

# **Tugas Besar 1**

## **Pencarian Solusi Diagonal Magic Cube dengan Local Search**

**IF3170 Inteligensi Artifisial**



Wilson Yusda 13522019

Filbert 13522021

Elbert Chailes 13522045

Benardo 13522055

**Program Studi Teknik Informatika**  
**Sekolah Teknik Elektro dan Informatika**  
**Institut Teknologi Bandung**  
**2024**

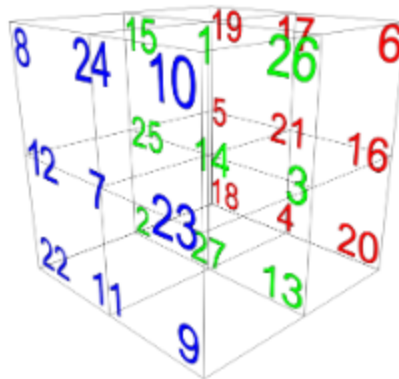
# DAFTAR ISI

<b>BAB I : DESKRIPSI PERSOALAN.....</b>	<b>3</b>
<b>BAB II : PEMBAHASAN.....</b>	<b>5</b>
2.1 Pemilihan Objective Function.....	5
2.2 Penjelasan implementasi algoritma local search.....	9
2.2.1 Hill Climbing.....	11
2.2.1.1 Steepest Ascent Hill Climbing.....	18
2.2.1.2 Hill Climbing With Sideways Move.....	21
2.2.1.3 Random Restart Hill Climbing.....	24
2.2.1.4 Stochastic Hill Climbing.....	24
2.2.2 Simulated Annealing.....	27
2.2.3 Genetic Algorithm.....	30
2.3. Hasil eksperimen dan analisis.....	36
2.3.1 Hill Climbing.....	36
2.3.1.1 Steepest Ascent Hill Climbing.....	36
2.3.1.1.1 Hasil Analisis Steepest Ascent Hill Climbing.....	39
2.3.1.2 Hill Climbing With Sideways Move.....	40
2.3.1.2.1 Hasil Analisis Hill Climbing With Sideways Move.....	43
2.3.1.3 Random Restart Hill Climbing.....	43
2.3.1.3.1 Hasil Analisis Hill Climbing With Random Restart.....	56
2.3.1.4 Stochastic Hill Climbing.....	57
2.3.1.4.1 Hasil Analisis Stochastic Hill Climbing.....	60
2.3.2 Simulated Annealing Algorithm.....	60
2.3.2.1. Hasil Analisis Simulated Annealing.....	65
2.3.3 Genetic Algorithm.....	66
2.3.3.1 Hasil Analisis Genetic Algorithm.....	71
2.3.4 Perbandingan tiap algoritma.....	72
<b>BAB III : KESIMPULAN DAN SARAN.....</b>	<b>74</b>
<b>BAB IV : PEMBAGIAN TUGAS TIAP KELOMPOK.....</b>	<b>76</b>
<b>REFERENSI.....</b>	<b>77</b>

## BAB I : DESKRIPSI PERSOALAN

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

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



**Gambar 1.1.** Magic cube 3x3x3

- Terdapat 9 potongan bidang, yaitu:

8	24	10
12	7	23
22	11	9
15	1	26
25	14	3
2	27	13
19	17	6
5	21	16
18	4	20
19	17	6
15	1	26
8	24	10
5	21	16
25	14	3
12	7	23
18	4	20
2	27	13
22	11	9
8	15	19
24	1	17
10	26	6
12	25	5
7	14	21
23	3	16
22	2	18
11	27	4
9	13	20

**Gambar 1.2.** Potongan Bidang Magic cube

- Diagonal yang dimaksud adalah yang dilingkari warna merah saja

## BAB II : PEMBAHASAN

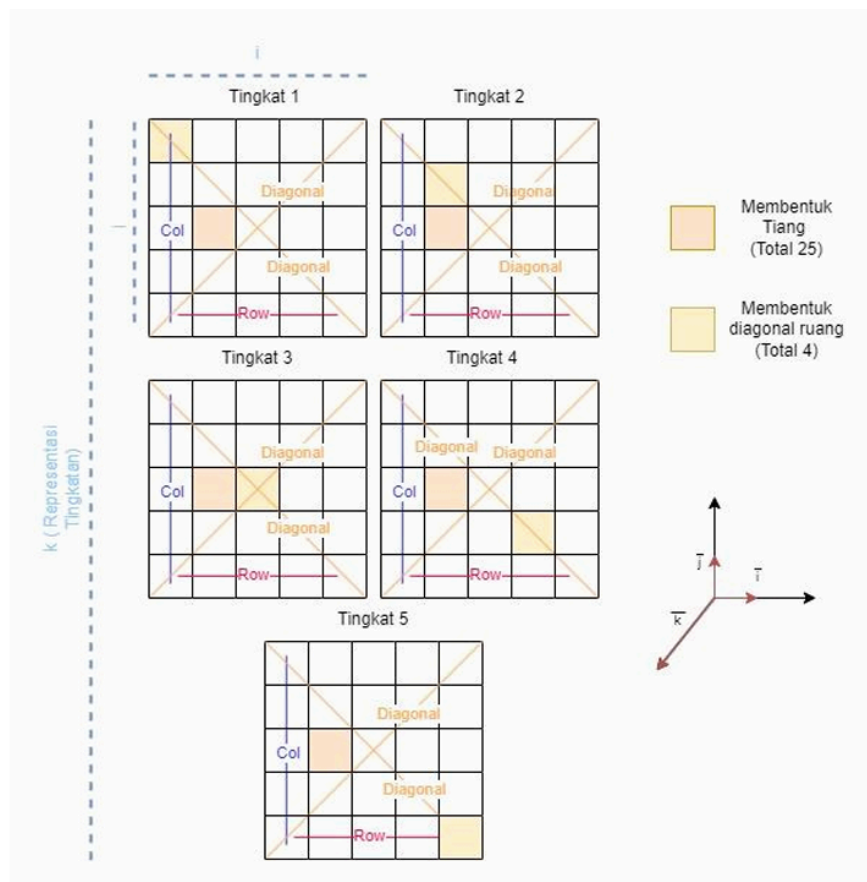
### 2.1 Pemilihan Objective Function

Untuk menyelesaikan persoalan Diagonal Magic Cube 5x5x5 seperti yang telah dijelaskan pada Bab 1, penulis menggunakan objective function seperti berikut.

$$\text{objective function} = Z = Drow + Dcolumn + Dpillar + Ddiagonal + Dtriagonal$$

**Formula 2.1.1.** Rumus untuk fungsi objektif (Z)

Berdasarkan rumus yang ditulis pada Formula 2.1.1., terdapat beberapa notasi, seperti D yang berarti deviation atau penyimpangan nilai, row yang berarti baris, column yang berarti kolom, pillar yang berarti tiang, diagonal yang diagonal yang berarti diagonal sisi, serta triagonal yang berarti diagonal ruang. Ilustrasi untuk setiap permasalahan variabel akan diilustrasikan dalam gambar berikut.



**Gambar 2.1.1.** Definisi dan ilustrasi gambar diagonal magic cube secara 2D

Dengan hasil ilustrasi tersebut, maka dapat disimpulkan bahwa sebuah Magic Cube 5x5x5 akan memiliki sebanyak 25 row, 25 column, 25 pillar, 30 diagonal, dan 4 triagonal. Pada konsep rumus ini, objective function dihitung dengan mempertimbangkan deviasi untuk setiap variabel yang dipertimbangkan. Maka dari itu, cara untuk memperhitungkan nilai D<sub>row</sub> adalah dengan menghitung total deviasi yang pada total 25 row. Cara yang sama dilakukan untuk D<sub>column</sub>, D<sub>pillar</sub>, D<sub>diagonal</sub>, dan D<sub>triagonal</sub>. Dengan begitu, rumus untuk perhitungan total deviasi untuk setiap variabel dapat ditulis secara matematis seperti berikut.

$$D_{row} = \sum_{k=1}^5 \sum_{j=1}^5 \left| \left( \sum_{i=1}^5 C_{ijk} \right) - 315 \right|$$

**Formula 2.1.2.** Rumus untuk menghitung variabel  $D_{row}$

$$D_{column} = \sum_{k=1}^5 \sum_{i=1}^5 \left| \left( \sum_{j=1}^5 C_{ijk} \right) - 315 \right|$$

**Formula 2.1.3.** Rumus untuk menghitung variabel  $D_{column}$

$$D_{pillar} = \sum_{i=1}^5 \sum_{j=1}^5 \left| \left( \sum_{k=1}^5 C_{ijk} \right) - 315 \right|$$

**Formula 2.1.4.** Rumus untuk menghitung variabel  $D_{pillar}$

Proses yang direpresentasikan oleh Formula 2.1.2. adalah dengan melakukan pencarian seluruh kemungkinan koordinat  $jk$  dengan menggunakan  $\sum_{k=1}^5 \sum_{j=1}^5$ , yang mana akan menghasilkan 25 kombinasi koordinat  $jk$ , yang sesuai dengan jumlah row sebanyak 25 pada Magic Cube. Setiap kombinasi row yang terdiri dari 5 cell, akan dijumlahkan menghasilkan angka yang disebut sebagai row constant. Setiap row itu akan menghasilkan row constant yang memiliki perbedaan atau jarak nilai dengan magic constant ( $S = 315$ ), yang disebut deviasi oleh penulis. Total deviasi pada 25 row tersebut yang kemudian disebut menjadi  $D_{Row}$ .

Konsep yang sama juga diterapkan terhadap  $D_{column}$  dan  $D_{pillar}$  seperti yang telah dipresentasikan oleh Formula 2.1.3. dan Formula 2.1.4. untuk menghasilkan berturut-turut total deviasi nilai column constant dan total deviasi nilai pillar constant. Sebagai tambahan informasi, bahwa magic constant ( $S = 315$ ) dengan variabel  $m$  adalah dimensi kubus dapat didapatkan menggunakan rumus seperti berikut.

$$S = \frac{m \cdot (m^3 + 1)}{2}$$

**Formula 2.1.5.** Rumus untuk menghitung *magic constant* (S)

Selain itu, terkait dengan perhitungan  $D_{diagonal}$  yang mana pada kubus 5x5x5 memiliki total 30 diagonal. Perhitungan untuk jumlah jarak antara *diagonal constant* yang dihasilkan dari 30 diagonal dengan *magic constant* (S) dapat diformulasikan dalam formulasi berikut.

*Bidang XY (ij)*

$$D_{diagonal-1-xy} = \sum_{k=1}^5 \left| (C_{1,1,k} + C_{2,2,k} + C_{3,3,k} + C_{4,4,k} + C_{5,5,k}) - 315 \right|$$

$$D_{diagonal-2-xy} = \sum_{k=1}^5 \left| (C_{1,5,k} + C_{2,4,k} + C_{3,3,k} + C_{4,2,k} + C_{5,1,k}) - 315 \right|$$

*Bidang YZ (jk)*

$$D_{diagonal-1-yz} = \sum_{i=1}^5 \left| (C_{i,1,1} + C_{i,2,2} + C_{i,3,3} + C_{i,4,4} + C_{i,5,5}) - 315 \right|$$

$$D_{diagonal-2-yz} = \sum_{i=1}^5 \left| (C_{i,1,5} + C_{i,2,4} + C_{i,3,3} + C_{i,4,2} + C_{i,5,1}) - 315 \right|$$

*Bidang XZ (ik)*

$$D_{diagonal-1-xz} = \sum_{j=1}^5 \left| (C_{1,j,1} + C_{2,j,2} + C_{3,j,3} + C_{4,j,4} + C_{5,j,5}) - 315 \right|$$

$$D_{diagonal-2-xz} = \sum_{j=1}^5 \left| (C_{1,j,5} + C_{2,j,4} + C_{3,j,3} + C_{4,j,2} + C_{5,j,1}) - 315 \right|$$

Maka,

$$D_{diagonal} = D_{diagonal-1-xy} + D_{diagonal-2-xy} + D_{diagonal-1-yz} + D_{diagonal-2-yz} + D_{diagonal-1-xz} + D_{diagonal-2-xz}$$

**Formula 2.1.6.** Rumus untuk menghitung deviasi diagonal pada kubus 5x5x5

Terakhir, pada Magic Cube 5x5x5 terdapat sebanyak 4 diagonal ruang yang perlu dihitung deviasinya untuk menghasilkan nilai dari variabel  $D_{triagonal}$ . Perhitungan jumlah nilai

yang dibentuk *triagonal* atau *triagonal constant* tidak dapat diabstraksi dengan menggunakan notasi matematika. Sehingga, rumusnya dapat diformulasikan seperti berikut.

$$D_{triagonal-1} = C_{1,1,1} + C_{2,2,2} + C_{3,3,3} + C_{4,4,4} + C_{5,5,5}$$

$$D_{triagonal-2} = C_{1,1,5} + C_{2,2,4} + C_{3,3,3} + C_{4,4,2} + C_{5,5,1}$$

$$D_{triagonal-3} = C_{1,5,1} + C_{2,4,2} + C_{3,3,3} + C_{4,2,4} + C_{5,1,5}$$

$$D_{triagonal-4} = C_{5,1,1} + C_{4,2,2} + C_{3,3,3} + C_{2,4,4} + C_{1,5,5}$$

Maka,

$$D_{triagonal} = D_{triagonal-1} + D_{triagonal-2} + D_{triagonal-3} + D_{triagonal-4}$$

**Formula 2.1.7.** Rumus untuk menghitung variabel  $D_{triagonal}$

Dengan hasil pembuatan rumus yang merepresentasikan nilai jarak atau deviasi yang terjadi pada *row*, *column*, *pillar*, *diagonal*, dan *triagonal*. Pada akhirnya, nilai *objective function* dapat dicari dengan terlebih dahulu mencari nilai-nilai variabel yang telah didefinisikan dan telah diberikan formula untuk perhitungannya pada Formula 2.1.2., Formula 2.1.3. Formula 2.1.4., Formula 2.1.6, dan Formula 2.1.7.

Jadi, inti dari pembuatan *objective function* adalah menggunakan seluruh kemungkinan *row*, *column*, *pillar*, *diagonal*, maupun *triagonal* yang dapat terbentuk pada kubus 5x5x5 yang berturut-turut 25, 25, 25, 30, dan 4. Kemudian, menghitung seluruh *constant value* baik dari *row constant*, *column constant*, *pillar constant*, *diagonal constant*, maupun *triagonal constant*. Lalu, setiap *constant value* yang didapatkan dikurangi dengan *magic constant* (S) untuk mendapatkan jarak angka atau disebut deviasi oleh penulis, menghasilkan total Z atau total deviasi yang terjadi. Oleh karena itu, nilai *objective function* yang mungkin berada pada batas  $(-\infty, 0]$ , dengan jarak yang terjadi diberikan tanda negatif dan jarak 0 berarti *state* telah mencapai *magic cube* karena tidak ada *constant value* yang memiliki deviasi dengan *magic constant*. Untuk setiap *objective function* yang memiliki deviasi, atau memiliki value  $>0$ , akan ditambahkan negatif pada angka tersebut, sehingga untuk *state* yang masih memiliki deviasi akan mempunyai nilai *objective function*  $< 0$ .



## 2.2 Penjelasan implementasi algoritma local search

Tugas ini mengimplementasikan 3 jenis algoritma local search, yakni *Hill Climbing*, *Simulated Annealing*, dan *Genetic Algorithm*. Sebelum itu, terdapat implementasi kelas *general* yang akan digunakan dalam kelas lain untuk mengurangi redundansi pemrograman.

```
import random
import numpy as np
import matplotlib.pyplot as plt

class CubeSolver:
    MAGIC_CONSTANT = 315

    def __init__(self, state=None):
        if state is None:
            self.state = self.generate_random_initial_state()
        else:
            self.state = state

    def calculate_objective(self, state):
        objective = 0
        #Untuk Perhitungan Baris
        for layer in range(5):
            for row in range(5):
                row_sum = sum(state[layer * 25 + row * 5 + col]
for col in range(5))
                objective += abs(row_sum - self.MAGIC_CONSTANT)
        #Untuk Perhitungan Kolom
        for layer in range(5):
            for col in range(5):
                col_sum = sum(state[layer * 25 + row * 5 + col]
for row in range(5))
                objective += abs(col_sum - self.MAGIC_CONSTANT)
        #Untuk Perhitungan Pillar
        for row in range(5):
            for col in range(5):
                pillar_sum = sum(state[layer * 25 + row * 5 +
col] for layer in range(5))
                objective += abs(pillar_sum -
self.MAGIC_CONSTANT)
        # Perhitungan Diagonal dalam Layer (Diagonal
Calculations within Layers)
        for layer in range(5):
            diag1_sum = sum(state[layer * 25 + i * 6] for i in
range(5))
            diag2_sum = sum(state[layer * 25 + (i + 1) * 4] for
```

```

i in range(5))
        objective += abs(diag1_sum - self.MAGIC_CONSTANT)
        objective += abs(diag2_sum - self.MAGIC_CONSTANT)
        #Perhitungan Diagonal Vertikal dalam Kolom (Vertical
        Diagonal Calculations across Columns)
        for col in range(5):
            diag3_sum = sum(state[layer * 25 + layer * 5 + col]
for layer in range(5))
            diag4_sum = sum(state[layer * 25 + (4 - layer) * 5
+ col] for layer in range(5))
            objective += abs(diag3_sum - self.MAGIC_CONSTANT)
            objective += abs(diag4_sum - self.MAGIC_CONSTANT)
            # Perhitungan Diagonal Vertikal dalam Baris (Vertical
            Diagonal Calculations across Rows)
            for row in range(5):
                diag5_sum = sum(state[layer * 25 + row * 5 + layer]
for layer in range(5))
                diag6_sum = sum(state[layer * 25 + row * 5 + (4 -
layer)] for layer in range(5))
                objective += abs(diag5_sum - self.MAGIC_CONSTANT)
                objective += abs(diag6_sum - self.MAGIC_CONSTANT)
            # Perhitungan Diagonal Ruang (Triagonal Calculations)
            triangular_indices = [
                [i * 31 for i in range(5)],
                [4, 33, 62, 91, 120],
                [20, 41, 62, 83, 104],
                [100, 81, 62, 43, 24]
            ]

            for indices in triangular_indices:
                triangular_sum = sum(state[idx] for idx in indices)
                objective += abs(triangular_sum -
self.MAGIC_CONSTANT)

        return -objective

    def generate_random_initial_state(self):
        state = list(range(1, 126))
        random.shuffle(state)
        return state

    def visualize_state(self, state, axes=None):
        layers = [np.array(state[i * 25:(i + 1) *
25]).reshape(5, 5) for i in range(5)]

        if axes is None:

```

```

fig, axes = plt.subplots(1, 5, figsize=(15, 3))

for i, layer in enumerate(layers):
    ax = axes[i]
    ax.matshow(layer, cmap='gray', vmin=0, vmax=1)
    for (row, col), val in np.ndenumerate(layer):
        ax.text(col, row, f'{val}', va='center',
ha='center', color='black', fontsize=6)
    ax.set_title(f'Layer {i + 1}', fontsize=8)
    ax.axis('off')

if axes is None:
    plt.suptitle('5x5x5 Cube Visualization')
    plt.show()

```

Kelas ini mendefinisikan sifat dari sebuah *magic cube*. Secara spesifik, mendefinisikan sebuah *magic cube* 5x5x5 dengan *magic constant* sebesar 315. Adapun fungsi yang terdefinisi yaitu:

1. Fungsi konstruktor dari kelas ini adalah dengan membentuk kubus dari kumpulan sub-kubus lainnya secara acak dengan penanda label dari angka 1 sampai dengan 125, kecuali memang dispesifikasikan *state* awal. Konstruktor memanggil fungsi *generate\_random\_initial\_state* dalam mengimplementasikan konsep tersebut.
2. Fungsi *calculate\_objective* yang menerima parameter *state* yang akan menghitung nilai *objective* sesuai dengan pengertian yang telah dijelaskan sebelumnya. Untuk lebih memperjelas pengertian mengenai perhitungan fungsi objektif, dapat dilihat pada potongan kode diatas dengan untuk tiap sub-bagian sudah dilabeli dengan penanda berwarna hijau.
3. Fungsi *generate\_random\_initial\_state* memberikan susunan kubus dalam bentuk *array* dari angka 1 sampai 125 secara acak.
4. Fungsi *visualize\_state* yang menerima *state* dan *axes* (opsional) yang berfungsi untuk memvisualisasikan kubus sesuai dengan *state* inputan untuk masing masing layer.

