
[ 88 43 62 96 113]]	[ 10 121 96 89 1]]
[[ 54 81 92 4 17]	[[110 38 66 20 74]
[ 77 7 19 52 60]	[ 43 34 117 84 76]
[108 40 41 16 51]	[ 7 103 6 50 116]
[ 69 21 59 10 12]	[ 58 122 8 109 2]
[110 94 35 103 46]]]	[ 98 3 108 51 52]]]

D. Initial Temperature = 1000 , Cooling rate = 0.9999



```
Final Obj Func: 1158
Duration: 38.822 s
'stuck' in local optimum: 25858
```

Initial State:	Final State:
[[ 48 107 100 112 50]	[[ 37 101 65 24 88]
[ 62 78 97 25 91]	[ 64 11 106 27 97]
[ 69 8 31 20 106]	[ 43 118 46 103 5]
[ 14 117 85 11 15]	[110 61 16 115 30]
[ 79 52 45 89 10]]	[ 58 25 67 51 99]]
[[ 66 90 83 114 19]	[[ 47 77 23 94 74]
[ 46 37 4 95 71]	[ 95 86 2 120 22]
[ 67 72 24 42 115]	[100 1 96 28 108]
[ 27 73 99 98 110]	[ 98 10 113 36 52]
[113 35 41 32 84]]	[ 14 114 107 32 39]]
[[119 23 51 65 29]	[[124 19 91 93 3]
[ 92 6 61 17 63]	[ 40 66 12 69 125]
[ 1 77 75 118 33]	[ 31 84 81 60 53]
[109 101 16 93 88]	[ 13 56 123 20 112]
[116 124 40 56 28]]	[111 68 7 80 26]]
[[ 76 22 57 58 105]	[[ 62 8 119 38 78]
[104 108 68 64 87]	[104 76 70 92 6]
[ 34 102 5 55 18]	[ 21 87 45 82 59]
[ 12 26 2 3 38]	[ 75 57 49 44 89]
[ 21 120 9 125 86]]	[ 48 109 17 63 90]]
[[ 13 60 111 39 103]	[[ 34 105 15 83 85]
[ 36 123 82 47 70]	[ 35 79 121 9 55]
[122 43 80 81 44]	[122 33 41 50 73]
[ 59 53 94 74 49]	[ 42 117 18 102 29]
[ 54 7 121 30 96]]]	[ 71 4 116 54 72]]]

Pembahasan : Pertama-tama terkait plotting untuk nilai entropi terhadap iterasi diputuskan untuk menggunakan scale logarithmic, dengan alasan ketika menggunakan plotting biasa atau linear, ada spike-spike besar yang diakibatkan oleh nilai  $e^{(\Delta E/T)}$  dimana  $\Delta E$  nya begitu besar, sehingga plotting-nya menjadi jelek. Hal ini diakibatkan ketika menggunakan skala linear, dari 1 ke 2 sama saja nilai/jaraknya dari 100 ke 101, sedangkan ketika menggunakan skala logaritmik, unit yang dipakai berupa multiplikasi misal ada nilai 1,10,100,1000 maka akan dipresentasikan dengan angka 0,1,2,3, dengan menggunakan ini maka jika ada spike di perhitungan, tidak akan terlalu mendominasi nilai lain di plottingan.

Selanjutnya, terkait jalannya algoritma itu sendiri, seperti yang sebelumnya dibahas di bagian penjelasan implementasi algoritma local search, bagaimana berjalannya simulated annealing sangat bergantung terhadap suhu dan cooling rate-nya. Saya memvariasikan suhu-nya dari 100, 1000, dan 10000. Bisa dilihat bahwa sesuai urutan tersebut, semakin rendah dimana suhu dimulai, maka semakin cepat algoritma tersebut bergerak ke titik global optimum. Sedangkan ketika dimulai di suhu tinggi, tingkat randomness jauh lebih tinggi karena ketika dijalankan fungsi  $e^{(\Delta E/T)}$  ketika  $neighbor\_state$  lebih buruk dari  $current\_state$ , suhu berpengaruh besar ke nilai entropi tersebut, dimana nilai entropi tersebut akan dibandingkan

dengan angka yang digenerate secara random antara 0-1, jika angka entropi lebih besar dari angka random tersebut maka algoritma akan tetap berpindah ke `neighbor_state` walaupun nilai obj functionnya lebih buruk.

Terkait dekatnya ke global optima, Simulated Annealing tergolong sangat dekat terhadap global optima dengan nilai terbaik yang berkisar antara 100-200, hal ini bisa didapatkan karena kecenderungan stuck di local optima sampai algoritma berakhir sangat kecil (dengan catatan, initial temperature yang digunakan cukup tinggi dan cooling rate tidak menurun secara drastis).

Apabila dibandingkan dengan algoritma local search yang lain, dibandingkan steepest ascent pasti sudah lebih baik SA, dibandingkan dengan stochastic juga lebih baik SA karena SA bisa dibilang adalah versi superior dari stochastic yang mencoba secara random tetapi ditambahkan aspek temperatur yang mencoba-coba state yang lebih buruk sehingga aspek randomness tetap ada yang membuat peluang algoritma untuk stuck di local optima jauh lebih kecil. Selanjutnya dibandingkan dengan sideways move,

Terkait cooling rate, saya mencoba membandingkan antara algoritma dengan suhu yang sama tetapi cooling rate-nya berbeda, disini saya bereksperimen dengan suhu 1000 yang cooling rate-nya divariasikan antara 0.9999 dan 0.99995. Terlihat bagaimana ketika cooling rate-nya dinaikkan maka efeknya terhadap berjalannya algoritma mirip seperti ketika algoritma berada di suhu yang sama di jangka waktu yang agak lama sehingga frekuensi stuck di local optima jauh lebih banyak dan algoritma cenderung bekerja seperti random walk algorithm.

Terkait durasi, sebenarnya tidak bisa dibandingkan dengan algoritma lain *entirely*, karena algoritma yang dijalankan berdasarkan banyak iterasi yang dilakukan (sebenarnya bisa juga stop loop-nya ketika  $T$  nya 0, tetapi ini juga sangat dipengaruhi variabel initial temperature dan cooling rate). Untuk percobaan dengan 70.000 iterasi didapatkan waktu yang hampir sama untuk tiap eksperimen yaitu sekitar 39 detik.

Terkait, konsistensi, dengan cooling rate (0.9999) dan initial temperature (100) yang sama hasil akhir dari objective function yang didapatkan sangat berdekatan

nilainya, setelah dijalankan 3 kali, kami menemukan selisih terbesar dari deviasi adalah 20.

### ○ Genetic Algorithm

Terlampir adalah hasil dari eksperimen dari Genetic Algorithm dengan **Iterasi** sebagai kontrol, dengan iterasi 1000. Dengan variasi populasi 10, 50 dan 100. Berikut hasilnya :

#### 1. Populasi 10

##### - Initial state cube

```
INITIAL STATE:
-----

Layer 1:
[[ 10  91  68  71   1]
 [105  94  77  19  36]
 [ 53  33  42  57 107]
 [ 52 119 124  21  31]
 [ 70  73  80  92  11]]

Layer 2:
[[112   6  58  63   5]
 [ 13 125  82  79  76]
 [ 78 114 104  23  54]
 [115  56 100 116  95]
 [ 25 106 113  35  26]]

Layer 3:
[[  8  93  66  97  16]
 [ 34  38  51  12  69]
 [102  86 111  29 117]
 [ 15  49  89  22   7]
 [ 55  81  59  61  44]]

Layer 4:
[[101  48  60  20  17]
 [  9  64  47  27 103]
 [ 32  39  41  83  43]
 [ 62  45  50  90  88]
 [ 84  28   3  18 122]]

Layer 5:
[[ 72  46 108  74  37]
 [ 14 118 110 120  98]
 [ 96  40  24   2  85]
 [ 67  65 109 121  75]
 [ 99  87   4  30 123]]

Initial Objective Function Value: 7469
```

##### - Final state cube

```

FINAL STATE:
-----

Layer 1:
[[ 55  27 117  23  76]
 [ 58  87  87  54  42]
 [ 89  40  17  79  82]
 [ 19 120  4 104  67]
 [ 50  42  89  55  46]]

Layer 2:
[[ 42  55 104  83  20]
 [ 70  82  10  70  79]
 [ 32  82 104  10  93]
 [ 67  55  96  67  19]
 [ 82  40  10  82 104]]

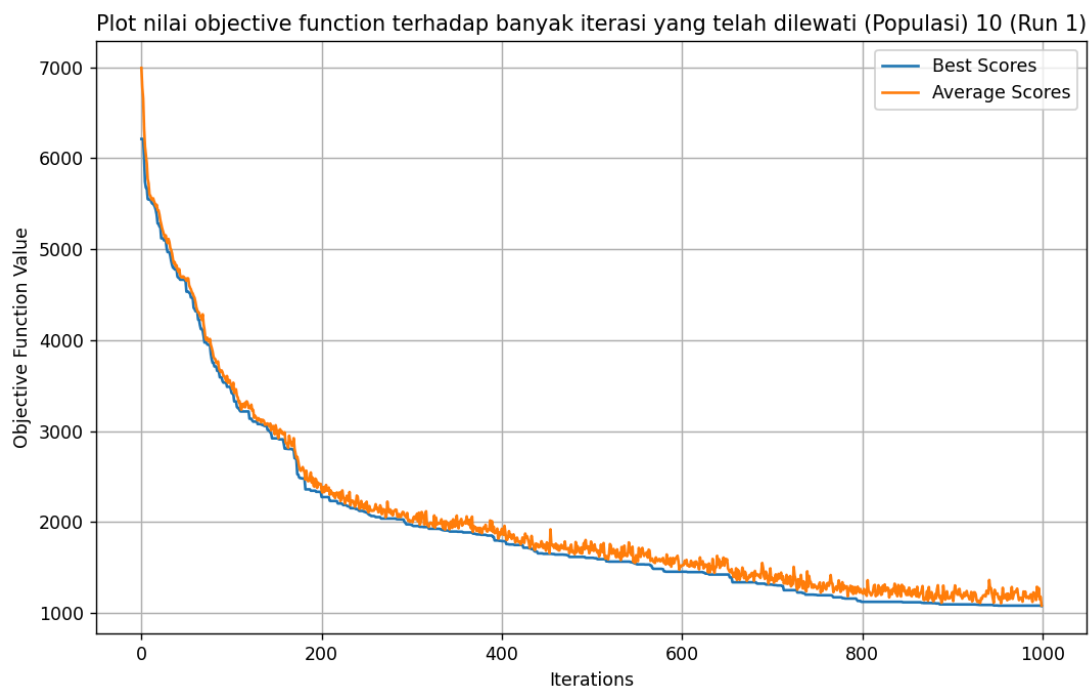
Layer 3:
[[ 84  51  40 112  32]
 [ 20  54  55  82 104]
 [ 20  95  65  73  81]
 [ 79  3  84  46  84]
 [112 120  67  3  12]]

Layer 4:
[[ 29  65  55  64  82]
 [ 76  32  64  78  64]
 [ 84  87  89  42  25]
 [120  62  40  67  35]
 [ 9  64  67  62 110]]

Layer 5:
[[104  96  3  32 104]
 [ 55  87  98  34  42]
 [ 82  11  42 116  42]
 [ 32  73  83  40  87]
 [ 55  49  83  84  42]]