### 2.2.1 Hill Climbing

Implementasi Hill Climbing dilakukan pada satu kelas dengan berbagai fungsi yang merepresentasikan berbagai metode *hill climbing*.

```

import random
import matplotlib.pyplot as plt
from base_solver import CubeSolver
from datetime import datetime

class HillClimbingSolver(CubeSolver):
    def __init__(self, state=None):
        super().__init__(state)

    def generate_neighbors(self, state):
        neighbors = []
        for i in range(len(state)):
            for j in range(i + 1, len(state)):
                neighbor = state.copy()
                neighbor[i], neighbor[j] = neighbor[j],
neighbor[i]
                neighbors.append(neighbor)
        return neighbors

    def steepest_ascent_hill_climbing(self, title = "Figure
1"):
        current_state = self.generate_random_initial_state()
        initial_state = current_state.copy()
        current_value = self.calculate_objective(current_state)
        iteration = 0
        iterations = []
        objective_values = []

        start_time = datetime.now()

        plt.ion()
        fig = plt.figure(figsize=(10, 8))
        fig.canvas.manager.set_window_title(title)

        gs = fig.add_gridspec(nrows=2, ncols=1,
height_ratios=[1, 2])

        ax1 = fig.add_subplot(gs[0])
        ax1.set_xlabel('Iteration')
        ax1.set_ylabel('Objective Value')
        ax1.set_title('Steepest Ascent Hill Climbing Objective
Value over Iterations')

        gs2 = gs[1].subgridspec(1, 5)
        cube_axes = [fig.add_subplot(gs2[0, i]) for i in
range(5)]

```

```

self.visualize_state(current_state, axes=cube_axes)
plt.pause(0.1)

while True:
    iterations.append(iteration)
    objective_values.append(current_value)

    ax1.clear()
    ax1.plot(iterations, objective_values,
color='blue')
    ax1.set_xlabel('Iteration')
    ax1.set_ylabel('Objective Value')
    ax1.set_title('Steepest Ascent Hill Climbing
Objective Value over Iterations')

    for ax in cube_axes:
        ax.clear()
    self.visualize_state(current_state, axes=cube_axes)

    plt.pause(0.1)

    neighbors = self.generate_neighbors(current_state)
    best_neighbor = None
    best_value = current_value

    for neighbor in neighbors:
        neighbor_value =
self.calculate_objective(neighbor)
        if neighbor_value > best_value:
            best_value = neighbor_value
            best_neighbor = neighbor

    if best_value <= current_value:
        plt.ioff()
        fig2 = plt.figure(figsize=(12, 6))
        fig2.suptitle('Initial and Final States')

        gs_init_final = fig2.add_gridspec(2, 5)

        axes_initial =
[fig2.add_subplot(gs_init_final[0, i]) for i in range(5)]
        axes_final = [fig2.add_subplot(gs_init_final[1,
i]) for i in range(5)]

        self.visualize_state(initial_state,

```

```

axes=axes_initial)
        self.visualize_state(current_state,
axes=axes_final)

        end_time = datetime.now()
        duration = (end_time -
start_time).total_seconds() * 1000

        fig2.text(0.5, 0.05, f'Final Objective Value:
{best_value} | Duration: {duration} ms', ha='center',
fontsize=10)

        for ax in axes_initial:
            ax.set_title('Initial State', fontsize=8)
        for ax in axes_final:
            ax.set_title('Final State', fontsize=8)

        plt.show()
        return current_state, iteration

    current_state = best_neighbor
    current_value = best_value
    iteration += 1

    def sideways_move_hill_climbing(self, title = "Sideways
Move Hill Climbing"):
        current_state = self.generate_random_initial_state()
        current_value = self.calculate_objective(current_state)
        initial_state = current_state.copy()
        iteration = 0
        iterations = []
        objective_values = []

        start_time = datetime.now()

        plt.ion()
        fig = plt.figure(figsize=(10, 8))
        fig.canvas.manager.set_window_title(title)

        gs = fig.add_gridspec(nrows=2, ncols=1,
height_ratios=[1, 2])

        ax1 = fig.add_subplot(gs[0])
        ax1.set_xlabel('Iteration')
        ax1.set_ylabel('Objective Value')
        ax1.set_title('Sideways Move Hill Climbing Objective

```

```

Value over Iterations')

    gs2 = gs[1].subgridspec(1, 5)
    cube_axes = [fig.add_subplot(gs2[0, i]) for i in
range(5)]

    self.visualize_state(current_state, axes=cube_axes)
    plt.pause(0.1)

    while True:
        iterations.append(iteration)
        objective_values.append(current_value)

        ax1.clear()
        ax1.plot(iterations, objective_values,
color='blue')
        ax1.set_xlabel('Iteration')
        ax1.set_ylabel('Objective Value')
        ax1.set_title('Sideways Move Hill Climbing
Objective Value over Iterations')

        for ax in cube_axes:
            ax.clear()
        self.visualize_state(current_state, axes=cube_axes)

        plt.pause(0.1)

        neighbors = self.generate_neighbors(current_state)
        best_neighbor = None
        best_value = float('-inf')

        for neighbor in neighbors:
            neighbor_value =
self.calculate_objective(neighbor)
            if neighbor_value > best_value:
                best_value = neighbor_value
                best_neighbor = neighbor

        if best_value < current_value:
            plt.ioff()
            fig2 = plt.figure(figsize=(12, 6))
            fig2.suptitle('Initial and Final States')

            gs_init_final = fig2.add_gridspec(2, 5)

            axes_initial =

```

```

[fig2.add_subplot(gs_init_final[0, i]) for i in range(5)]
        axes_final = [fig2.add_subplot(gs_init_final[1,
i]) for i in range(5)]

        self.visualize_state(initial_state,
axes=axes_initial)
        self.visualize_state(current_state,
axes=axes_final)

        end_time = datetime.now()
        duration = (end_time -
start_time).total_seconds() * 1000

        fig2.text(0.5, 0.05, f'Final Objective Value:
{best_value} | Duration: {duration} ms', ha='center',
fontsize=10)

        for ax in axes_initial:
            ax.set_title('Initial State', fontsize=8)
        for ax in axes_final:
            ax.set_title('Final State', fontsize=8)

        plt.show()
        return current_state, iteration

    current_state = best_neighbor
    current_value = best_value
    iteration += 1

def random_restart_hill_climbing(self, max_restart):
    all_state = []
    for i in range(max_restart):
        title = f"Restart {i + 1}"
        final_state , iteration =
self.steepest_ascent_hill_climbing(title)
        print("Num of iteration for restart ", i+1 , ": " ,
iteration)
        all_state.append([final_state,iteration])

    return all_state[-1][0], all_state[-1][1]

def stochastic_hill_climbing(self, max_iteration = 100):
    current_state = self.generate_random_initial_state()
    initial_state = current_state.copy()
    current_value = self.calculate_objective(current_state)

```



```

        iteration = 0
        iterations = []
        objective_values = []

        start_time = datetime.now()

        plt.ion()
        fig= plt.figure(figsize=(10, 8))
        gs = fig.add_gridspec(nrows=2, ncols=1,
height_ratios=[1, 2])

        ax1 = fig.add_subplot(gs[0])
        ax1.set_xlabel('Iteration')
        ax1.set_ylabel('Objective Value')

        ax1.set_title('Stochastic Hill Climbing Objective Value
over Iterations')

        gs2 = gs[1].subgridspec(1, 5)
        cube_axes = [fig.add_subplot(gs2[0, i]) for i in
range(5)]

        self.visualize_state(current_state,axes=cube_axes)
        plt.pause(0.1)

        while iteration <= max_iteration:
            iterations.append(iteration)
            objective_values.append(current_value)

            ax1.plot(iterations, objective_values,
color='blue')

            for ax in cube_axes:
                ax.clear()
            self.visualize_state(current_state, axes=cube_axes)
            plt.pause(0.1)

            neighbors= self.generate_neighbors(current_state)
            random_neighbor = random.choice(neighbors)

            neighbor_value =
self.calculate_objective(random_neighbor)
            if neighbor_value > current_value:
                current_state = random_neighbor
                current_value = neighbor_value

```

```

        iteration += 1

    plt.ioff()
    fig2 = plt.figure(figsize=(12, 6))
    fig2.suptitle('Initial and Final States')

    gs_init_final = fig2.add_gridspec(2, 5)

    axes_initial = [fig2.add_subplot(gs_init_final[0, i])
for i in range(5)]
    axes_final = [fig2.add_subplot(gs_init_final[1, i]) for
i in range(5)]

    self.visualize_state(initial_state, axes=axes_initial)
    self.visualize_state(current_state, axes=axes_final)

    end_time = datetime.now()
    duration = (end_time - start_time).total_seconds() *
1000

    fig2.text(0.5, 0.05, f'Final Objective Value:
{current_value} Duration: {duration} ms', ha='center',
fontsize=10)

    for ax in axes_initial:
        ax.set_title('Initial State', fontsize=8)
    for ax in axes_final:
        ax.set_title('Final State', fontsize=8)

    plt.show()
    return current_state, iteration

```

Definisi untuk tiap fungsi akan dijelaskan pada bagian dibawah sesuai dengan penggunaannya pada berbagai metode *hill climbing*. Adapun detail yang perlu dituliskan yakni berupa bahwa kelas ini memiliki konstruktor yang meng-*inherit* dari *parent class*-nya yaitu kelas *CubeSolver*.

### 2.2.1.1 Steepest Ascent Hill Climbing

```

def steepest_ascent_hill_climbing(self, title = "Figure 1"):
    current_state = self.generate_random_initial_state()
    initial_state = current_state.copy()
    current_value = self.calculate_objective(current_state)
    iteration = 0
    iterations = []

```

```

        objective_values = []

        start_time = datetime.now()

        plt.ion()
        fig = plt.figure(figsize=(10, 8))
        fig.canvas.manager.set_window_title(title)

        gs = fig.add_gridspec(nrows=2, ncols=1,
height_ratios=[1, 2])

        ax1 = fig.add_subplot(gs[0])
        ax1.set_xlabel('Iteration')
        ax1.set_ylabel('Objective Value')
        ax1.set_title('Steepest Ascent Hill Climbing Objective
Value over Iterations')

        gs2 = gs[1].subgridspec(1, 5)
        cube_axes = [fig.add_subplot(gs2[0, i]) for i in
range(5)]

        self.visualize_state(current_state, axes=cube_axes)
        plt.pause(0.1)

        while True:
            iterations.append(iteration)
            objective_values.append(current_value)

            ax1.clear()
            ax1.plot(iterations, objective_values,
color='blue')
            ax1.set_xlabel('Iteration')
            ax1.set_ylabel('Objective Value')
            ax1.set_title('Steepest Ascent Hill Climbing
Objective Value over Iterations')

            for ax in cube_axes:
                ax.clear()
            self.visualize_state(current_state, axes=cube_axes)

            plt.pause(0.1)

            neighbors = self.generate_neighbors(current_state)
            best_neighbor = None
            best_value = current_value

```

```

        for neighbor in neighbors:
            neighbor_value =
self.calculate_objective(neighbor)
            if neighbor_value > best_value:
                best_value = neighbor_value
                best_neighbor = neighbor

        if best_value <= current_value:
            plt.ioff()
            fig2 = plt.figure(figsize=(12, 6))
            fig2.suptitle('Initial and Final States')

            gs_init_final = fig2.add_gridspec(2, 5)

            axes_initial =
[fig2.add_subplot(gs_init_final[0, i]) for i in range(5)]
            axes_final = [fig2.add_subplot(gs_init_final[1,
i]) for i in range(5)]

            self.visualize_state(initial_state,
axes=axes_initial)
            self.visualize_state(current_state,
axes=axes_final)

            end_time = datetime.now()
            duration = (end_time -
start_time).total_seconds() * 1000

            fig2.text(0.5, 0.05, f'Final Objective Value:
{best_value} | Duration: {duration} ms', ha='center',
fontsize=10)

            for ax in axes_initial:
                ax.set_title('Initial State', fontsize=8)
            for ax in axes_final:
                ax.set_title('Final State', fontsize=8)

            plt.show()
            return current_state, iteration

        current_state = best_neighbor
        current_value = best_value
        iteration += 1

```

Cara kerja dari fungsi ini dapat dijelaskan dengan tahapan dibawah:

1. Fungsi melakukan inisialisasi variabel yang akan digunakan baik untuk ditampilkan maupun untuk perhitungan.
2. Fungsi kemudian menginisialisasi berbagai *canvas* seperti grafik nilai objektif terhadap iterasi dan juga penampilan *current state*.
3. Iterasi kemudian dimulai, dimana untuk setiap iterasi akan dicatat jumlah iterasinya. Untuk setiap iterasi, akan dilakukan perubahan terhadap *canvas* grafik serta *canvas current state*.
4. Pada setiap iterasi, akan dibangkitkan seluruh *neighbor* dan diambil salah satu *neighbor* dengan nilai objektif tertinggi. Nilai tersebut kemudian dibandingkan dengan nilai objektif *current state*. Program hanya akan berhenti jika nilai objektif *current*  $\geq$  nilai objektif *best-neighbor*. Jika tidak, program akan terus melanjutkan iterasinya dengan mengubah *current state* menjadi *neighbor state* yang diambil dengan seleksi nilai objektif.
5. Jika iterasi selesai, akan ditampilkan *initial state*, *final state*, durasi, jumlah iterasi, dan waktu eksekusi program.

### 2.2.1.2 Hill Climbing With Sideways Move

```
def sideways_move_hill_climbing(self, title = "Sideways Move Hill Climbing"):  
    current_state = self.generate_random_initial_state()  
    current_value = self.calculate_objective(current_state)  
    initial_state = current_state.copy()  
    iteration = 0  
    iterations = []  
    objective_values = []  
  
    start_time = datetime.now()  
  
    plt.ion()  
    fig = plt.figure(figsize=(10, 8))  
    fig.canvas.manager.set_window_title(title)  
  
    gs = fig.add_gridspec(nrows=2, ncols=1,  
height_ratios=[1, 2])  
  
    ax1 = fig.add_subplot(gs[0])  
    ax1.set_xlabel('Iteration')
```

```

        ax1.set_ylabel('Objective Value')
        ax1.set_title('Sideways Move Hill Climbing Objective
Value over Iterations')

        gs2 = gs[1].subgridspec(1, 5)
        cube_axes = [fig.add_subplot(gs2[0, i]) for i in
range(5)]

        self.visualize_state(current_state, axes=cube_axes)
        plt.pause(0.1)

        while True:
            iteration += 1
            iterations.append(iteration)
            objective_values.append(current_value)

            ax1.clear()
            ax1.plot(iterations, objective_values,
color='blue')
            ax1.set_xlabel('Iteration')
            ax1.set_ylabel('Objective Value')
            ax1.set_title('Sideways Move Hill Climbing
Objective Value over Iterations')

            for ax in cube_axes:
                ax.clear()
            self.visualize_state(current_state, axes=cube_axes)

            plt.pause(0.1)

            neighbors = self.generate_neighbors(current_state)
            best_neighbor = None
            best_value = float('-inf')

            for neighbor in neighbors:
                neighbor_value =
self.calculate_objective(neighbor)
                if neighbor_value > best_value:
                    best_value = neighbor_value
                    best_neighbor = neighbor

            if best_value < current_value:
                plt.ioff()
                fig2 = plt.figure(figsize=(12, 6))
                fig2.suptitle('Initial and Final States')

```

```

        gs_init_final = fig2.add_gridspec(2, 5)

        axes_initial =
[fig2.add_subplot(gs_init_final[0, i]) for i in range(5)]
        axes_final = [fig2.add_subplot(gs_init_final[1,
i]) for i in range(5)]

        self.visualize_state(initial_state,
axes=axes_initial)
        self.visualize_state(current_state,
axes=axes_final)

        end_time = datetime.now()
        duration = (end_time -
start_time).total_seconds() * 1000

        fig2.text(0.5, 0.05, f'Final Objective Value:
{best_value} | Duration: {duration} ms | Iteration:
{iteration}', ha='center', fontsize=10)

        for ax in axes_initial:
            ax.set_title('Initial State', fontsize=8)
        for ax in axes_final:
            ax.set_title('Final State', fontsize=8)

        plt.show()
        return current_state, iteration

    current_state = best_neighbor
    current_value = best_value

```

Cara kerja dari fungsi ini dapat dijelaskan dengan tahapan dibawah:

1. Fungsi melakukan inisialisasi variabel yang akan digunakan baik untuk ditampilkan maupun untuk perhitungan.
2. Fungsi kemudian menginisialisasi berbagai *canvas* seperti grafik nilai objektif terhadap iterasi dan juga penampilan *current state*.
3. Iterasi kemudian dimulai, dimana untuk setiap iterasi akan dicatat jumlah iterasinya. Untuk setiap iterasi, akan dilakukan perubahan terhadap *canvas* grafik serta *canvas current state*.