```

- Grafik objective function value terhadap iterasi



- Waktu dan final objective function

```

Final Objective Function Value: 1076
Duration: 4.65 seconds

```

## 2. Populasi 50

- Initial state cube

```

INITIAL STATE:
-----

Layer 1:
[[ 90 40 58 87 2]
 [114 62 88 55 25]
 [ 24 48 59 26 50]
 [122 61 83 8 4]
 [ 28 16 42 31 41]]

Layer 2:
[[ 17 13 12 10 96]
 [ 15 94 49 124 38]
 [ 98 91 77 73 118]
 [ 86 29 34 9 116]
 [ 56 89 1 52 74]]

Layer 3:
[[ 23 82 80 78 53]
 [ 76 123 14 115 69]
 [ 93 92 43 11 79]
 [ 47 35 7 117 70]
 [ 57 104 54 51 125]]

Layer 4:
[[120 95 66 99 37]
 [ 33 106 20 81 85]
 [ 21 46 109 32 102]
 [ 30 97 67 6 39]
 [ 22 121 108 72 64]]

Layer 5:
[[110 45 111 19 103]
 [ 65 100 105 68 71]
 [ 18 113 36 101 63]
 [ 27 75 44 60 5]
 [112 3 107 119 84]]

Initial Objective Function Value: 6623

```

- Final state cube

```

FINAL STATE:
-----

Layer 1:
[[ 56 32 77 77 70]
 [ 57 75 70 78 36]
 [ 52 118 34 6 118]
 [118 23 28 95 52]
 [ 32 68 102 59 52]]

Layer 2:
[[ 23 118 95 57 23]
 [ 47 57 48 75 91]
 [124 12 98 25 40]
 [ 53 53 53 94 69]
 [ 63 73 21 63 94]]

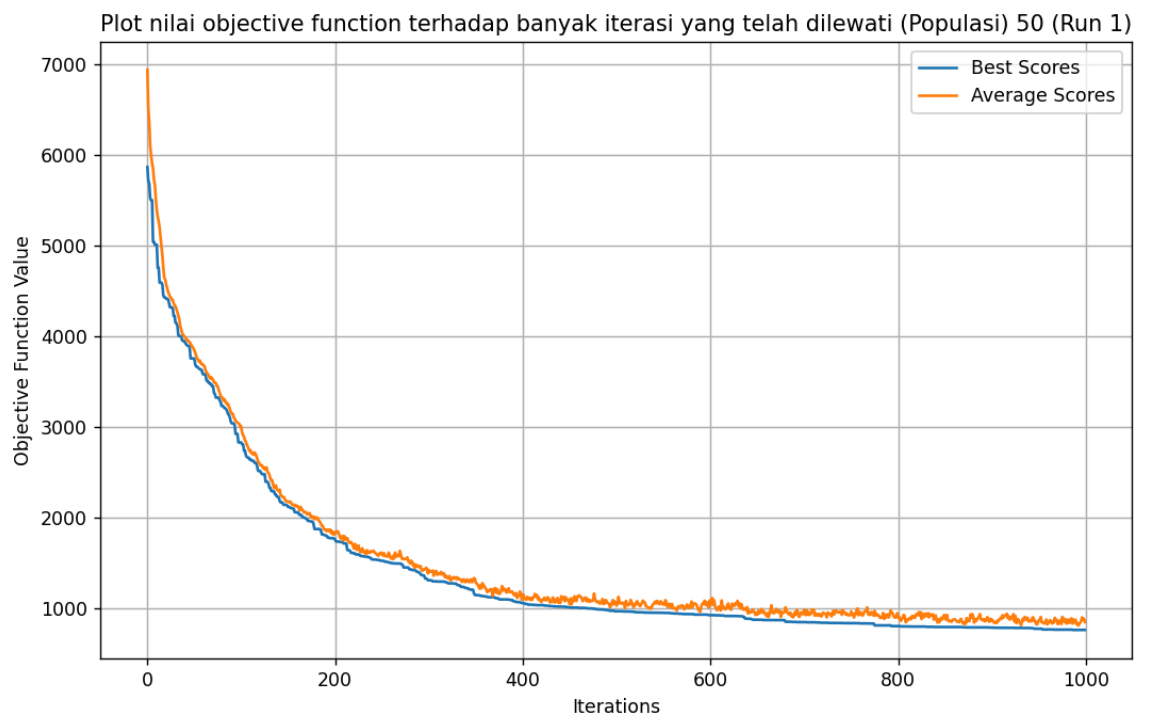
Layer 3:
[[ 69 25 12 81 124]
 [ 76 52 78 50 59]
 [ 74 95 75 68 3]
 [ 75 39 70 59 69]
 [ 28 98 74 57 60]]

Layer 4:
[[ 72 94 69 59 21]
 [ 75 59 6 60 114]
 [ 24 73 74 91 53]
 [ 46 76 70 48 75]
 [ 98 14 95 62 50]]

Layer 5:
[[ 95 43 63 35 75]
 [ 60 73 112 52 20]
 [ 40 20 35 124 97]
 [ 23 124 94 32 48]
 [ 94 54 23 70 74]]

```

- Grafik objective function value terhadap iterasi



- Waktu dan final objective function

```
Final Objective Function Value: 766  
Duration: 22.55 seconds
```

### 3. Populasi 100

- Initial state cube

```

INITIAL STATE:
-----

Layer 1:
[[ 72 83 17 42 88]
 [ 41 110 11 59 94]
 [119 1 33 63 114]
 [ 40 12 13 64 2]
 [ 98 5 106 103 8]]

Layer 2:
[[ 75 58 77 105 101]
 [ 4 57 69 86 92]
 [ 48 124 109 70 100]
 [ 47 37 95 55 85]
 [ 35 90 108 120 16]]

Layer 3:
[[ 93 23 102 29 73]
 [ 25 46 26 14 117]
 [ 54 53 34 84 62]
 [ 68 27 71 113 7]
 [115 38 6 118 50]]

Layer 4:
[[ 21 66 76 36 74]
 [125 78 61 51 89]
 [ 56 24 31 80 81]
 [ 45 79 30 32 91]
 [ 20 49 99 15 97]]

Layer 5:
[[ 39 107 96 3 116]
 [ 60 122 112 52 65]
 [ 43 121 104 67 123]
 [ 82 87 44 22 10]
 [ 9 28 18 111 19]]

Initial Objective Function Value: 6994

```

- Final state cube

```

FINAL STATE:
-----

Layer 1:
[[ 61 118 52 67 16]
 [ 51 48 69 44 101]
 [ 16 51 113 65 77]
 [ 69 31 8 86 118]
 [118 64 77 53 6]]

Layer 2:
[[ 26 52 107 82 44]
 [107 68 34 69 33]
 [124 69 68 10 41]
 [ 16 93 72 53 93]
 [ 41 33 34 101 106]]

Layer 3:
[[ 68 8 67 88 88]
 [ 28 107 56 2 123]
 [ 32 43 61 82 94]
 [114 85 53 59 5]
 [ 73 72 64 85 20]]

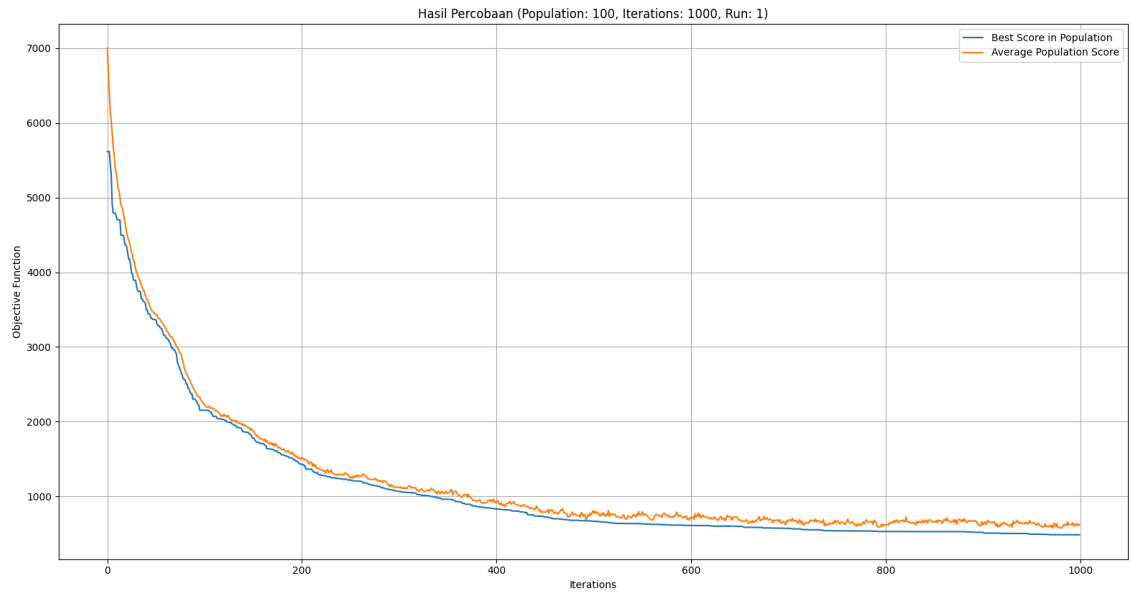
Layer 4:
[[ 73 64 34 59 82]
 [ 93 51 40 107 28]
 [ 59 62 54 106 34]
 [ 59 40 113 34 69]
 [ 31 98 77 8 101]]

Layer 5:
[[ 86 73 51 20 85]
 [ 34 41 116 93 33]
 [ 88 88 19 52 68]
 [ 56 67 69 82 41]
 [ 51 44 62 67 93]]

```

- Grafik objective function value terhadap iterasi





- Waktu dan final objective function

```
Final Obj Func: 483
Duration: 141.834 s
```

Sekarang hasil dari eksperimen dari Genetic Algorithm dengan **Populasi** sebagai kontrol, dengan populasi 50. Dengan variasi iterasi 100, 500 dan 1000. Berikut hasilnya :