4. Pada setiap iterasi, akan dibangkitkan seluruh *neighbor* dan diambil salah satu *neighbor* dengan nilai objektif tertinggi. Nilai tersebut kemudian dibandingkan dengan nilai objektif *current state*. Program hanya akan berhenti jika nilai objektif *current* > nilai objektif *best-neighbor*. Jika tidak, program akan terus melanjutkan iterasinya dengan mengubah *current state* menjadi *neighbor state* yang diambil dengan seleksi nilai objektif. Atas hal ini, program akan sering mengalami gerakan horizontal akibat persyaratan yang lebih sulit untuk berhenti ketika dibandingkan dengan algoritma *steepest ascent hill climbing*.
5. Jika iterasi selesai, akan ditampilkan *initial state*, *final state*, durasi, jumlah iterasi, dan waktu eksekusi program.

### 2.2.1.3 Random Restart Hill Climbing

```
def random_restart_hill_climbing(self, max_restart):
    all_state = []
    for i in range(max_restart):
        title = f"Restart {i + 1}"
        final_state, iteration =
self.steepest_ascent_hill_climbing(title)
        print("Num of iteration for restart ", i+1, ": ",
iteration)
        all_state.append([final_state, iteration])
    return all_state[-1][0], all_state[-1][1]
```

Fungsi ini memakai fungsi dari *steepest\_ascent\_hill\_climbing*. Secara detail, fungsi ini menerima parameter jumlah perulangan maksimal, yang akan digunakan sebagai basis jumlah iterasi. Fungsi ini akan menjalankan algoritma *steepest\_ascent\_hill\_climbing* sebanyak jumlah iterasi maksimum dan mengembalikan *state* akhir setelah melewati jumlah iterasi maksimal tersebut.

### 2.2.1.4 Stochastic Hill Climbing

```
def stochastic_hill_climbing(self, max_iteration = 100):
    current_state = self.generate_random_initial_state()
    initial_state = current_state.copy()
    current_value = self.calculate_objective(current_state)
```



```

        iteration = 1
        iterations = []
        objective_values = []

        start_time = datetime.now()

        plt.ion()
        fig= plt.figure(figsize=(10, 8))
        gs = fig.add_gridspec(nrows=2, ncols=1,
height_ratios=[1, 2])

        ax1 = fig.add_subplot(gs[0])
        ax1.set_xlabel('Iteration')
        ax1.set_ylabel('Objective Value')
        ax1.set_title('Stochastic Hill Climbing Objective Value
over Iterations')

        gs2 = gs[1].subgridspec(1, 5)
        cube_axes = [fig.add_subplot(gs2[0, i]) for i in
range(5)]

        self.visualize_state(current_state,axes=cube_axes)
        plt.pause(0.1)

        while iteration < max_iteration:
            iterations.append(iteration)
            objective_values.append(current_value)

            ax1.plot(iterations, objective_values,
color='blue')

            for ax in cube_axes:
                ax.clear()
                self.visualize_state(current_state, axes=cube_axes)
                plt.pause(0.1)

            neighbors= self.generate_neighbors(current_state)
            random_neighbor = random.choice(neighbors)

            neighbor_value =
self.calculate_objective(random_neighbor)
            if neighbor_value > current_value:
                current_state = random_neighbor
                current_value = neighbor_value
            iteration += 1
        plt.ioff()

```

```

fig2 = plt.figure(figsize=(12, 6))
fig2.suptitle('Initial and Final States')

gs_init_final = fig2.add_gridspec(2, 5)

axes_initial = [fig2.add_subplot(gs_init_final[0, i])
for i in range(5)]
axes_final = [fig2.add_subplot(gs_init_final[1, i]) for
i in range(5)]

self.visualize_state(initial_state, axes=axes_initial)
self.visualize_state(current_state, axes=axes_final)

end_time = datetime.now()
duration = (end_time - start_time).total_seconds() *
1000

fig2.text(0.5, 0.05, f'Final Objective Value:
{current_value} | Duration: {duration} ms | Iteration:
{iteration}', ha='center', fontsize=10)

for ax in axes_initial:
    ax.set_title('Initial State', fontsize=8)
for ax in axes_final:
    ax.set_title('Final State', fontsize=8)

plt.show()
return current_state, iteration

```

Cara kerja dari fungsi ini dapat dijelaskan dengan tahapan dibawah:

1. Fungsi melakukan inisialisasi variabel yang akan digunakan baik untuk ditampilkan maupun untuk perhitungan. Fungsi ini menerima jumlah iterasi maksimal sebagai parameternya.
2. Fungsi kemudian menginisialisasi berbagai *canvas* seperti grafik nilai objektif terhadap iterasi dan juga penampilan *current state*.
3. Iterasi kemudian dimulai, dimana untuk setiap iterasi akan dicatat jumlah iterasinya. Untuk setiap iterasi, akan dilakukan perubahan terhadap *canvas* grafik serta *canvas current state*.
4. Pada setiap iterasi, akan dibangkitkan seluruh *neighbor* dan diambil salah satu *neighbor* secara acak. Nilai dari *neighbor* tersebut kemudian dibandingkan dengan nilai objektif

*current state*. Jika nilai objektif *neighbor state* > nilai objektif *current state*, maka *neighbor state* menjadi *current state*. Iterasi akan berhenti jika telah mencapai jumlah iterasi maksimal yang diberikan pada parameter.

5. Jika iterasi selesai, akan ditampilkan *initial state*, *final state*, durasi, jumlah iterasi, dan waktu eksekusi program.

### 2.2.2 Simulated Annealing

```
from datetime import datetime
import random
import math
import matplotlib.pyplot as plt
from base_solver import CubeSolver

class SimulatedAnnealingSolver(CubeSolver):
    def __init__(self, state=None):
        super().__init__(state)

    def generate_random_neighbor(self, state):
        neighbor = state.copy()
        i, j = random.sample(range(len(state)), 2)
        neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
        return neighbor

    def simulated_annealing(self, initial_temp=1000,
cooling_rate=0.99, min_temp=0.01):
        current_state = self.state
        initial_state = current_state.copy()
        current_value = self.calculate_objective(current_state)
        best_state = current_state
        best_value = current_value
        temperature = initial_temp
        iteration = 0
        iterations = []
        objective_values = []
        stuck_count = 0

        start_time = datetime.now()

        plt.ion()
        fig = plt.figure(figsize=(10, 8))
        gs = fig.add_gridspec(nrows=2, ncols=1,
height_ratios=[1, 2])
```

```

        ax = fig.add_subplot(gs[0])
        ax.set_xlabel('Iteration')
        ax.set_ylabel('Objective Value')
        ax.set_title('Simulated Annealing Objective Value over
Iterations')

        gs2 = gs[1].subgridspec(1, 5)
        cube_axes = [fig.add_subplot(gs2[0, i]) for i in
range(5)]

        self.visualize_state(current_state, axes=cube_axes)
        plt.pause(0.1)

        while temperature > min_temp:
            iterations.append(iteration)
            objective_values.append(current_value)

            ax.plot(iterations, objective_values, color='red')
            for ax1 in cube_axes:
                ax1.clear()
            self.visualize_state(current_state, axes=cube_axes)
            plt.pause(0.1)

            neighbor =
self.generate_random_neighbor(current_state)
            neighbor_value = self.calculate_objective(neighbor)

            delta_e = neighbor_value - current_value

            if delta_e >= 0 or math.exp(delta_e / temperature)
> random.random():
                current_state = neighbor
                current_value = neighbor_value

                if current_value > best_value:
                    best_state = current_state
                    best_value = current_value
            else:
                stuck_count += 1

            temperature *= cooling_rate
            iteration += 1

        plt.ioff()
        fig2 = plt.figure(figsize=(12, 6))

```

```

        fig2.suptitle('Initial and Final States')

        gs_init_final = fig2.add_gridspec(2, 5)

        axes_initial = [fig2.add_subplot(gs_init_final[0, i])
for i in range(5)]
        axes_final = [fig2.add_subplot(gs_init_final[1, i]) for
i in range(5)]

        self.visualize_state(initial_state, axes=axes_initial)
        self.visualize_state(current_state, axes=axes_final)

        end_time = datetime.now()
        duration = (end_time - start_time).total_seconds() *
1000

        fig2.text(0.5, 0.05, f'Final Objective Value:
{current_value} Duration: {duration} ms', ha='center',
fontsize=10)

        for ax in axes_initial:
            ax.set_title('Initial State', fontsize=8)
        for ax in axes_final:
            ax.set_title('Final State', fontsize=8)

        plt.show()
        print("Stuck Frequency:", stuck_count)
        return best_state

```

Kelas ini mendefinisikan salah satu algoritma penyelesaian dengan *local search* yaitu *Simulated Annealing*. Adapun fungsi yang dideklarasikan dalam kelas ini yaitu:

1. Konstruktor dari kelas ini cukup hanya berupa inisialisasi dari *class parent* yaitu *CubeSolver*.
2. Fungsi *generate\_random\_neighbor* yang memilih sebuah *neighbor* secara acak dari *state* masukan. Cara kerja dari fungsi ini yaitu dengan sembarang memilih 2 angka yang merepresentasikan index pada *array cube*. Kedua posisi tersebut kemudian di-*swap* dan dikembalikan sebagai *neighbor*.
3. Fungsi *simulated\_annealing* merupakan fungsi utama untuk menyelesaikan permasalahan *magic cube*. Dalam *simulated\_annealing*, terdapat beberapa variabel yang juga dideklarasikan sebagai parameter dalam fungsi ini, yaitu suhu awal , laju

pendinginan, serta suhu minimal. Iterasi akan dilakukan sampai suhu sewaktu iterasi lebih kecil dari suhu minimal, dimulai dari suhu awal. Untuk setiap iterasi, suhu akan menurun sebesar suhu-state \* laju pendinginan. Fungsi dimulai dengan deklarasi berbagai variabel yang akan dijadikan sebagai informasi sebagai bagian dari representasi *final result*. Beberapa variabel ini seperti *state awal*, nilai *state awal*, iterasi ( iterasi akan berubah sesuai dengan perubahan suhu terhadap laju pendinginan ) , dan *stuck\_count* sebagai penanda dalam perhitungan dengan algoritma ini. Setelah deklarasi berbagai variabel yang diperlukan, dilakukan inisialisasi untuk berbagai jenis *plotting*. Setelah itu, tahapan iterasi dimulai.

4. Untuk setiap iterasi, akan dihasilkan satu *neighbor* dari state saat ini. Kemudian, dilakukan pencarian nilai *delta* dengan menghitung selisih antara nilai objektif untuk *current state* dan *neighbor state*. Jika nilai *delta* lebih besar dari 0, maka *neighbor state* akan menjadi *neighbor state*. Jika tidak, *current state* tetap akan bisa menjadi *current state* dengan probabilitas sebesar  $e^{\frac{\Delta E}{T}}$ , dengan T merupakan suhu saat perhitungan. Jika masih gagal mengambil *neighbor state* sebagai *current state*, maka kondisi ini akan terhitung sebagai *stuck condition*. Ketika iterasi selesai, akan dilakukan penampilan *final state*, *final objective value*, *duration*, dan berbagai variabel lainnya.

### 2.2.3 Genetic Algorithm

```
import random
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
from base_solver import CubeSolver

class GeneticAlgorithmSolver(CubeSolver):
    def __init__(self, population_size=50, num_iterations=100):
        super().__init__()
        self.population_size = population_size
        self.num_iterations = num_iterations
        self.population = []
        self.generate_population(population_size)

    def generate_population(self, pop_size):
        return [self.generate_random_initial_state() for _ in range(pop_size)]
```

```

def selection(self, population):
    fitness_values = [-self.calculate_objective(ind) for
ind in population]
    total_fitness = sum(fitness_values)
    if total_fitness == 0:
        selection_probs = [1 / len(population)] *
len(population)
    else:
        selection_probs = [fitness / total_fitness for
fitness in fitness_values]
    chosen_index = random.choices(
        population=range(len(population)),
        weights=selection_probs,
        k=1
    )[0]
    return population[chosen_index]

def crossover(self, parent1, parent2):
    size = len(parent1)
    cxpoint1 = random.randint(0, size - 2)
    cxpoint2 = random.randint(cxpoint1 + 1, size - 1)
    child1 = [None] * size
    child2 = [None] * size
    child1[cxpoint1:cxpoint2] = parent1[cxpoint1:cxpoint2]
    current_pos = cxpoint2 % size
    parent2_pos = cxpoint2 % size
    while None in child1:
        if parent2[parent2_pos % size] not in child1:
            child1[current_pos % size] =
parent2[parent2_pos % size]
            current_pos += 1
            parent2_pos += 1
    child2[cxpoint1:cxpoint2] = parent2[cxpoint1:cxpoint2]
    current_pos = cxpoint2 % size
    parent1_pos = cxpoint2 % size
    while None in child2:
        if parent1[parent1_pos % size] not in child2:
            child2[current_pos % size] =
parent1[parent1_pos % size]
            current_pos += 1
            parent1_pos += 1
    return child1, child2

```

```

def mutate(self, individual, mutation_rate=0.5):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            swap_index = random.randint(0, len(individual)
- 1)
            individual[i], individual[swap_index] =
individual[swap_index], individual[i]
    return individual

def run(self):
    best_solution = None
    best_fitness = float('-inf')
    avg_fitness_per_iteration = []
    max_fitness_per_iteration = []
    fitness_values_per_iteration = []

    for state in self.population:
        initial_value = self.calculate_objective(state)
        if initial_value > best_fitness:
            best_fitness = initial_value
            best_initial_state = state

    start_time = datetime.now()

    plt.ion()

    fig = plt.figure(figsize=(14, 12))
    gs = fig.add_gridspec(nrows=3, ncols=1,
height_ratios=[1, 1, 2])

    ax1 = fig.add_subplot(gs[0])
    ax1.set_xlabel('Iteration')
    ax1.set_ylabel('Objective Value')
    ax1.set_title('Genetic Algorithm Performance (Average
and Max Fitness)')

    ax2 = fig.add_subplot(gs[1])
    fig.subplots_adjust(hspace=0.5)
    ax2.set_xlabel('Iteration')
    ax2.set_ylabel('Fitness Value')
    ax2.set_title('Fitness Value of Each Individual per
Iteration')
    gs2 = gs[2].subgridspec(1, 5)
    cube_axes = [fig.add_subplot(gs2[0, i]) for i in
range(5)]
    self.visualize_state(best_initial_state,

```



```

axes=cube_axes)

    for iteration in range(self.num_iterations):
        fitness_values = [self.calculate_objective(ind) for
ind in self.population]
        fitness_values_per_iteration.append(fitness_values)
        max_fitness = max(fitness_values)
        avg_fitness = sum(fitness_values) /
len(fitness_values)

        avg_fitness_per_iteration.append(avg_fitness)
        max_fitness_per_iteration.append(max_fitness)

        if max_fitness > best_fitness:
            best_fitness = max_fitness
            best_solution =
self.population[fitness_values.index(max_fitness)]
            ax1.clear()
            ax1.plot(range(len(avg_fitness_per_iteration)),
avg_fitness_per_iteration, label='Average Fitness',
color='blue')
            ax1.plot(range(len(max_fitness_per_iteration)),
max_fitness_per_iteration, label='Max Fitness', color='green')
            ax1.set_xlabel('Iteration')
            ax1.set_ylabel('Fitness Value')
            ax1.set_title('Genetic Algorithm Performance')
            ax1.legend()
            ax2.clear()
            for i, fitness_vals in
enumerate(fitness_values_per_iteration):
                ax2.scatter([i] * len(fitness_vals),
fitness_vals, color='blue', alpha=0.3, s=10)
            ax2.set_xlabel('Iteration')
            ax2.set_ylabel('Fitness Value')
            ax2.set_title('Fitness Value of Each Individual per
Iteration')

        # for ax in cube_axes:
        #     ax.clear()
        # self.visualize_state(best_solution,
axes=cube_axes)
        plt.pause(0.1)

        new_population = []
        while len(new_population) < self.population_size:
            parent1 = self.selection(self.population)
            parent2 = self.selection(self.population)

```

```

        child1, child2 = self.crossover(parent1,
parent2)

        temp = self.mutate(child1)
        new_population.append(temp)
        if len(new_population) < self.population_size:
            temp = self.mutate(child2)
            new_population.append(temp)
        self.population = new_population
        print(f"Iteration {iteration +
1}/{self.num_iterations} - Best Fitness: {-best_fitness}")
        print("ukuran",self.population_size)
    plt.ioff()
    fig2 = plt.figure(figsize=(12, 6))
    fig2.suptitle('Initial and Final States')

    gs_init_final = fig2.add_gridspec(2, 5)

    axes_initial = [fig2.add_subplot(gs_init_final[0, i])
for i in range(5)]
    axes_final = [fig2.add_subplot(gs_init_final[1, i]) for
i in range(5)]

    best_final_value = float('-inf')
    for state in self.population:
        final_value = self.calculate_objective(state)
        if final_value > best_final_value:
            best_final_value = final_value
            best_final_state = state
    self.visualize_state(best_initial_state,
axes=axes_initial)
    self.visualize_state(best_final_state, axes=axes_final)

    end_time = datetime.now()
    duration = (end_time - start_time).total_seconds() *
1000

    fig2.text(0.5, 0.05, f'Final Objective Value:
{best_final_value} Duration: {duration} ms Population Size:
{self.population_size}', ha='center', fontsize=10)

    for ax in axes_initial:
        ax.set_title('Initial State', fontsize=8)
    for ax in axes_final:
        ax.set_title('Final State', fontsize=8)

    plt.show()

```

```
return best_solution
```

Kelas ini menjalankan algoritma *Genetic Algorithm* dengan berbagai fungsi yaitu:

1. Konstruktor yang menginisialisasi jumlah populasi , jumlah iterasi serta memenuhi populasi dengan fungsi *generate\_population*. Kelas ini juga merupakan *children* dari kelas *CubeSolver*
2. Fungsi *generate\_population* menghasilkan *initial state* acak dengan memanfaatkan fungsi pada *CubeSolver* yaitu *generate\_random\_initial\_state*.
3. Fungsi *selection* memilih *state* sesuai dengan proporsi masing-masing *state* pada populasi dengan mengandalkan *weight* dari *fitness function*. *Fitness* dalam konteks persoalan ini mengacu pada nilai objektif untuk tiap *state* pada populasi dan bobot dipertimbangkan dengan membagi *fitness value* dengan *total fitness*.
4. Fungsi *crossover* menerima 2 *parent* dan melakukan *crossover* pada keduanya. Fungsi dimulai dengan mengambil secara acak 2 titik , satu titik penanda awal dan satu titik penanda akhir. Nilai yang akan di-*crossover* yaitu nilai yang berada di kedua titik. *Child* 1 akan memiliki nilai dari *parent* 1 untuk posisi penanda awal sampai akhir. Sisanya akan diambil dari *parent* 2. Hal ini juga berlaku secara berbalik untuk *Child* 2.
5. Fungsi *mutate* menerima individual yang akan dimutasi beserta dengan kemungkinan terjadinya mutasi (yakni 0.5). Fungsi ini mengiterasi untuk setiap anggota pada individual dan menghitung peluang mutasi secara acak. Jika hasil acakan > persentase mutasi, akan terjadi pertukaran antara nilai pada posisi individual tersebut dengan salah satu index yang juga dipilih secara acak.
6. Fungsi *run* merupakan fungsi utama dalam penggunaan algoritma *genetic algorithm*.