#### 1. Iterasi 100

- Initial state cube

```

INITIAL STATE:
-----

Layer 1:
[[ 19 106 111 91 68]
 [119 66 44 69 82]
 [ 73 94 104 109 80]
 [105 16 32 61 10]
 [124 110 56 97 28]]

Layer 2:
[[112 12 64 81 42]
 [ 55 51 115 75 9]
 [ 65 120 60 6 46]
 [ 74 13 20 3 84]
 [ 18 90 93 17 40]]

Layer 3:
[[ 52 38 41 26 15]
 [ 1 5 33 107 88]
 [122 113 101 123 59]
 [ 21 85 98 48 45]
 [ 30 125 99 50 8]]

Layer 4:
[[ 11 49 43 108 87]
 [ 79 103 78 34 4]
 [ 7 83 121 63 72]
 [ 62 35 89 71 117]
 [ 54 58 102 95 70]]

Layer 5:
[[118 114 57 96 29]
 [ 2 25 14 31 92]
 [116 53 67 86 39]
 [ 22 27 77 37 47]
 [ 23 76 100 36 24]]

Initial Objective Function Value: 7315

```

- Final state cube

```

FINAL STATE:
-----

Layer 1:
[[ 33 48 124 17 84]
 [ 3 119 66 115 29]
 [ 51 44 48 82 64]
 [120 42 62 20 70]
 [115 67 6 28 89]]

Layer 2:
[[ 76 92 49 38 42]
 [ 61 73 32 96 5]
 [115 2 85 74 59]
 [ 37 83 50 52 109]
 [ 17 62 115 55 78]]

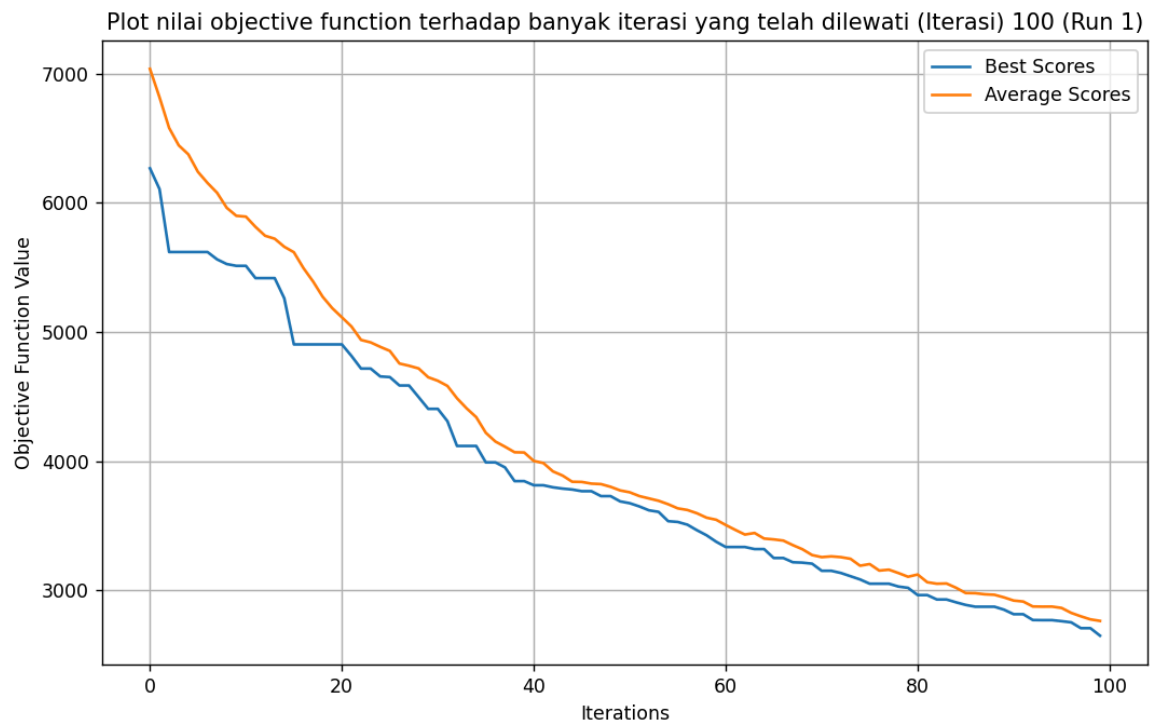
Layer 3:
[[ 53 103 7 90 71]
 [ 84 53 93 41 54]
 [ 40 100 61 31 52]
 [ 28 7 84 101 95]
 [123 75 40 49 46]]

Layer 4:
[[ 54 106 6 121 119]
 [ 80 30 76 48 73]
 [ 67 116 76 110 49]
 [ 88 28 64 78 6]
 [ 45 79 82 66 37]]

Layer 5:
[[ 87 25 84 78 3]
 [ 83 58 26 23 124]
 [ 34 63 67 30 101]
 [ 21 124 37 65 33]
 [ 95 41 52 125 53]]

```

- Grafik objective function value terhadap iterasi



- Waktu dan final objective function

Final Objective Function Value: 2638

Duration: 2.23 seconds

## 2. Iterasi 500

- Initial state cube

```

INITIAL STATE:
-----

Layer 1:
[[ 19 23 6 78 96]
 [ 88 102 57 43 54]
 [ 66 97 76 119 5]
 [ 35 71 55 121 92]
 [ 95 52 93 118 22]]

Layer 2:
[[ 30 39 20 7 100]
 [ 41 21 108 44 28]
 [105 106 122 82 109]
 [ 56 68 17 103 85]
 [ 87 101 1 110 62]]

Layer 3:
[[120 70 34 15 65]
 [ 53 94 64 26 81]
 [117 24 116 10 29]
 [ 16 114 49 112 58]
 [ 14 84 11 40 69]]

Layer 4:
[[124 125 115 37 3]
 [ 75 31 113 45 86]
 [ 89 12 98 83 59]
 [ 42 72 60 79 111]
 [ 18 38 50 90 4]]

Layer 5:
[[ 77 9 63 36 27]
 [123 51 48 61 67]
 [104 2 74 99 80]
 [107 32 25 33 8]
 [ 91 73 13 46 47]]

Initial Objective Function Value: 6890