Adapun tahapan dari fungsi ini yaitu:

- a. Fungsi melakukan inisialisasi variabel yang akan digunakan baik untuk ditampilkan maupun untuk perhitungan. Inisialisasi yang dilakukan mencari *best\_state* berdasarkan *fitness value* dari populasi awal.
- b. Fungsi kemudian menginisialisasi berbagai canvas seperti grafik nilai objektif terhadap iterasi dan juga penampilan current state.

- c. Setelah itu, dilakukan iterasi sebanyak jumlah iterasi yang sudah dideklarasikan pada konstruktor.
- d. Iterasi dimulai dengan menghitung berbagai jenis variabel seperti nilai *fitness* maksimal serta nilai rata-rata *fitness* yang nantinya akan digunakan dalam visualisasi.
- e. Untuk setiap iterasi, akan mencoba untuk memenuhi populasi baru yang awalnya kosong dengan menerapkan algoritma *genetic algorithm* pada setiap populasi. Tahapan yang dilakukan yaitu dengan memilih 2 *parent* dengan fungsi *selection* , menghasilkan *child 1* dan 2 dari fungsi *crossover* dan memanggil fungsi *mutation* untuk kedua *child* selama jumlah populasi belum melewati jumlah yang dideklarasikan sebelumnya.
- f. Untuk setiap iterasi, akan memiliki populasi dari iterasi sebelumnya. Perulangan dilakukan sebanyak jumlah iterasi yang sudah dideklarasikan, dan setelah selesai, akan dicari *best state* dengan mencari dari populasi akhir *state* yang memiliki *fitness value* paling baik.

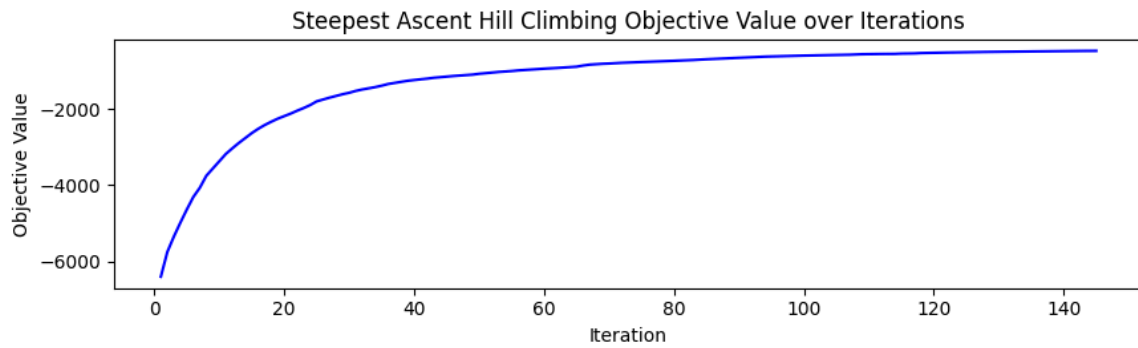
## 2.3. Hasil eksperimen dan analisis

### 2.3.1 Hill Climbing

#### 2.3.1.1 Steepest Ascent Hill Climbing

##### Test Case 1

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



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

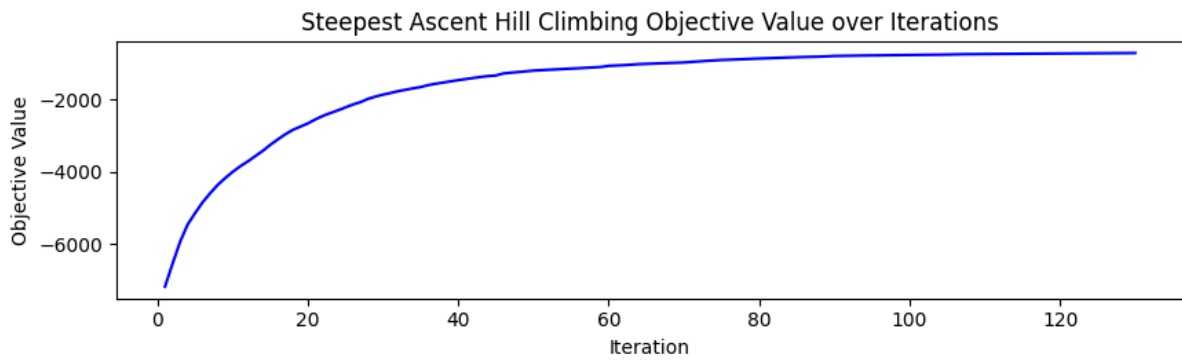
Final Objective Value: -472 | Duration: 88119.274 ms | Iteration: 145

**Gambar 2.3.1.1.1.** Hasil dari Test Case 1 untuk Algoritma Steepest Ascent Hill Climbing

Dari eksperimen ini , diketahui bahwa dari *initial state* persoalan diatas, diperlukan iterasi sebanyak 145 kali untuk mencapai maksimum (bisa lokal maupun global , namun dalam kondisi ini lokal). Nilai objektif akhir yang didapat yakni adalah -472 , dan memiliki durasi selama kurang lebih 88 detik.

## Test Case 2

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



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

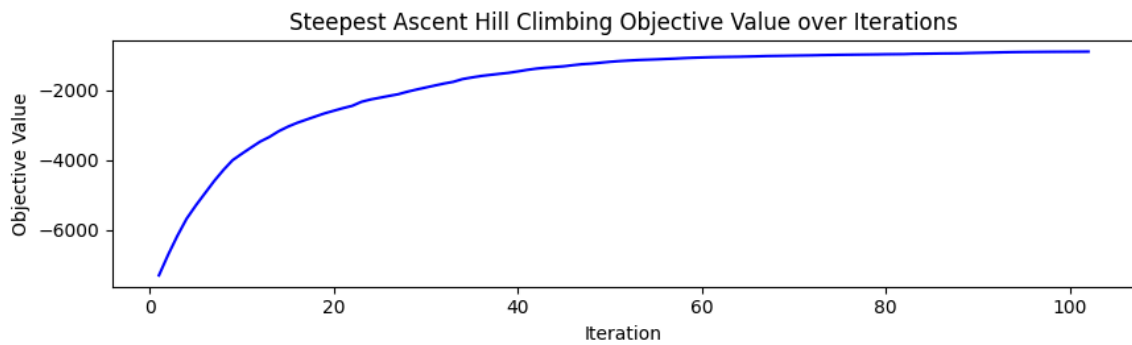
Final Objective Value: -704 | Duration: 79735.305 ms | Iteration: 130

### Gambar 2.3.1.1.2. Hasil dari Test Case 2 untuk Algoritma Steepest Ascent Hill Climbing

Dari eksperimen ini , diketahui bahwa dari *initial state* persoalan diatas, diperlukan iterasi sebanyak 130 kali untuk mencapai maksimum (bisa lokal maupun global , namun dalam kondisi ini lokal). Nilai objektif akhir yang didapat yakni adalah -704 , dan memiliki durasi selama kurang lebih 79 detik.

### Test Case 3

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



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

Final Objective Value: -900 | Duration: 61460.528 ms | Iteration: 102

**Gambar 2.3.1.1.3.** Hasil dari Test Case 3 untuk Algoritma Steepest Ascent Hill Climbing

Dari eksperimen ini , diketahui bahwa dari *initial state* persoalan diatas, diperlukan iterasi sebanyak 102 kali untuk mencapai maksimum (bisa lokal maupun global , namun dalam kondisi ini lokal). Nilai objektif akhir yang didapat yakni adalah -900 , dan memiliki durasi selama kurang lebih 61 detik.

#### 2.3.1.1.1 Hasil Analisis Steepest Ascent Hill Climbing

Algoritma ini bergantung pada jumlah iterasi dalam mencari hasil yang paling optimal. Dibandingkan dengan algoritma lain pada laporan ini, algoritma *steepest ascent hill climbing* memiliki cara kerja yang lebih terus terang. Algoritma ini akan terus membandingkan dirinya dengan *neighbor* terbaik dan hanya akan berhenti jika dirinya lebih baik atau sama dengan *neighbor*-nya. Oleh karena itu, memiliki perubahan acak antar jumlah iterasi yang diperlukan untuk mencapai nilai objektif terbaik menurut algoritma ini (*local maximum*), karena untuk setiap percobaan tentu memiliki *initial state* dan *neighbor* yang berbeda pula.

Algoritma ini memerlukan sumber daya yang lebih besar, yakni karena diperlukan pencarian *neighbor* terbaik sehingga memakan lebih banyak waktu , tergantung jumlah iterasi

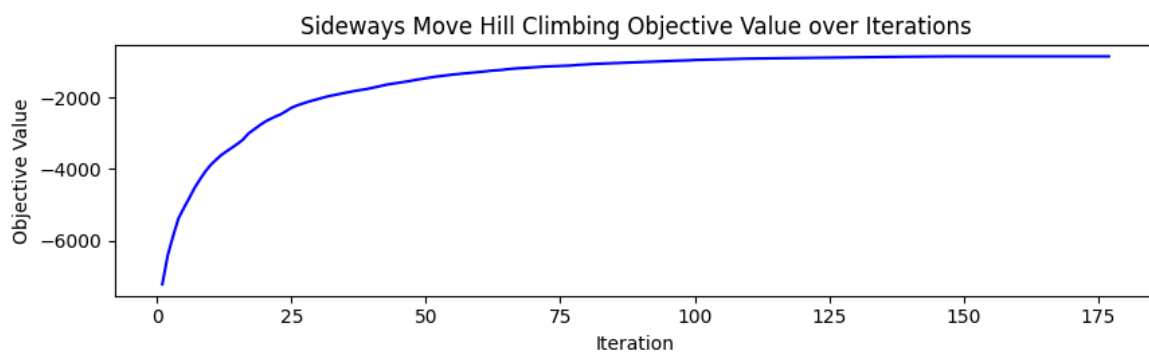
yang diperlukan. Algoritma ini memberikan hasil yang dapat terbilang cukup memuaskan , terlihat dari 3 percobaan yang dilakukan diatas umumnya memiliki hasil objektif lebih dari -1000.

### 2.3.1.2 Hill Climbing With Sideways Move

#### Test Case 1 ( max-sideways = 20)

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

Final Objective Value: -856 | Duration: 110338.59599999999 ms | Iteration: 177



**Gambar 2.3.1.2.1.** Hasil dari Test Case 1 untuk Metode Hill Climbing With Sideways Move

Dari eksperimen ini , diketahui bahwa dari *initial state* persoalan diatas, diperlukan iterasi sebanyak 177 kali untuk mencapai maksimum (bisa lokal maupun global , namun dalam kondisi

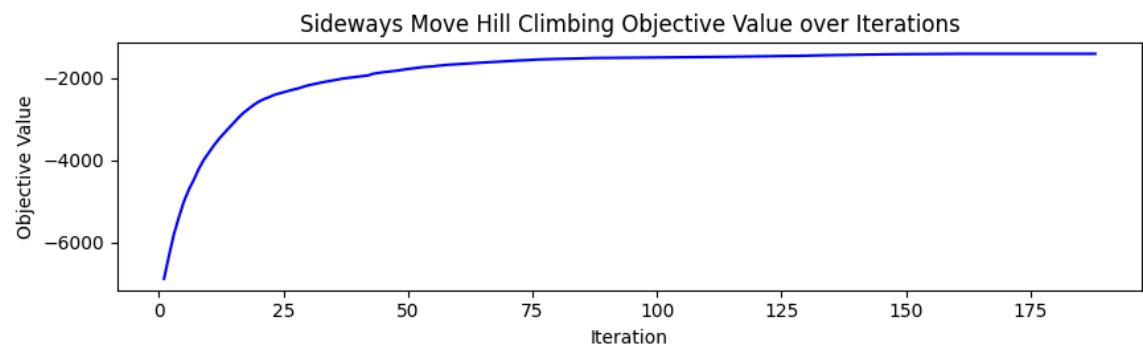


ini lokal) , terhitung pula sideways move yang mungkin terjadi secara berulang. Nilai objektif akhir yang didapat yakni adalah -856 , dan memiliki durasi selama kurang lebih 110 detik.

Test Case 2 ( max-sideways = 20)

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

Final Objective Value: -1407 | Duration: 116868.4 ms | Iteration: 188



Gambar 2.3.1.2.2. Hasil dari Test Case 2 untuk Metode Hill Climbing With Sideways Move

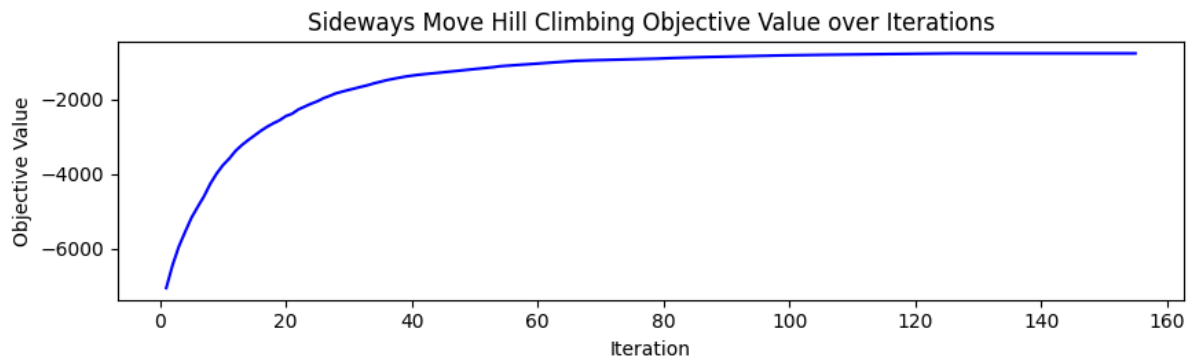
Dari eksperimen ini , diketahui bahwa dari *initial state* persoalan diatas, diperlukan iterasi sebanyak 188 kali untuk mencapai maksimum (bisa lokal maupun global , namun dalam kondisi ini lokal) , terhitung pula sideways move yang mungkin terjadi secara berulang. Nilai objektif akhir yang didapat yakni adalah -1407 , dan memiliki durasi selama kurang lebih 116 detik.

### Test Case 3 ( max-sideways = 20)

Initial and Final States

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

Final Objective Value: -768 | Duration: 97406.673 ms | Iteration: 155



**Gambar 2.3.1.2.3.** Hasil dari Test Case 3 untuk Metode Hill Climbing With Sideways Move

Dari eksperimen ini , diketahui bahwa dari *initial state* persoalan diatas, diperlukan iterasi sebanyak 155 kali untuk mencapai maksimum (bisa lokal maupun global , namun dalam kondisi ini lokal) , terhitung pula sideways move yang mungkin terjadi secara berulang. Nilai objektif akhir yang didapat yakni adalah -768 , dan memiliki durasi selama kurang lebih 97 detik.

### 2.3.1.2.1 Hasil Analisis Hill Climbing With Sideways Move

Algoritma ini juga bergantung pada jumlah iterasi dalam mencari hasil yang paling optimal. Dibandingkan dengan algoritma lain pada laporan ini, algoritma *hill climbing with sideways move* memiliki cara kerja yang lebih terus terang namun selektif dalam memilih solusi. Algoritma ini akan terus membandingkan dirinya dengan *neighbor* terbaik dan hanya akan berhenti jika dirinya lebih baik dari *neighbor*-nya. Oleh karena itu, memiliki perubahan acak antar jumlah iterasi yang diperlukan untuk mencapai nilai objektif terbaik menurut algoritma ini (*local maximum*), karena untuk setiap percobaan tentu memiliki *initial state* dan *neighbor* yang berbeda pula.

Algoritma ini memerlukan sumber daya yang lebih besar lagi, terutama jika dibandingkan dengan *steepest ascent hill climbing*, yakni karena diperlukan pencarian *neighbor* terbaik sehingga memakan lebih banyak waktu dan juga seleksi yang hanya terjadi jika *neighbor* lebih baik, tidak hanya sama, tergantung jumlah iterasi yang diperlukan.

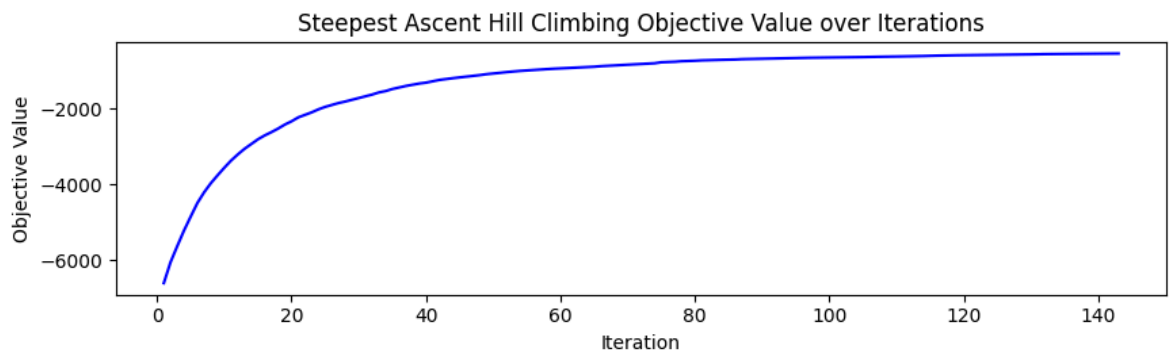
Algoritma ini memberikan hasil yang dapat dibilang cukup memuaskan, terlihat dari 3 percobaan yang dilakukan diatas umumnya memiliki hasil objektif lebih dari -1000. Namun algoritma ini memakan terlalu banyak waktu karena seleksi yang memaksa untuk mencari hasil paling optimal, sehingga harus menggunakan parameter iterasi maksimal untuk memastikan tidak terjadi *sideways* terlalu banyak.

### 2.3.1.3 Random Restart Hill Climbing

#### Test Case 1 ( Max Restart = 3 )

#### Restart 1

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



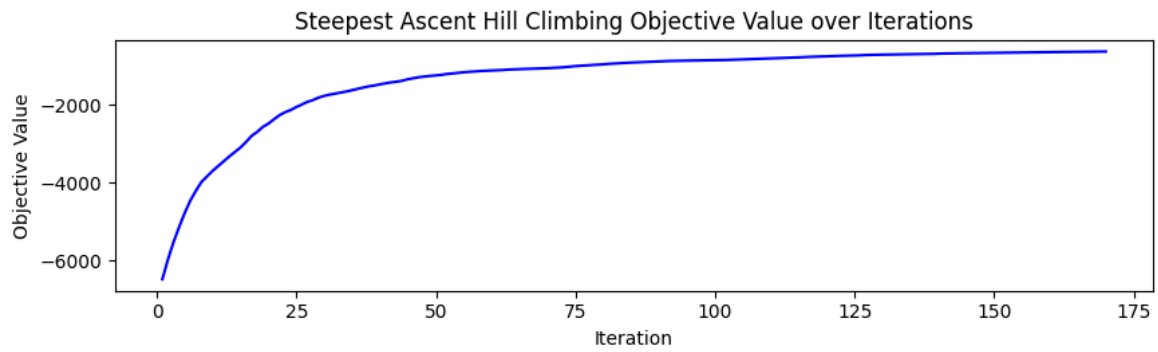
Final State					Final State					Final State					Final State					Final State				
33	66	77	64	75	41	57	113	54	49	45	67	61	109	28	99	25	40	80	72	97	100	24	8	86
116	121	1	89	19	56	27	38	91	103	73	50	120	13	58	39	53	108	12	102	31	47	95	110	32
6	104	92	29	84	118	115	44	35	4	55	63	81	90	26	21	14	96	43	125	114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

Final Objective Value: -538 | Duration: 130934.07199999999 ms | Iteration: 143

**Gambar 2.3.1.3.1.** Hasil dari Restart 1 pada Test Case 1 untuk Algoritma Random Restart Hill Climbing

## Restart 2

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



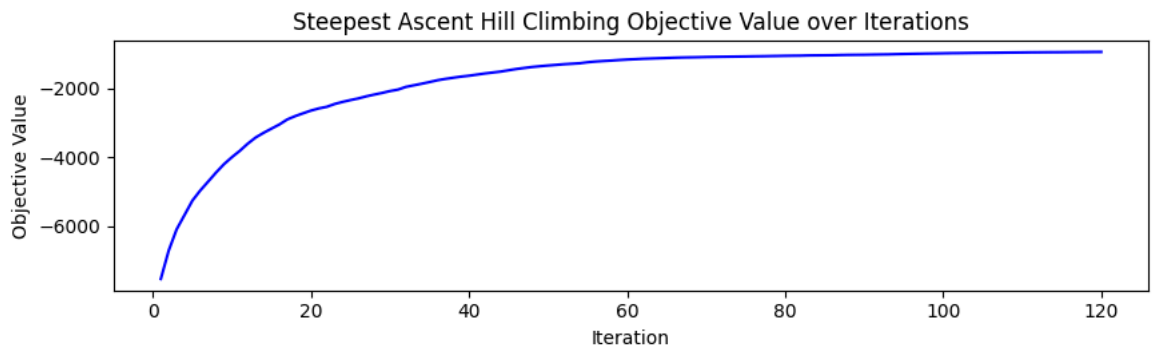
Final State					Final State					Final State					Final State					Final State				
50	106	30	33	98	123	37	43	108	4	96	85	81	44	9	18	13	116	53	119	27	75	45	77	90
115	20	124	14	42	5	73	63	59	117	1	67	26	120	103	113	95	62	40	6	93	60	32	83	47
97	66	57	78	17	34	38	48	102	94	109	28	71	56	51	72	122	22	24	65	3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

Final Objective Value: -624 | Duration: 152920.17200000002 ms | Iteration: 170

**Gambar 2.3.1.3.2.** Hasil dari Restart 2 pada Test Case 1 untuk Algoritma Random Restart Hill Climbing

### Restart 3

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



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

Final Objective Value: -945 | Duration: 99802.965 ms | Iteration: 120

**Gambar 2.3.1.3.3.** Hasil dari Restart 3 pada Test Case 1 untuk Algoritma Random Restart Hill Climbing

**Hasil Akhir :**

```
Final state: [33, 66, 77, 64, 75, 116, 121, 1, 89, 19, 6, 104,
92, 29, 84, 107, 7, 87, 15, 98, 52, 17, 93, 117, 36, 41, 57, 11
3, 54, 49, 56, 27, 38, 91, 103, 118, 115, 44, 35, 4, 70, 83, 9,
88, 65, 30, 34, 111, 46, 94, 45, 67, 61, 109, 28, 73, 50, 120,
13, 58, 55, 63, 81, 90, 26, 22, 74, 37, 60, 122, 119, 59, 16,
42, 79, 99, 25, 40, 80, 72, 39, 53, 108, 12, 102, 21, 14, 96, 4
3, 125, 105, 82, 48, 68, 10, 51, 124, 23, 112, 5, 97, 100, 24,
8, 86, 31, 47, 95, 110, 32, 114, 18, 2, 106, 76, 11, 69, 123, 8
5, 20, 62, 78, 71, 3, 101]
Iteration: 143
Objective value: -538
```

**Gambar 2.3.1.3.4.** Hasil Akhir pada Test Case 1 untuk Algoritma Random Restart Hill Climbing

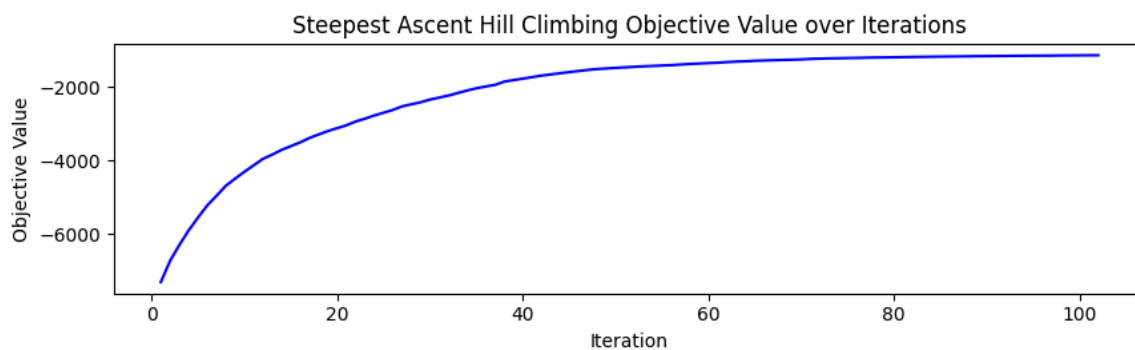
Di eksperimen ini, dilakukan 3 kali pengulangan terhadap pencarian solusi menggunakan algoritma Steepest Ascent Hill Climbing. Dari ketiga pengulangan tersebut, akan dicatat jumlah

iterasi dan nilai objektifnya , lalu akan dipilih nilai objektif yang paling mendekati 0. Pada eksperimen ini dipilih state akhir pada pengulangan ke-1, diperlukan iterasi sebanyak 143 kali untuk mencapai maksimum (bisa lokal maupun global, namun dalam kondisi ini lokal), dan nilai objektif yang didapat adalah -538 .

## Test Case 2 ( Max Restart = 4 )

### Restart 1

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



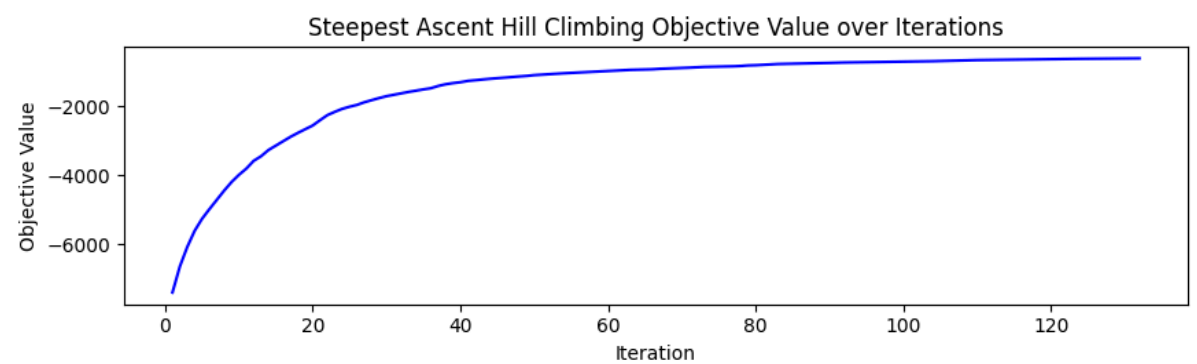
Final State					Final State					Final State					Final State					Final State				
90	53	117	42	13	121	68	38	21	67	2	65	45	89	114	44	115	47	106	3	59	14	73	75	94
51	72	25	124	43	1	9	78	108	119	95	120	34	30	36	91	26	66	31	100	104	88	112	4	20
28	64	86	15	122	123	46	22	55	70	101	83	80	17	33	11	6	110	125	56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

Final Objective Value: -1141 | Duration: 101355.041 ms | Iteration: 102

**Gambar 2.3.1.3.5.** Hasil dari test case 2 pada restart 1 untuk algoritma Random Restart Hill Climbing.

Restart 2

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



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

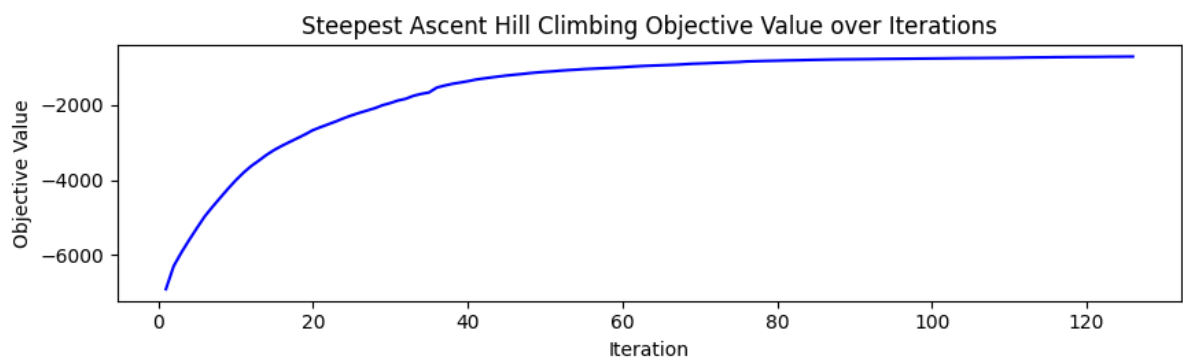
Final Objective Value: -602 | Duration: 112558.544 ms | Iteration: 132

Gambar 2.3.1.3.6. Hasil dari test case 2 pada restart 2 untuk algoritma Random Restart Hill Climbing.

Restart 3



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



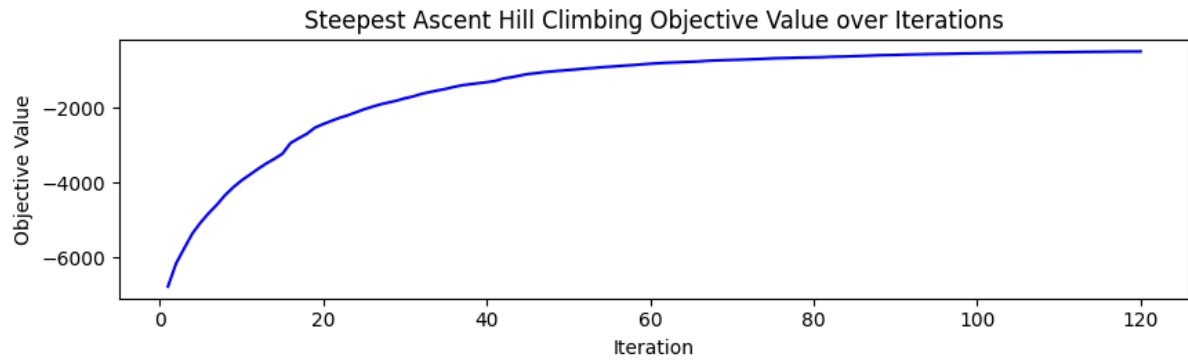
Final State					Final State					Final State					Final State					Final State				
78	3	110	108	16	47	105	1	89	88	32	77	63	83	60	39	31	123	15	97	117	106	18	20	54
22	90	91	41	70	64	55	50	100	28	87	111	13	14	103	61	51	80	118	6	81	8	84	36	109
112	10	79	66	48	74	93	58	71	19	9	38	68	62	125	122	120	37	11	25	5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

Final Objective Value: -714 | Duration: 107887.653 ms | Iteration: 126

**Gambar 2.3.1.3.7.** Hasil dari test case 2 pada restart 3 untuk algoritma Random Restart Hill Climbing.

### Restart 4

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



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

Final Objective Value: -494 | Duration: 106624.25600000001 ms | Iteration: 120

**Gambar 2.3.1.3.8.** Hasil dari test case 2 pada restart 4 untuk algoritma Random Restart Hill Climbing.

**Hasil Akhir :**

```
Final state: [47, 8, 116, 79, 65, 69, 98, 13, 9, 125, 46, 20, 67, 122, 60, 73, 112, 48, 61, 19, 84, 74, 71, 42, 44, 36, 56, 124, 49, 38, 75, 66, 51, 23, 107, 91, 33, 100, 40, 50, 28, 68, 6, 102, 109, 85, 93, 25, 101, 11, 22, 83, 15, 108, 90, 64, 95, 31, 111, 14, 103, 81, 80, 41, 10, 118, 26, 86, 3, 87, 4, 30, 110, 55, 115, 123, 106, 12, 72, 2, 18, 29, 121, 114, 32, 17, 89, 16, 77, 117, 53, 70, 62, 54, 76, 105, 21, 104, 1, 88, 82, 63, 45, 7, 120, 97, 27, 99, 59, 35, 57, 92, 52, 34, 78, 43, 39, 113, 96, 24, 37, 94, 5, 119, 58]
Iteration: 120
Objective value: -494
```

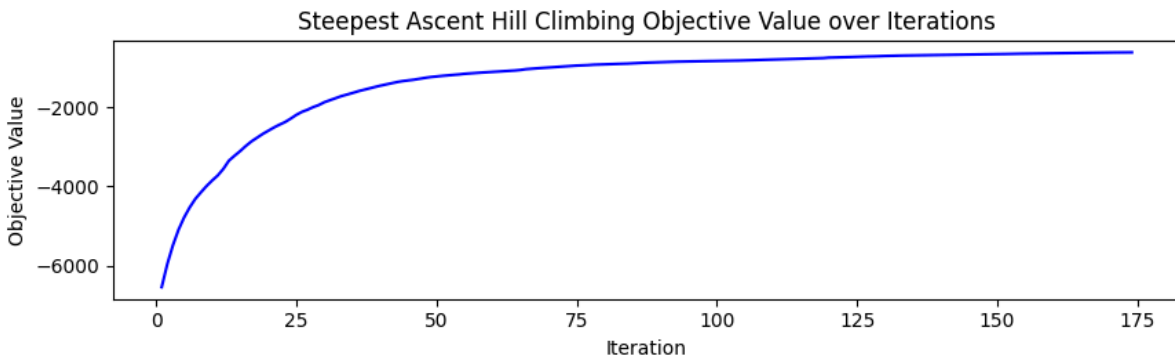
**Gambar 2.3.1.3.9.** Hasil akhir Test Case 2 untuk algoritma Random Restart Hill Climbing.

Di eksperimen ini, dilakukan 4 kali pengulangan terhadap pencarian solusi menggunakan algoritma Steepest Ascent Hill Climbing. Dari keempat pengulangan tersebut, akan dicatat jumlah iterasi dan nilai objektifnya, lalu akan dipilih nilai objektif yang paling mendekati 0. Pada eksperimen ini, hasil dari nilai objektif dari keempat pengulangan sangat bervariasi dan memiliki selisih yang tinggi (dapat dilihat nilai objektif pada pengulangan pertama hampir 3 kali dari nilai objektif dari pengulangan keempat). Karena state akhir pada pengulangan keempat mempunyai nilai objektif yang paling kecil, maka dipilihlah state akhir hasil dari pengulangan keempat menjadi hasil akhir dengan jumlah iterasi 120 kali dan nilai objektif sebesar -494.