```

- Final state cube

```

FINAL STATE:
-----

Layer 1:
[[108 123 57 18 16]
 [ 68 3 124 89 25]
 [ 12 52 44 124 86]
 [ 64 71 53 51 77]
 [ 66 66 39 35 107]]

Layer 2:
[[ 85 44 62 53 71]
 [ 11 20 105 85 89]
 [110 80 57 28 42]
 [ 77 66 20 96 57]
 [ 27 103 71 52 60]]

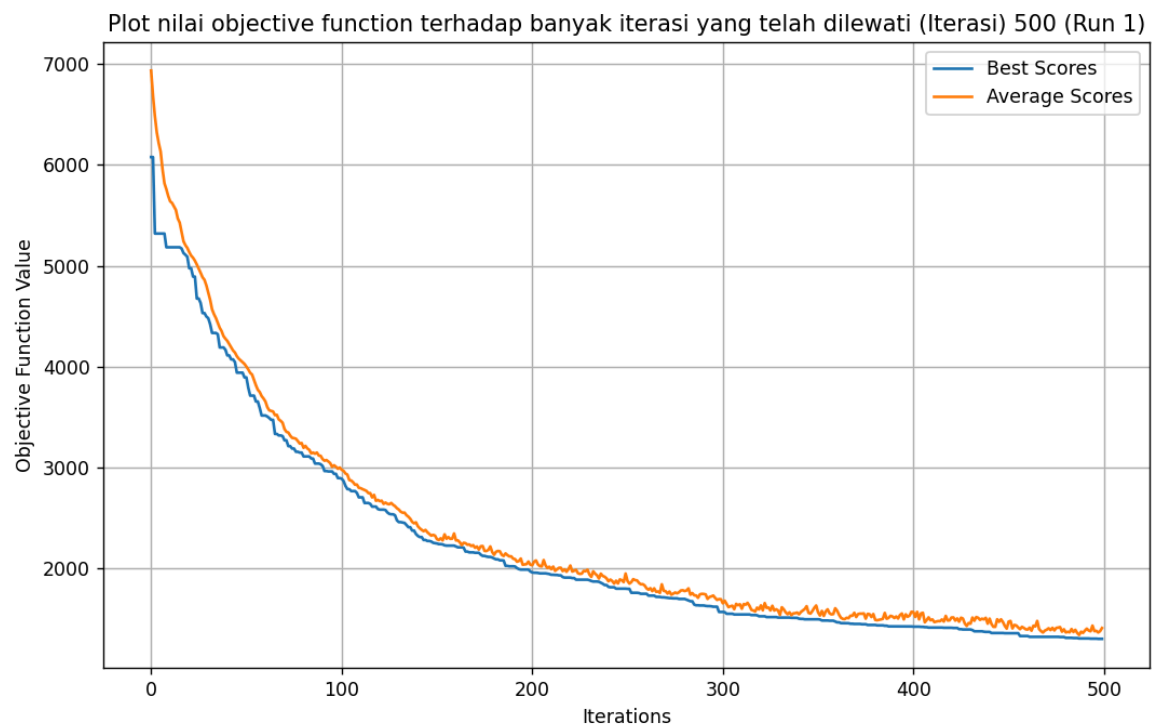
Layer 3:
[[ 20 66 99 65 64]
 [ 42 105 11 62 96]
 [ 80 25 80 81 44]
 [106 44 102 16 45]
 [ 66 74 25 91 66]]

Layer 4:
[[ 86 106 12 64 49]
 [ 80 90 51 83 12]
 [ 17 64 60 71 103]
 [ 64 38 64 41 102]
 [ 66 16 125 53 42]]

Layer 5:
[[ 16 13 123 106 60]
 [106 25 25 3 106]
 [ 89 102 77 12 38]
 [ 4 97 77 106 42]
 [ 84 77 9 86 68]]

```

- Grafik objective function value terhadap iterasi



- Waktu dan final objective function

```
Final Objective Function Value: 1298  
Duration: 11.30 seconds
```

### 3. Iterasi 1000

- Initial state cube

```

INITIAL STATE:
-----

Layer 1:
[[112 91 32 103 105]
 [ 71 96 35 111 100]
 [  5 92 116 125 123]
 [ 93 30 10 43 24]
 [  4 62 47 25 87]]

Layer 2:
[[ 60 46 118  2 28]
 [ 68 26 48 65 23]
 [102 54 42 117 88]
 [ 67 53 31 38 37]
 [ 85 57 77 120 113]]

Layer 3:
[[ 86 76 11 17 41]
 [ 27 115 98 13 12]
 [ 59  3 72 15  7]
 [ 21 64 16 114  6]
 [ 56 124 110 83 89]]

Layer 4:
[[104 40 39 69 63]
 [ 75 51 79 58 122]
 [ 55 36 99 20 45]
 [121 61 78 107 44]
 [ 73 18 49 97 106]]

Layer 5:
[[109 34 19 29 74]
 [ 66 101 33 94 84]
 [108 82 70 95  8]
 [ 90  9 22 81 14]
 [  1 80 50 52 119]]

Initial Objective Function Value: 7842

```

- Final state cube

```

FINAL STATE:
-----

Layer 1:
[[ 79 44 26 122 44]
 [  8 69 117 24 97]
 [ 81 35 86 99 18]
 [ 90 86  8 37 95]
 [ 57 76 87 35 60]]

Layer 2:
[[ 27 37 106 68 49]
 [ 53 95 90 20 90]
 [ 95 74 26 64 57]
 [ 20 99 45 106 50]
 [117  8 64 62 69]]

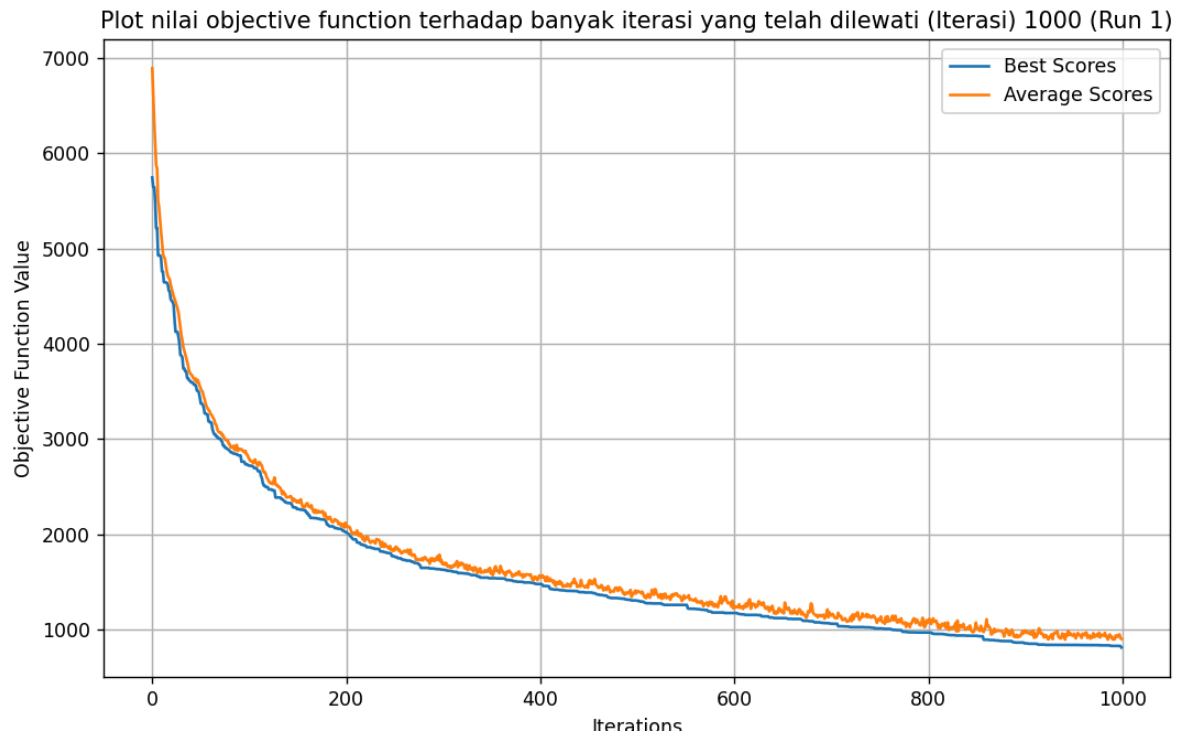
Layer 3:
[[ 59 45 68 74 69]
 [ 58 74 69 76 26]
 [ 37 57 44 86 75]
 [106 71 26 27 86]
 [ 55 68 73 53 67]]

Layer 4:
[[ 90 76 20 43 87]
 [110 43 11 101 53]
 [ 35 122 81 17 60]
 [ 74 26 124 37 49]
 [ 13 49 75 117 64]]

Layer 5:
[[ 59 112 95  7 65]
 [ 85 37 27 92 49]
 [ 67 27 59 59 106]
 [ 34 27 110 106 37]
 [ 73 112 20 52 57]]

```

- Grafik objective function value terhadap iterasi



- Waktu dan final objective function

Final Objective Function Value: 816

Duration: 22.57 seconds

Berdasarkan eksperimen yang dilakukan, terlihat bahwa variasi ukuran populasi dan jumlah iterasi memberikan dampak yang kuat terhadap nilai objective function akhir dan durasi waktu. Dalam eksperimen dengan iterasi yang terkontrol, dilakukan variasi ukuran populasi. dapat dilihat dari hasil *screenshot* tersebut, semakin besar ukuran populasi maka akan menghasilkan nilai objective function yang lebih baik (lebih kecil). Akan tetapi waktu komputasi akan menjadi lebih lama. ini karena populasi yang lebih besar akan memberikan ruang eksplor yang lebih luas. Namun peningkatan populasi diikuti dengan peningkatan waktu komputasi yang terlihat pada bukti bahwa semakin banyak populasi waktu juga akan semakin lama.

Pada eksperimen dengan populasi yang terkontrol dilakukan variasi jumlah iterasi. dari hasil ini terlihat bahwa peningkatan jumlah iterasi berbanding lurus dengan perbaikan nilai objective function. semakin banyak iterasinya maka akan semakin besar kemungkinan algoritma dalam melakukan eksplorasi untuk menghasilkan solusi yang lebih baik. sama seperti peningkatan populasi, peningkatan iterasi juga akan semakin banyak memakan waktu.