### Test Case 3 ( Max Restart = 5 )

#### Restart 1

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



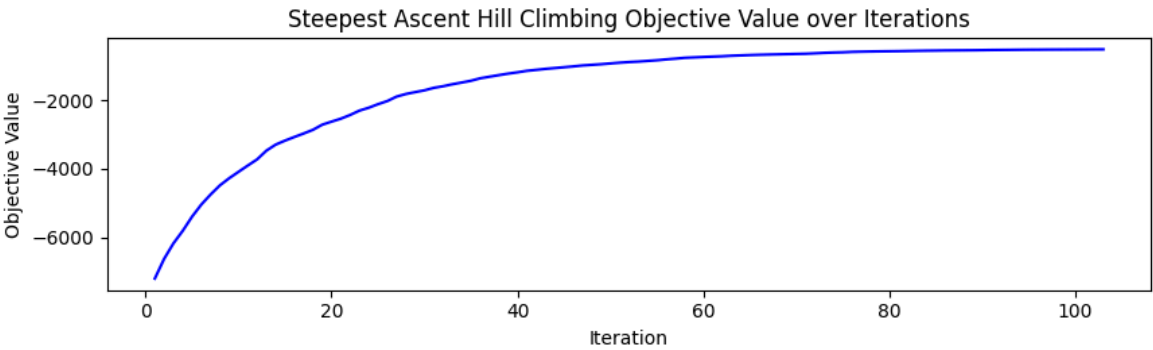
Final State					Final State					Final State					Final State					Final State				
31	66	69	112	37	109	51	44	8	102	40	32	118	4	121	55	79	49	99	33	73	85	35	103	22
106	88	18	7	96	91	23	30	74	97	38	101	107	68	2	64	21	47	61	122	17	83	113	105	1
34	41	76	114	46	98	27	82	20	90	59	117	58	26	56	52	120	54	80	9	72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

Final Objective Value: -620 | Duration: 150535.96 ms | Iteration: 174

**Gambar 2.3.1.3.10.** Hasil dari test case 3 pada restart 1 untuk algoritma Random Restart Hill Climbing.

### Restart 2

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



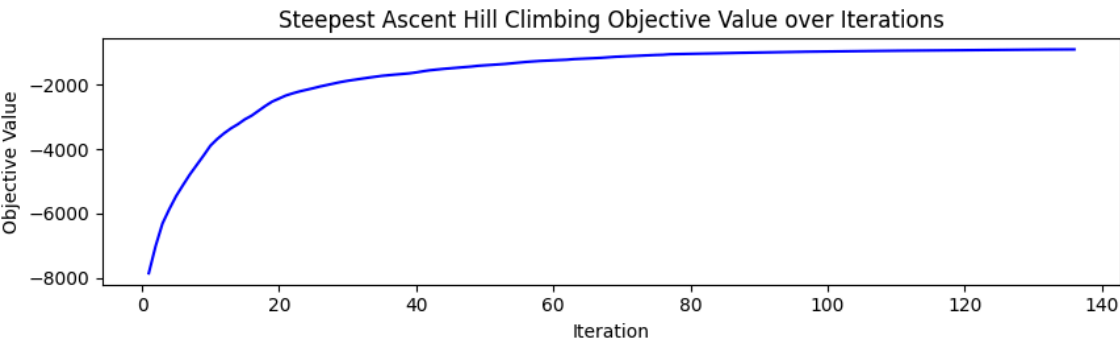
Final State					Final State					Final State					Final State					Final State				
81	96	17	106	15	55	22	115	30	93	87	24	107	13	89	56	123	43	46	42	37	49	33	117	80
59	1	112	23	120	79	75	6	77	78	121	62	16	45	64	47	91	82	110	3	9	103	99	63	44
18	10	98	105	84	26	92	114	72	11	48	69	53	60	85	125	31	14	70	52	88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

Final Objective Value: -516 | Duration: 90876.728 ms | Iteration: 103

**Gambar 2.3.1.3.11.** Hasil dari test case 3 pada restart 2 untuk algoritma Random Restart Hill Climbing.

Restart 3

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



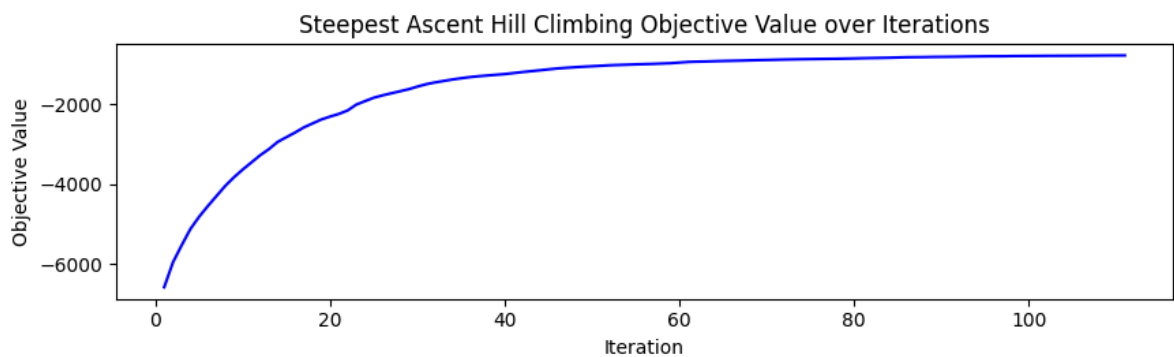
Final State					Final State					Final State					Final State					Final State				
59	18	118	66	55	100	14	20	115	61	3	121	34	81	76	110	77	31	91	6	46	43	108	1	117
74	82	22	122	17	56	107	37	45	72	23	50	84	53	106	41	48	78	39	109	120	70	92	26	10
112	28	40	21	114	51	30	95	42	98	73	85	57	86	11	16	38	111	87	63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

Final Objective Value: -896 | Duration: 121599.158 ms | Iteration: 136

**Gambar 2.3.1.3.12.** Hasil dari test case 3 pada restart 3 untuk algoritma Random Restart Hill Climbing.

Restart 4

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



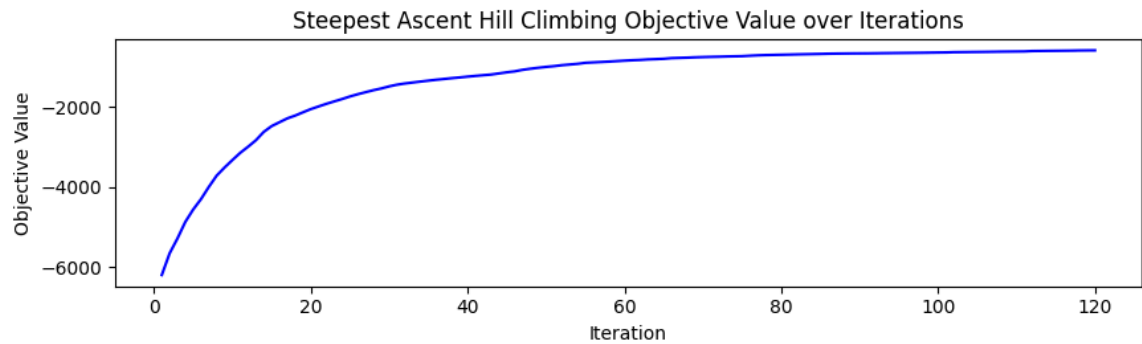
Final State					Final State					Final State					Final State					Final State				
75	45	112	56	26	28	77	52	35	121	41	46	97	101	29	106	61	50	15	84	65	86	4	109	51
47	83	6	102	79	115	69	24	80	27	42	81	67	12	113	13	31	123	82	66	94	58	95	38	30
1	34	54	118	108	125	68	74	40	8	116	19	57	36	87	63	96	25	111	20	10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

Final Objective Value: -785 | Duration: 96384.694 ms | Iteration: 111

**Gambar 2.3.1.3.13.** Hasil dari test case 3 pada restart 4 untuk algoritma Random Restart Hill Climbing.

## Restart 5

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



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

Final Objective Value: -585 | Duration: 105965.102 ms | Iteration: 120

**Gambar 2.3.1.3.14.** Hasil dari test case 3 pada restart 5 untuk algoritma Random Restart Hill Climbing.

#### Hasil Akhir:

```
Final state: [81, 96, 17, 106, 15, 59, 1, 112, 23, 120, 18, 10, 98, 105, 84, 90, 109, 34, 61, 21, 67, 101, 54, 20, 74, 55, 22, 115, 30, 93, 79, 75, 6, 77, 78, 26, 92, 114, 72, 11, 116, 5, 76, 27, 94, 40, 122, 4, 108, 41, 87, 24, 107, 13, 89, 121, 62, 16, 45, 64, 48, 69, 53, 60, 85, 19, 95, 104, 73, 25, 39, 66, 35, 124, 51, 56, 123, 43, 46, 42, 47, 91, 82, 110, 3, 125, 31, 14, 70, 52, 7, 68, 65, 57, 118, 71, 2, 111, 32, 100, 37, 49, 33, 117, 80, 9, 103, 99, 63, 44, 88, 113, 28, 8, 83, 86, 38, 36, 97, 58, 102, 12, 119, 29, 50]
Iteration: 103
Objective value: -516
```

**Gambar 2.3.1.3.15.** Hasil dari pemilihan state yang paling bagus setelah melakukan restart sebanyak kali

Di eksperimen ini, dilakukan 5 kali pengulangan terhadap pencarian solusi menggunakan algoritma Steepest Ascent Hill Climbing. Dari kelima pengulangan tersebut, akan dicatat jumlah iterasi dan nilai objektifnya, lalu akan dipilih nilai objektif yang paling mendekati 0. Pada eksperimen ini, hasil yang diperoleh dari 5 kali pengulangan bervariasi, namun selisihnya tidak terlalu jauh (rentang nilai objektif dari -896 sampai -516). Karena state akhir pada pengulangan ke 2 memiliki nilai objektif paling kecil, maka dipilihlah state akhir dari pengulangan kedua menjadi hasil akhir dengan jumlah iterasi 103 kali dan nilai objektif sebesar -516.

#### 2.3.1.3.1 Hasil Analisis Hill Climbing With Random Restart

Algoritma ini pada intinya memiliki cara kerja yang sama dengan algoritma Steepest Ascent Hill Climbing, karena algoritma ini hanya melakukan pengulangan pada algoritma Steepest Ascent Hill Climbing. Algoritma ini menambah peluang untuk terhindar dari lokal maksimum yang buruk dengan melakukan beberapa kali pengulangan, sehingga pada akhirnya



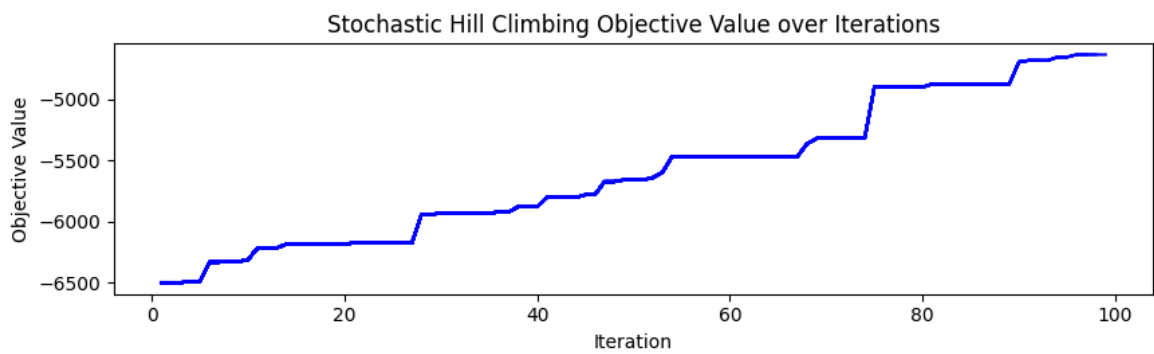
bisa memilih state akhir yang memiliki nilai objektif terbaik.

Algoritma ini membutuhkan waktu yang lebih panjang dari algoritma hill climbing yang lain tergantung jumlah iterasi yang dipilih pada algoritma ini. Semakin besar jumlah pengulangan, akan meningkatkan peluang mendapatkan state yang lebih baik dan terhindar dari jebakan lokal maksimum yang buruk. Namun di sisi lain, besarnya jumlah pengulangan akan memerlukan sumber daya yang lebih besar, karena akan memakan lebih banyak banyak waktu. Algoritma ini memberikan hasil yang dapat dibilang cukup memuaskan, terlihat dari 3 percobaan yang dilakukan di atas umumnya memiliki hasil objektif dibawah -550.

2.3.1.4 Stochastic Hill Climbing

Test Case 1

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



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

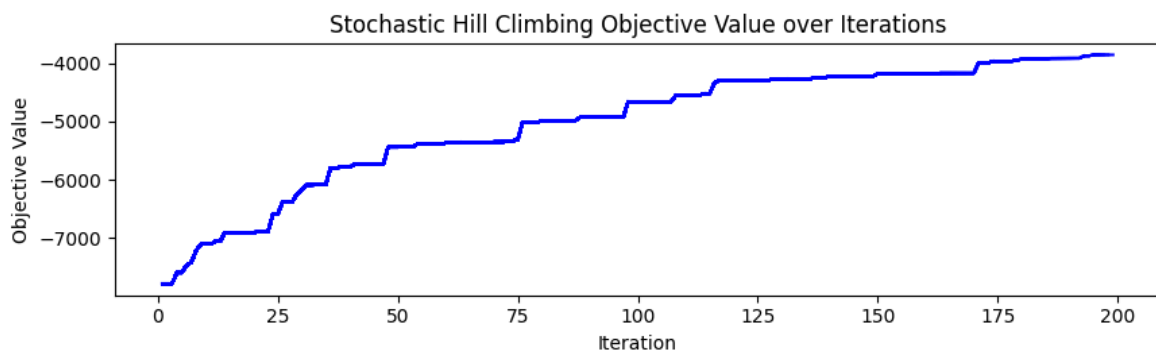
Final Objective Value: -4632 | Duration: 16303.543999999998 ms | Iteration: 100

### Gambar 2.3.1.4.1. Hasil dari test case 1 untuk algoritma Stochastic Hill Climbing.

Dari eksperimen ini , diketahui bahwa dari *initial state* persoalan diatas, dilakukan iterasi sebanyak 100 kali. Nilai objektif akhir yang didapat yakni adalah -4632 , dan memiliki durasi selama kurang lebih 16 detik.

## Test Case 2

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



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

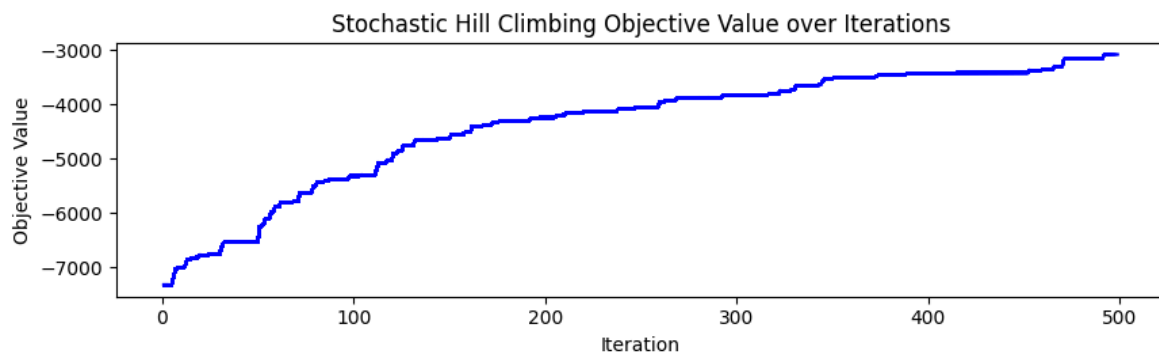
Final Objective Value: -3856 | Duration: 31909.746 ms | Iteration: 200

**Gambar 2.3.1.4.2.** Hasil dari test case 2 untuk algoritma Stochastic Hill Climbing.

Dari eksperimen ini , diketahui bahwa dari *initial state* persoalan diatas, dilakukan iterasi sebanyak 200 kali. Nilai objektif akhir yang didapat yakni adalah -3856 , dan memiliki durasi selama kurang lebih 32 detik.

### Test Case 3

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



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

Final Objective Value: -3078 | Duration: 82899.35800000001 ms | Iteration: 500

#### **Gambar 2.3.1.4.3.** Hasil dari test case 3 untuk algoritma Stochastic Hill Climbing.

Dari eksperimen ini , diketahui bahwa dari *initial state* persoalan diatas, dilakukan iterasi sebanyak 500 kali. Nilai objektif akhir yang didapat yakni adalah -3078 , dan memiliki durasi selama kurang lebih 83 detik.

#### **2.3.1.4.1 Hasil Analisis Stochastic Hill Climbing**

Pencarian *stochastic hill climbing* lebih berfokus pada keringanan iterasi ( yang dapat disesuaikan ) , namun sangat bergantung dengan jumlah iterasi. Iterasi yang semakin banyak memberikan hasil yang lebih optimal.

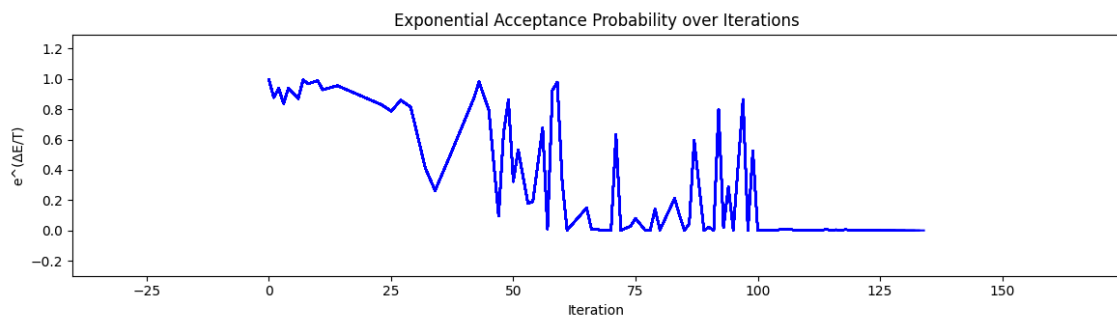
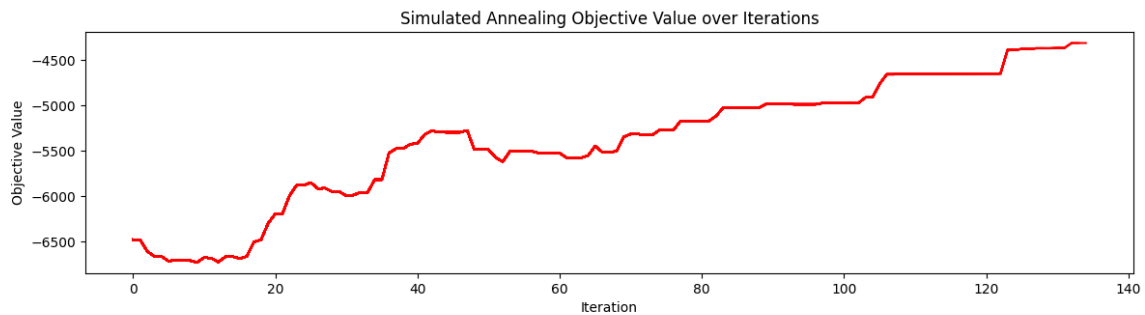
Namun, jika dibandingkan dengan algoritma hill climbing pada umumnya, terlihat bahwa terdapat efisiensi yang kurang baik dari sisi perbandingan iterasi dengan hasil objektif yang didapat. Hal ini disebabkan karena *stochastic hill climbing* memerlukan sumber daya yang lebih sedikit dimana algoritma ini mencari *neighbor* secara acak, bukan mencari *neighbor* terbaik ( perlu melakukan iterasi lagi ).

Atas hal tersebut, algoritma *stochastic hill climbing* lebih memerlukan waktu serta faktor acak dalam memberikan hasil optimal. Namun, cocok jika tidak ingin menghasilkan banyak sumber daya dalam pencarian solusi.

#### **2.3.2 Simulated Annealing Algorithm**

**Test Case 1 ( T0 = 1000, CoolingRate = 0.95, T1 = 1)**

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



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

Final Objective Value: -4311 Duration: 45120.686 ms Stuck Count: 63

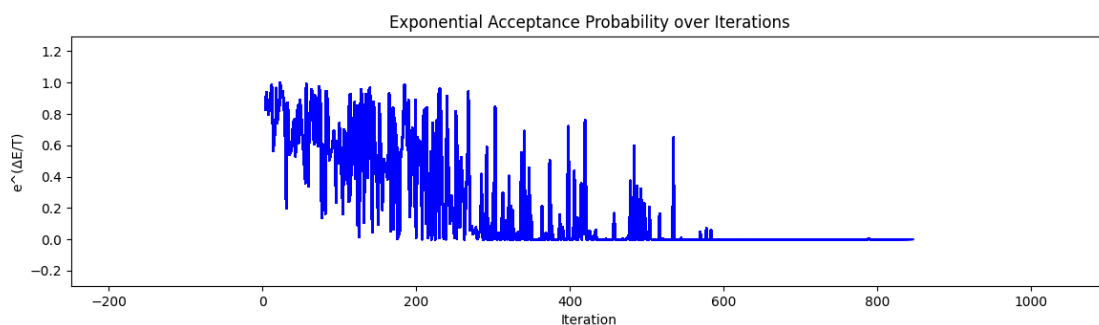
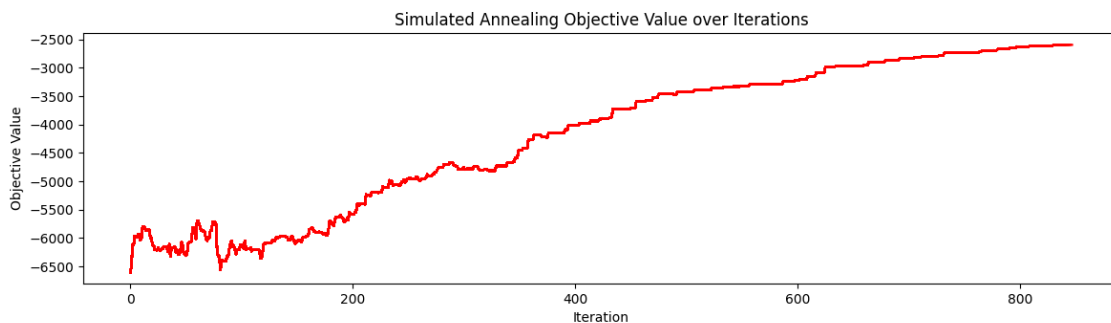
**Gambar 2.3.2.1.** Hasil dari test case 1 untuk algoritma Simulated Annealing.

Dari hasil eksperimen pertama yang dilakukan dengan menggunakan algoritma Simulated Annealing, didapatkan bahwa hasil dari pencarian yang dilakukan mendapatkan final state seperti yang terlampir dari final state yang terlampir. Nilai objektif dari final state yang

diperoleh bernilai -4311, dengan durasi eksekusi selama kurang lebih 46 detik. Kemudian, untuk banyak kali ditemukan neighbor dengan nilai objektif yang sama, yaitu sebanyak 63 kali, namun algoritma berhasil melepaskan diri. Kemudian, dari grafik Exponential Acceptance Probability, juga dapat dilihat bahwa probabilitas untuk berubah state menjadi sangat rendah (mendekati 0) ketika berada di atas iterasi 100.

## Test Case 2 ( $T_0 = 500$ , $\text{CoolingRate} = 0.99$ , $T_1 = 0.1$ )

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



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

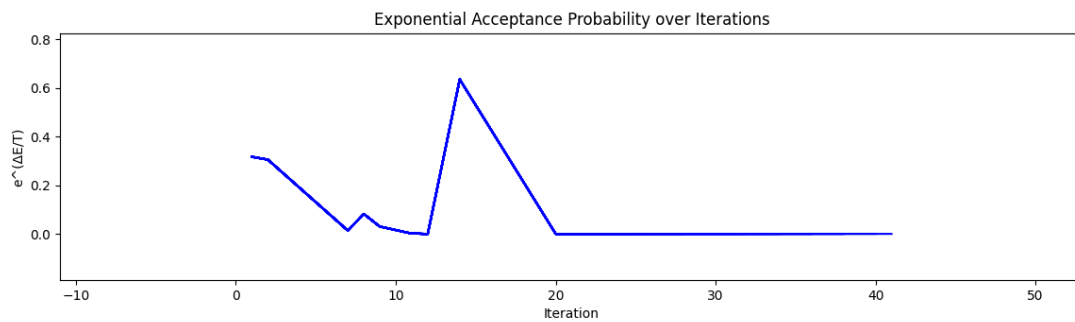
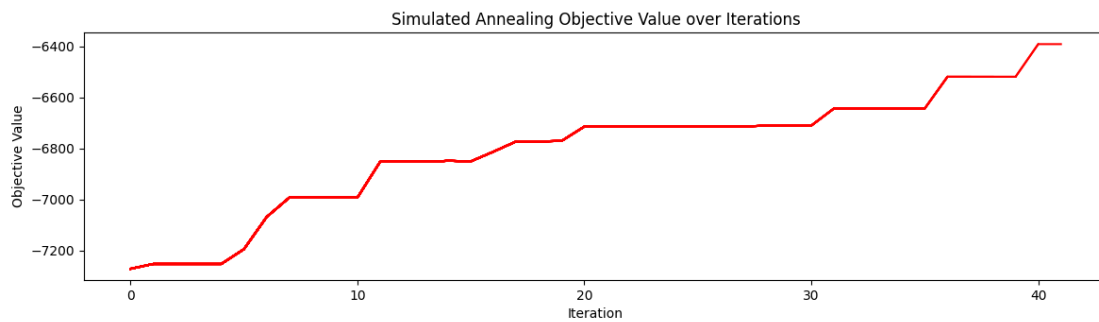
Final Objective Value: -2596 Duration: 570439.298 ms Stuck Count: 565

### Gambar 2.3.2.1. Hasil dari test case 2 untuk algoritma Simulated Annealing.

Dari hasil eksperimen kedua dengan menggunakan algoritma simulated annealing, dengan menggunakan initial state seperti yang terlampir pada gambar di atas. Dihasilkan final state juga seperti yang terlampir di atas. Hasil eksperimen menunjukkan bahwa nilai objektif final state bernilai -2596, dengan durasi eksekusi sebesar kurang lebih 570 detik. Kemudian, didapatkan pula informasi bahwa banyaknya kondisi yang ditemukan sama dengan current value, yaitu sebanyak 565 kali. Dari hasil visualisasi grafik Exponential Acceptance Probability, dapat dilihat bahwa probabilitas mendekati 0 setelah 600 iterasi dalam algoritma. Ini membuktikan bahwa algoritma berhasil lolos dari keadaan stuck sebanyak 565 kali.

### Test Case 3 ( $T_0 = 100$ , $CoolingRate = 0.8$ , $T_1 = 0.01$ )

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



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

Final Objective Value: -6392 Duration: 12487.579 ms Stuck Count: 23

**Gambar 2.3.2.3.** Hasil dari test case 3 untuk algoritma Simulated Annealing.

Dari hasil eksperimen ketiga yang dilakukan di atas, didapatkan final state seperti yang terlampir dengan initial state yang terlampir juga seperti di atas. Terdapat informasi bahwa nilai objektif untuk final state bernilai -6392, dengan durasi eksekusi kurang lebih 12 detik, dan banyak kondisi nilai objektif tetangga sama dengan nilai objektif sekarang sebanyak 23 kali. Kondisi ini dilakukan dengan parameter suhu awal 100, *cooling rate* sebesar 0.8, dan suhu threshold sebagai kondisi terminasi sebesar 0.01.



### 2.3.2.1. Hasil Analisis Simulated Annealing

Nomor Eksperimen	T0	Cooling Rate	T1	Nilai Objektif Final State	Waktu Eksekusi (ms)	Stuck Count	Banyak Iterasi
1	1000	0.95	1	-4311	45120	63	134
2	500	0.99	0.1	-2596	570439	565	833
3	100	0.8	0.01	-6392	12487	23	42

**Tabel 2.3.2.1.1.** Tabel Hasil Eksperimen yang dilakukan dengan Simulated Annealing

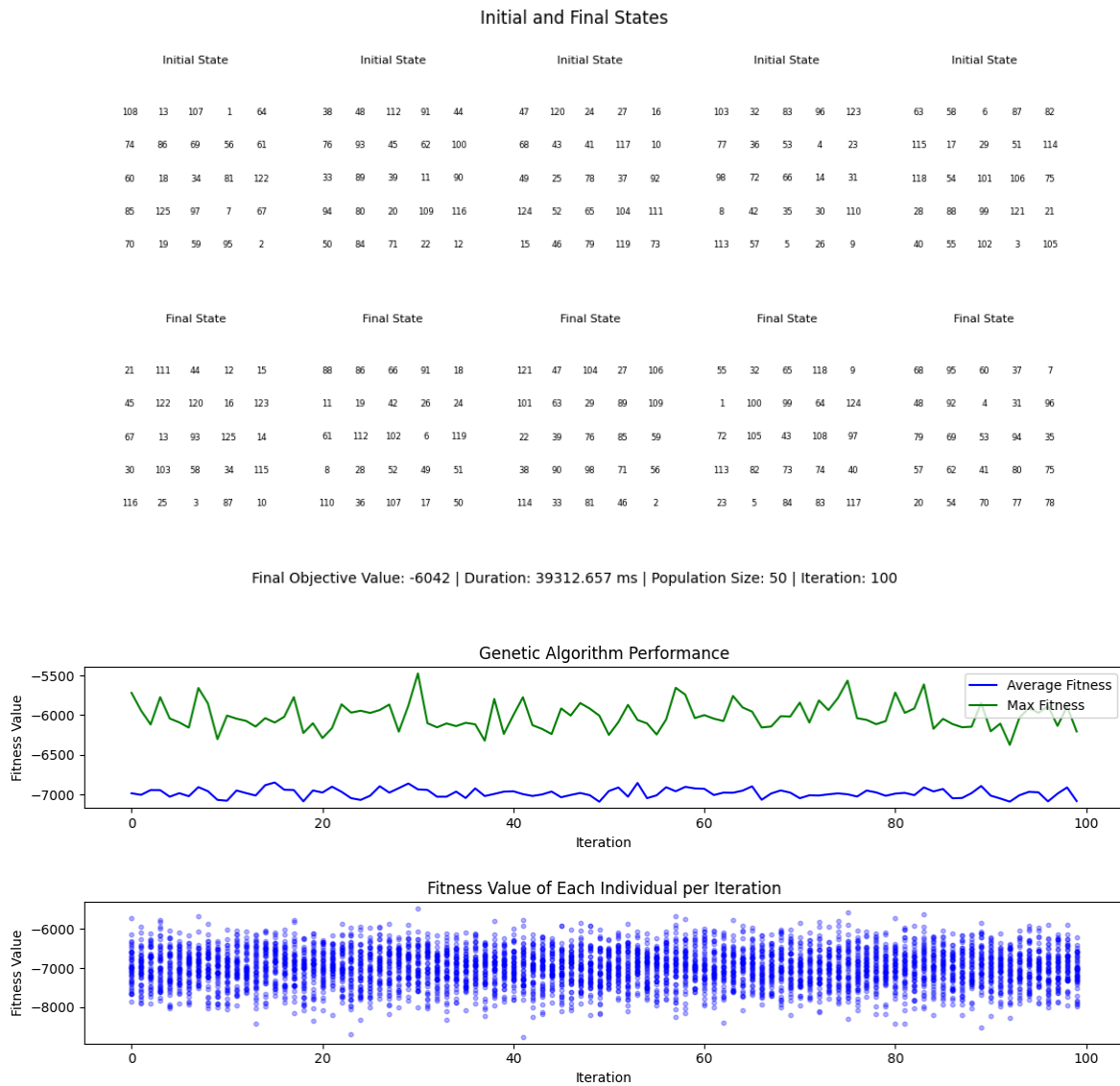
Dari hasil ketiga eksperimen yang telah dilakukan dengan menggunakan simulated annealing, dapat dilihat bahwa nilai objektif final state yang didapatkan bernilai -2596. Namun, hasil terbaik yang didapatkan melewati iterasi sebanyak 833 kali. Sedangkan, untuk yang terburuk adalah dengan nilai objektif final state bernilai -6392 pada eksperimen ketiga, dengan banyak iterasi sebanyak 42 kali.

Dari hasil ketiga eksperimen, dengan T0, *cooling rate*, dan T1 yang berbeda-beda, dapat dilihat pada akhirnya yang menentukan apakah simulated annealing akan menghasilkan state final yang bagus adalah banyak iterasi dengan konfigurasi parameter yang telah ditentukan. Semakin banyak iterasi yang dapat dibentuk oleh konfigurasi parameter tersebut, maka akan semakin tinggi kemungkinannya untuk mendapatkan final state yang memiliki nilai objektif yang mendekati 0. Namun, kekurangan dari hasil bagus yang didapatkan melalui algoritma simulated annealing adalah waktu eksekusi yang cenderung meningkat secara eksponensial searah dengan berkembangnya banyak iterasi yang terjadi dengan konfigurasi parameter tersebut.

Oleh karena itu, untuk mendapatkan nilai yang baik pada Simulated Annealing diperlukan *tuning* pada konfigurasi parameter, sehingga mendapatkan hasil yang sesuai dengan keinginan atau kebutuhan yang diperlukan pengguna. Kemudian, keuntungan lain simulated annealing, adalah fleksibilitasnya dalam pengaturan banyak iterasi yang dapat dilakukan dengan melakukan *tuning* pada parameter konfigurasi yang telah disebutkan sebelumnya.

### 2.3.3 Genetic Algorithm

#### Test Case 1 ( Untuk perubahan iterasi terhadap populasi yang sama)



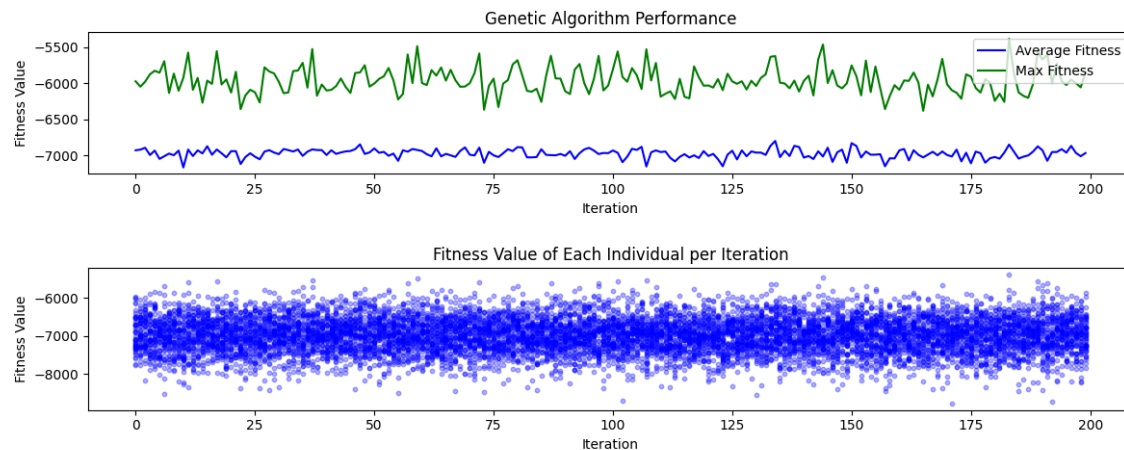
**Gambar 2.3.3.1.** Hasil dari test case 1 untuk algoritma Genetic.

Dari eksperimen ini dilakukan iterasi sebanyak 100 kali dengan jumlah populasi sebanyak 50. Nilai objektif akhir yang didapat yakni adalah -6042 , dan memiliki durasi selama kurang lebih 39 detik.

#### Test Case 2 ( Untuk perubahan iterasi terhadap populasi yang sama)

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

Final Objective Value: -6046 | Duration: 93766.611 ms | Population Size: 50 | Iteration: 200



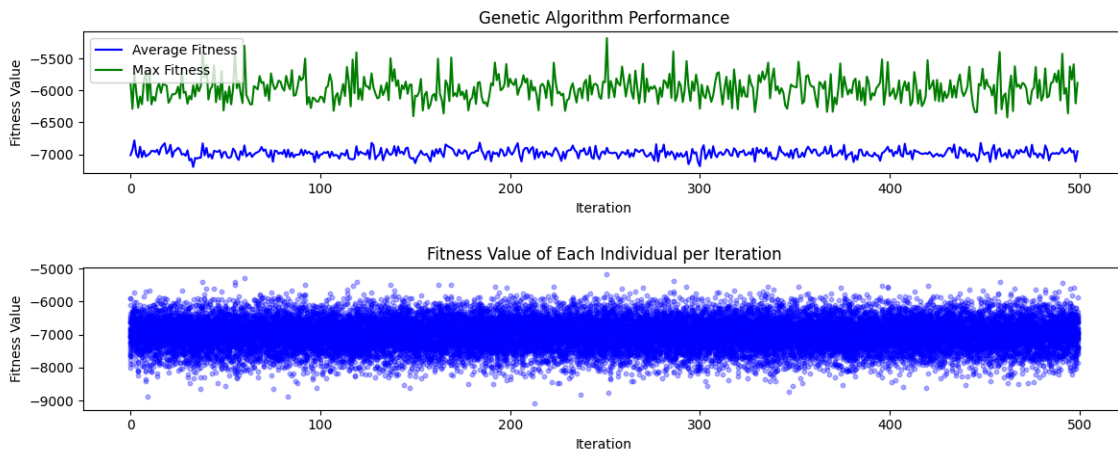
**Gambar 2.3.3.2.** Hasil dari test case 2 untuk algoritma Genetic.