Jika dilihat dari plot yang dihasilkan, dapat diamati pola konvergensi yang konsisten dimana terjadi penurunan nilai objective function yang cepat di awal iterasi, tapi kemudian melambat dan cenderung stabil di akhir iterasi. Ini menunjukkan algoritma genetik mampu menemukan solusi yang bagus dengan cepat di awal proses, namun memerlukan proses yang lebih banyak untuk menemukan solusi yang lebih baik.

## ● Kesimpulan dan Saran

Permasalahan magic cube dalam penelitian ini berfokus ke bagaimana cara menyusun angka dalam struktur kubus  $5 \times 5 \times 5$ , dengan tujuan memenuhi kriteria dimana jumlah angka di setiap baris, kolom, tiang, dan diagonal harus sama dengan "magic number". Dalam upaya menyelesaikan permasalahan ini, ada enam algoritma yang telah dicoba untuk diimplementasikan: Steepest Ascent Hill-Climbing, Hill-Climbing dengan Sideways Move, Random Restart Hill-Climbing, Stochastic Hill-Climbing, Simulated Annealing (SA) dan Genetic Algorithm (GA). SA menunjukkan kelebihan dalam menghindari local optima, sementara GA memberikan hasil yang bervariasi tergantung pada ukuran populasi yang digunakan. Terkait algoritma local search hill climbing, semuanya menunjukkan peningkatan yang signifikan pada iterasi awal tetapi mengalami kesulitan dalam mencapai global optima.

Kinerja dari algoritma-algoritma ini sangat dipengaruhi oleh pemilihan parameter, seperti initial temperature dan cooling rate untuk SA, serta ukuran populasi dan tingkat mutasi untuk GA. Mengingat hal tersebut, diperlukan eksperimen lebih lanjut untuk menentukan parameter optimal untuk setiap algoritma, mengingat pengaruhnya yang signifikan terhadap hasil akhir. Saran lain adalah terkait kemungkinan untuk mengkombinasikan antar algoritma-algoritma, misalnya menggabungkan aspek Random Restart Hill-Climbing pada Simulated Annealing yang kemungkinan dapat menjadi solusi untuk mengatasi keterbatasan algoritma. Selanjutnya, terkait perhitungan entropi di SA, diperlukan suatu cara untuk membatasi



hasil delta E, dikarenakan seringkali apabila cooling rate terlalu drastis atau initial temperature terlalu rendah, akan menyebabkan error math overflow yang diakibatkan oleh penghitungan pangkat yang terlalu besar. Terakhir, analisis lebih mendalam terhadap hasil yang diperoleh sangat diperlukan untuk memahami lebih baik bagaimana performa setiap algoritma dan mengidentifikasi kekuatan serta kelemahan masing-masing algoritma secara lebih spesifik.

### ● **Pembagian Tugas**

NIM	Nama	Tugas
18222019	Jonathan Wiguna	<ol style="list-style-type: none"> <li>1. General function</li> <li>2. Simulated annealing</li> <li>3. Laporan</li> </ol>
18222052	Muhammad Yaafi Wasesa Putra	<ol style="list-style-type: none"> <li>1. Membuat algoritma Genetik</li> <li>2. Laporan</li> </ol>
18222058	Matthew Nicholas Gunawan	<ol style="list-style-type: none"> <li>1. Membuat algoritma genetic</li> <li>2. Membuat main</li> <li>3. Laporan</li> </ol>
18222081	Harry Truman Suhalim	<ol style="list-style-type: none"> <li>1. Steepest Hill Climbing</li> <li>2. Hill-Climbing dengan Sideways Move</li> <li>3. Random Restart Hill-Climbing,</li> <li>4. Stochastic Hill-Climbing</li> <li>5. Laporan</li> </ol>

## Referensi

W3Schools. "Python yield Keyword." W3Schools,  
[https://www.w3schools.com/python/ref\\_keyword\\_yield.asp](https://www.w3schools.com/python/ref_keyword_yield.asp).

GeeksforGeeks. "Genetic Algorithms." GeeksforGeeks,  
<https://www.geeksforgeeks.org/genetic-algorithms/>.

Stack Overflow. "Stochastic Hill Climbing vs Random Restart Hill Climbing Algorithms." Stack Overflow,  
<https://stackoverflow.com/questions/49595577/stochastic-hill-climbing-vs-random-restart-hill-climbing-algorithms>.

Algorithm Afternoon. "Stochastic Hill Climbing with Random Restarts." Algorithm Afternoon,  
[https://algorithmafternoon.com/stochastic/stochastic\\_hill\\_climbing\\_with\\_random\\_restarts/](https://algorithmafternoon.com/stochastic/stochastic_hill_climbing_with_random_restarts/).


Magisch Vierkant. "Features of the magic cube" *Magisch Vierkant*,  
<https://www.magischvierkant.com/three-dimensional-eng/magic-features/>.

GeeksforGeeks. "Local Search Algorithm in Artificial Intelligence." GeeksforGeeks,  
<https://www.geeksforgeeks.org/local-search-algorithm-in-artificial-intelligence/>.

TutorialsPoint. "Genetic Algorithms - Parent Selection." TutorialsPoint,  
[https://www.tutorialspoint.com/genetic\\_algorithms/genetic\\_algorithms\\_parent\\_selection.htm](https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm).

GeeksforGeeks. "Python Slicing Multi Dimensional Arrays." GeeksforGeeks,  
<https://www.geeksforgeeks.org/python-slicing-multi-dimensional-arrays/>.

 simulated annealing

 hillClimbing 8 queens