Dari eksperimen ini dilakukan iterasi sebanyak 200 kali dengan jumlah populasi sebanyak 50. Nilai objektif akhir yang didapat yakni adalah -6046 , dan memiliki durasi selama kurang lebih 94 detik.

### Test Case 3 ( Untuk perubahan iterasi terhadap populasi yang sama)

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

Final Objective Value: -5811 | Duration: 382995.221 ms | Population Size: 50 | Iteration: 500



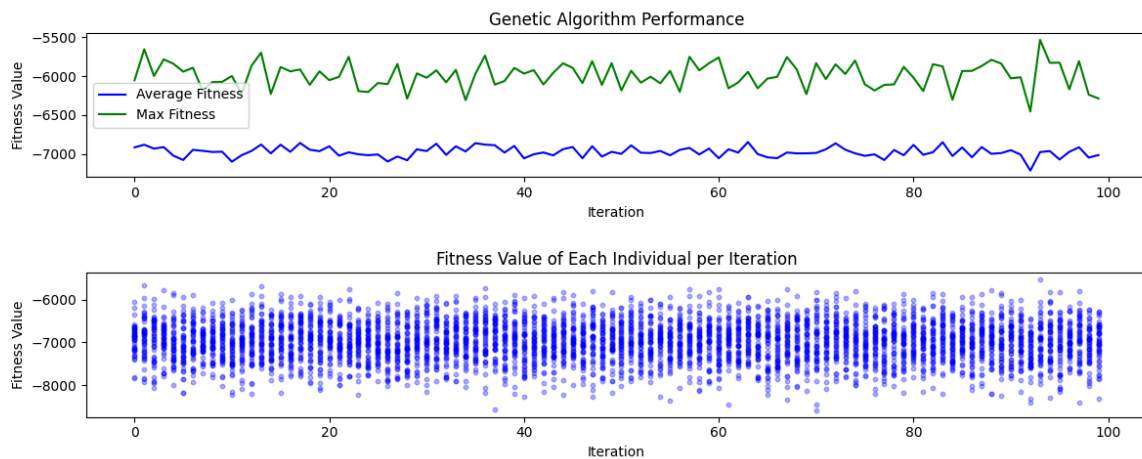
**Gambar 2.3.3.3.** Hasil dari test case 3 untuk algoritma Genetic.

Dari eksperimen ini dilakukan iterasi sebanyak 500 kali dengan jumlah populasi sebanyak 50. Nilai objektif akhir yang didapat yakni adalah -5811 , dan memiliki durasi selama kurang lebih 382 detik.

**Test Case 4 ( Untuk perubahan populasi terhadap iterasi yang sama)**

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

Final Objective Value: -6105 | Duration: 37409.295999999995 ms | Population Size: 50 | Iteration: 100



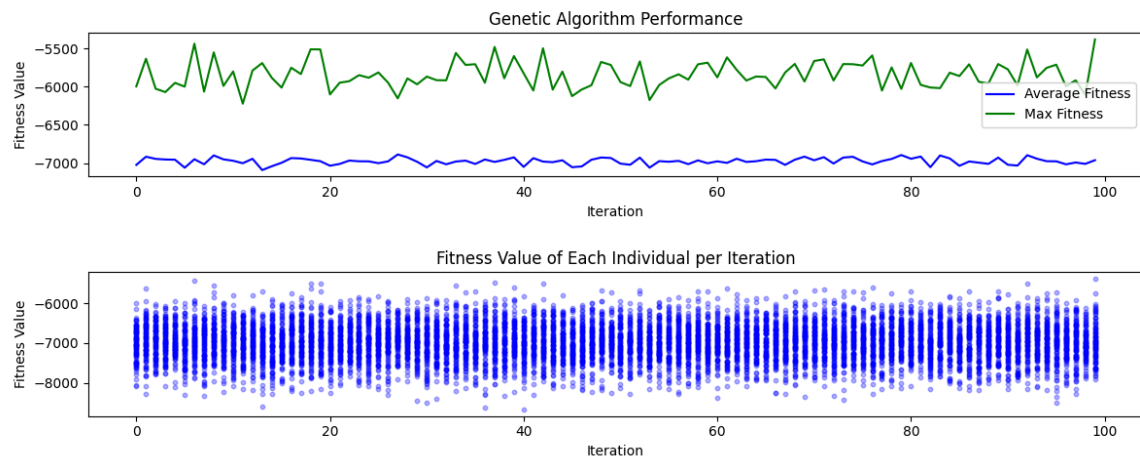
**Gambar 2.3.3.4.** Hasil dari test case 4 untuk algoritma Genetic.

Dari eksperimen ini dilakukan iterasi sebanyak 100 kali dengan jumlah populasi sebanyak 50. Nilai objektif akhir yang didapat yakni adalah -6105 , dan memiliki durasi selama kurang lebih 37 detik.

### Test Case 5 ( Untuk perubahan populasi terhadap iterasi yang sama)

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

Final Objective Value: -5773 | Duration: 81729.47 ms | Population Size: 100 | Iteration: 100



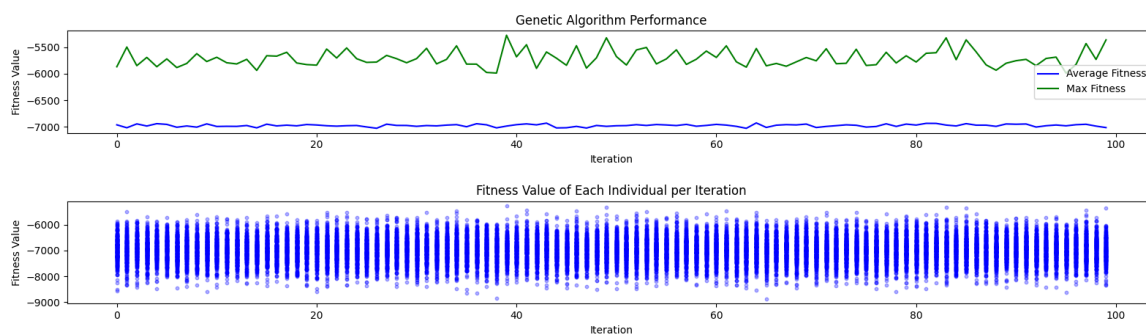
**Gambar 2.3.3.5.** Hasil dari test case 5 untuk algoritma Genetic.

Dari eksperimen ini dilakukan iterasi sebanyak 100 kali dengan jumlah populasi sebanyak 100. Nilai objektif akhir yang didapat yakni adalah -5773 , dan memiliki durasi selama kurang lebih 81 detik.

### Test Case 6 ( Untuk perubahan populasi terhadap iterasi yang sama)

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

Final Objective Value: -5515 | Duration: 544610.454 ms | Population Size: 300 | Iteration: 100



**Gambar 2.3.36.** Hasil dari test case 6 untuk algoritma Genetic.

Dari eksperimen ini dilakukan iterasi sebanyak 100 kali dengan jumlah populasi sebanyak 300. Nilai objektif akhir yang didapat yakni adalah -5515 , dan memiliki durasi selama kurang lebih 544 detik.

### 2.3.3.1 Hasil Analisis Genetic Algorithm

Setelah melakukan uji coba untuk berbagai jenis perubahan, dari sisi iterasi maupun populasi, dan juga memperhatikan faktor waktu dan nilai objektif akhir, diambil kesimpulan bahwa *genetic algorithm* tidak cocok untuk dijadikan metode untuk menyelesaikan persoalan *magic cube*, mengingat bahwa program ini memerlukan sumber daya yang sangat banyak serta memberikan hasil yang kurang memuaskan.

Terkait dengan penggunaan kontrol untuk setiap variabel, didapati bahwa perubahan terhadap iterasi lebih sedikit mempengaruhi hasil akhir dibandingkan dengan perubahan populasi. Hal ini disebabkan dari hasil *plot* terhadap *fitness* terlihat bahwa untuk setiap iterasi sebenarnya memiliki *average* serta *max fitness* yang cukup konsisten. Nilai dari distribusi nilai *fitness* untuk setiap anggota pada populasi juga terlihat konsisten untuk seluruh iterasi.

Di sisi lain, dengan menambah populasi, membuka lebih banyak kesempatan untuk menghasilkan *state* yang memiliki nilai objektif yang lebih besar dalam sebuah iterasi. Hal ini terlihat dari perubahan dari nilai objektif akhir yang lebih berefek ketika ditambahkan populasi dibandingkan dengan penambahan iterasi. Penyebabnya dapat dijelaskan dengan hasil visualisasi nilai *fitness* dibandingkan untuk seluruh anggota populasi pada setiap iterasi yang menunjukkan keseringan adanya butiran yang sempat melewati batas atas yang dibuat oleh iterasi sebelumnya, dimana hal ini lebih jarang terlihat pada populasi rendah namun iterasi tinggi. Pada iterasi tinggi dan populasi rendah, terlihat bahwa hasil visualisasi hanya menyentuh ambang batas -6000, sedangkan untuk populasi tinggi berkisar antara -5500 sampai -5000. Ini membuktikan bahwa populasi lebih berpengaruh dalam penentuan nilai objektif yang lebih baik.

### 2.3.4 Perbandingan tiap algoritma

Perbandingan pemilihan tiap algoritma dapat dihitung dari 3 sisi, yakni durasi, konsistensi dan ketepatan dibandingkan dengan global optima.

#### 1. Durasi

Dari sisi durasi, pendekatan yang lebih kompleks tentu akan memiliki durasi yang lebih lama. Pendekatan seperti *random restart* dan *hill climbing with sideways move* tentu akan memberikan durasi yang lebih lama jika dibandingkan dengan pendekatan lainnya seperti *steepest ascent hill climbing*. (Tergantung dari jumlah iterasi maksimal). Untuk komposisi penghasil solusi yang paling cepat ada pada *steepest ascent hill climbing*. Hal ini disebabkan karena pendekatan ini lebih tidak selektif dalam pemilihan solusi yang dianggap sebagai solusi paling baik, berbeda dengan *random restart* yang terus melakukan perulangan dengan algoritma *steepest ascent hill climbing* ataupun *hill climbing with sideways move* yang lebih selektif dalam memilih solusi.

#### 2. Konsistensi



Dari sisi konsistensi, *random restart hill climbing* memiliki konsistensi yang paling tinggi. Hal ini disebabkan karena pendekatan ini pasti akan memberikan solusi paling optimal dari berbagai pendekatan *steepest ascent hill climbing*. Dengan begitu, hasilnya akan selalu berada di lingkup *state* nilai objektif yang bagus. Pendekatan lain yaitu *Genetic algorithm*, yang selalu konsisten namun dalam artian buruk. Pembuktian ini dapat terlihat dari hasil visualisasi yang selalu konsisten untuk tiap iterasi, jika iya terjadi perubahan dalam setiap pengambilan populasi ataupun iterasi, penyimpangan nilai objektif yang terjadi cukup kecil. Jika diambil dari hasil eksperimen, untuk tipe algoritma *genetic algorithm*, perbedaan nilai objektif untuk setiap eksperimen hanya memiliki penyimpangan <500, dengan begitu dapat dikatakan memiliki nilai objektif yang cukup konsisten. Namun, tentu saja, memiliki nilai objektif dengan ambang -5000 hingga -6000 bukan merupakan sebuah hasil akhir yang diharapkan.

### 3. Ketepatan

Dari seluruh hasil eksperimen yang telah dilakukan dengan berbagai jenis algoritma, didapatkan bahwa algoritma *steepest ascent hill climbing* memberikan nilai objektif yang paling mendekati dengan nilai nol. Jika dilihat dari hasil eksperimen, terlihat bahwa rata-rata nilai objektif yang dihasilkan oleh algoritma ini adalah di atas -1000, yang mana memberikan hasil yang paling baik atau nilai objektif yang paling mendekati nol dibandingkan dengan algoritma lainnya.

Selain itu, didapatkan pula informasi bahwa, algoritma yang memberikan ketepatan yang paling rendah dibandingkan dengan algoritma-algoritma lainnya adalah algoritma *genetic algorithm*. Algoritma ini memberikan hasil nilai objektif final state yang berkisar -6000 hingga -5000, yang mana jika dibandingkan dengan algoritma lainnya, memberikan nilai objektif yang cenderung rendah.

Terdapat juga algoritma Simulated Annealing, yang pada salah eksperimennya, memiliki nilai objektif yang berada di bawah nilai -6000, yang mana juga rendah, namun hasil tersebut tidak konsisten tergantung dengan nilai konfigurasi parameter yang ditentukan untuk menjalankan algoritma simulated annealing tersebut. Hal tersebut dibuktikan dengan hasil dua eksperimen lainnya, yang memberikan hasil yang lebih baik,

dan memiliki gap yang cukup jauh, yang mana nilai objektif yang lebih tinggi berbanding lurus dengan banyak iterasi yang dialami per eksperimennya.

### **BAB III : KESIMPULAN DAN SARAN**

Dalam rangka pencarian solusi *diagonal magic cube* dengan *local search*, penulis menggunakan beberapa algoritma *local search*, seperti algoritma *steepest ascent hill climbing*, *stochastic hill climbing*, *random restart hill climbing*, *sideways move hill climbing*, *simulated annealing*, dan *genetic algorithm*. Melalui hasil penyelesaian permasalahan *diagonal magic cube* 5x5x5 tersebut dengan menggunakan berbagai jenis algoritma local search tersebut, para penulis mengimplementasikan algoritma tersebut ke dalam kode yang kemudian dieksperimenkan, lalu mendapatkan hasil eksperimen yang kemudian dianalisis oleh penulis. Untuk membandingkan algoritma-algoritma tersebut, penulis melakukan komparasi akhir berdasarkan tiga *scope*, yaitu durasi, konsistensi, dan ketepatan.

Dari sudut pandang durasi, penulis menyimpulkan bahwa algoritma *steepest ascent hill climbing* merupakan algoritma yang paling baik dari segi durasi, yang mana algoritma ini memberikan hasil akhir final state beserta dengan final state dengan waktu yang lebih cepat dibandingkan dengan algoritma-algoritma lainnya. Algoritma *steepest ascent hill climbing* lebih cepat dikarenakan pendekatannya yang memiliki banyak kondisi terminasi dikarenakan algoritma hanya memilih state *neighbor* yang memiliki nilai objektif yang lebih baik. Dengan adanya kondisi terminasi ini, banyak iterasi yang dilakukan oleh algoritma ini cenderung lebih sedikit, namun hasil yang diberikan cukup baik dikarenakan pengambilan *neighbor state* ditentukan berdasarkan nilai objektif yang terbaik.

Selain itu, penulis juga melakukan analisis dari segi konsistensi pemberian nilai objektif final state oleh algoritma. Penulis memutuskan bahwa algoritma dengan konsistensi yang paling tinggi adalah algoritma *random restart hill climbing*. Hal ini disebabkan karena pendekatan ini pasti akan memberikan solusi paling optimal dari berbagai pendekatan *steepest ascent hill climbing*. Dengan begitu, hasilnya akan selalu berada di lingkup *state* nilai objektif yang bagus. Selain itu, terdapat juga algoritma *genetic algorithm* yang cukup konsisten, namun tidak terlalu dipertimbangkan penulis karena hasil final state yang diberikan mempunyai nilai objektif yang relatif rendah.

Terakhir, terdapat juga analisis oleh penulis dari segi ketepatan. Para penulis memutuskan bahwa algoritma yang paling tepat dalam menentukan keputusan akhir berupa final state adalah *steepest ascent hill climbing*. Hal tersebut terbukti dengan hasil eksperimen yang telah dilakukan oleh para penulis yang menunjukkan bahwa algoritma ini memberikan hasil final state dengan nilai objektif di atas -1000, sebanyak 3 kali percobaan secara berturut. Hasil algoritma ini juga telah dibandingkan dengan algoritma yang lainnya, dan hanya algoritma *steepest ascent hill climbing* yang memberikan nilai objektif yang paling baik dan dengan sifat konsisten pula.

Dengan demikian, dari seluruh hasil analisis yang telah dilakukan oleh penulis, penulis memutuskan bahwa algoritma yang paling baik untuk menyelesaikan persoalan diagonal magic cube 5x5x5 dengan metode local search adalah algoritma *steepest ascent hill climbing*. Keputusan ini bertentangan dengan hipotesis yang telah ditentukan oleh keempat penulis pada penelitian rancangan yang sebelumnya pernah dilakukan. Sedangkan, untuk algoritma *simulated annealing*, yang sempat dikatakan sebagai algoritma yang paling direkomendasikan pada hipotesis awal, atau sebelum dilakukan eksperimen, ternyata memberikan hasil yang kurang memuaskan dibandingkan ekspektasi yang diberikan. Namun, penulis tetap menyatakan bahwa algoritma *simulated annealing* merupakan algoritma yang paling fleksibel karena pengguna dapat mengatur nilai objektif akhir yang ingin dicapai, dengan mengatur konfigurasi parameter berupa  $T_0$ , *cooling rate*, dan  $T_1$ , namun dengan *trade off*, waktu eksekusi yang lebih lama dibandingkan dengan algoritma *steepest ascent hill climbing*.

## **BAB IV : PEMBAGIAN TUGAS TIAP KELOMPOK**

Nama / NIM	Tugas yang dikerjakan
Wilson Yusda / 13522019	- Genetic Algorithm
Filbert / 13522021	- Simulated Annealing
Elbert Chailes / 13522045	- Steepest Ascent Hill Climbing - Sideways Move Hill Climbing
Benardo / 13522055	- Random Restart Hill Climbing - Stochastic Hill Climbing

## REFERENSI

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