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

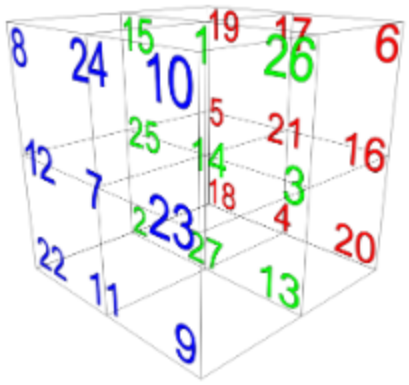
2024

DAFTAR ISI

BAB I : DESKRIPSI PERSOALAN

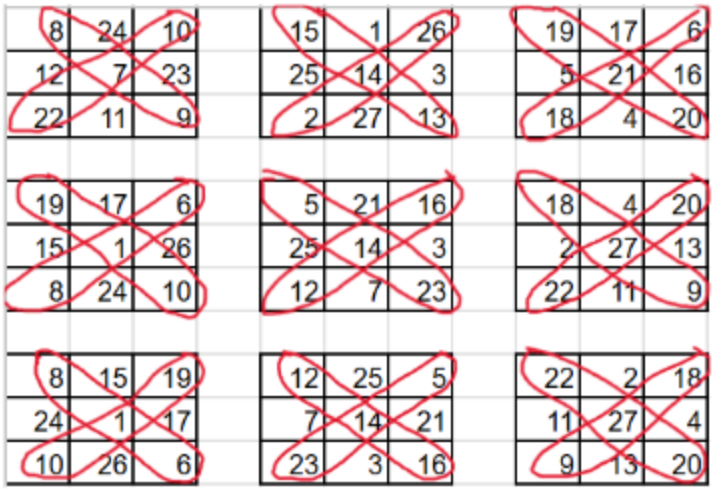
Diagonal magic cube merupakan kubus yang tersusun dari angka 1 hingga n3 tanpa pengulangan dengan n adalah panjang sisi pada kubus tersebut. Angka-angka pada tersusun sedemikian rupa sehingga properti-properti berikut terpenuhi:

- Terdapat satu angka yang merupakan magic number dari kubus tersebut (Magic number tidak harus termasuk dalam rentang 1 hingga n3, 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:



Gambar 1.2. Potongan Bidang Magic cube

- Diagonal yang dimaksud adalah yang dilingkari warna merah saja

BAB II : PEMBAHASAN

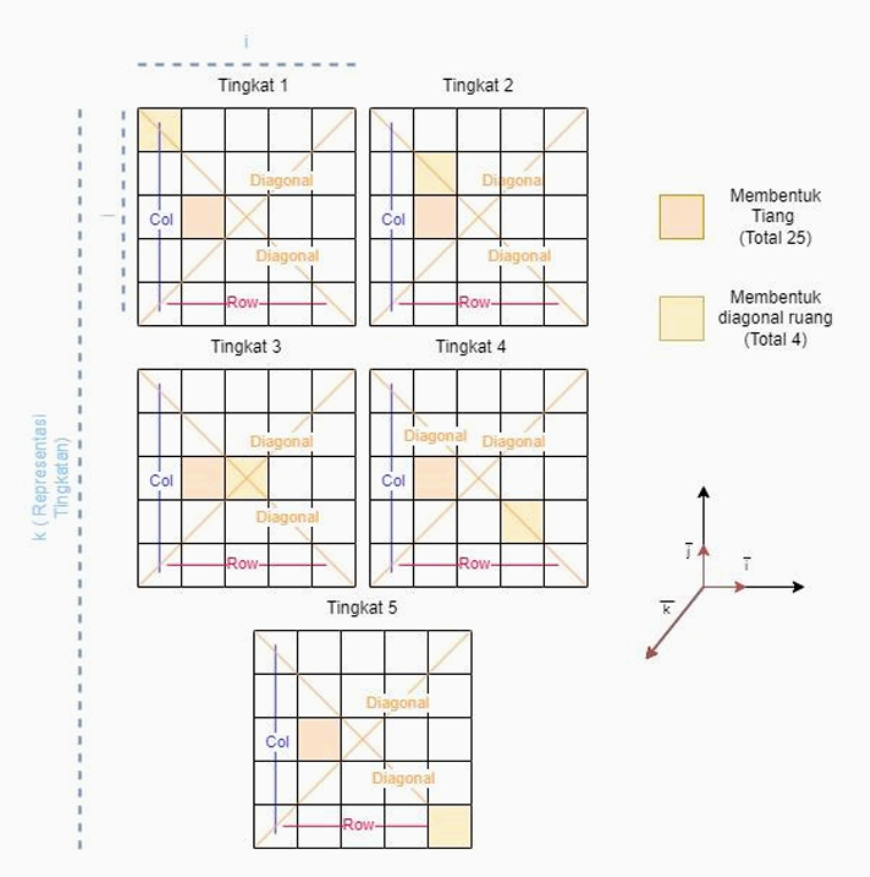
2.1 Pemilihan Objective Function

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

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

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 permisalan 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 Drow adalah dengan menghitung total deviasi yang pada total 25 row. Cara yang sama dilakukan untuk D_{column} , D_{pillar} , $D_{diagonal}$, dan $D_{triagonal}$. 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)

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

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

Bidang YZ (jk)

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

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

Bidang XZ (ik)

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

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

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 (-∞,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 redudansi 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)
        triagonal_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 triagonal_indices:
            triagonal_sum = sum(state[idx] for idx in indices)
            objective += abs(triagonal_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* >= 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 detil,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 *simmulated_annealing* merupakan fungsi utama untuk menyelesaikan permasalahan *magic cube*. Dalam *simmulated_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

122	21	12	42	30
82	52	125	102	2
93	10	103	26	14
4	117	19	120	106
38	83	47	85	28

Initial State

80	40	65	32	69
86	60	84	62	56
91	55	87	76	72
81	24	43	7	113
88	27	124	18	57

Initial State

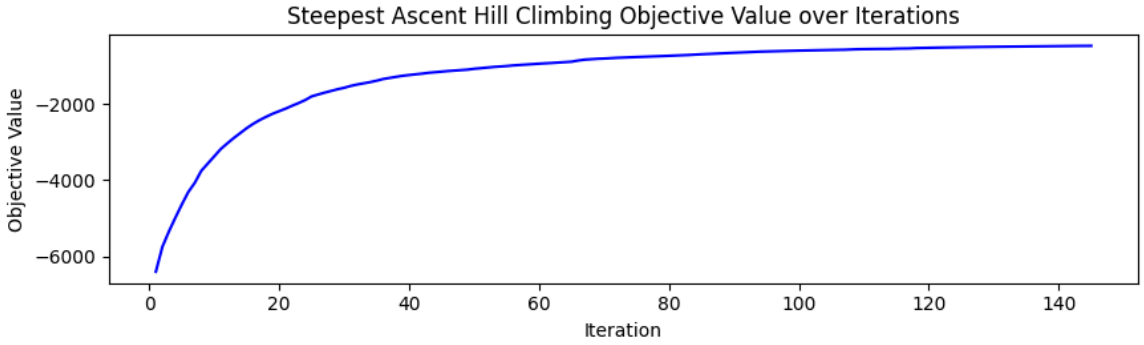
114	118	29	6	96
44	48	121	111	39
25	3	74	16	119
115	101	77	11	8
107	63	45	99	100

Initial State

15	36	61	94	98
70	89	59	31	33
35	104	50	75	79
9	37	64	58	78
108	123	105	53	13

Initial State

95	1	22	54	34
116	97	23	73	5
90	49	20	68	41
112	66	51	17	71
67	110	92	109	46



Final State

70	123	13	26	83
76	6	125	14	75
94	51	103	25	42
10	53	29	124	100
65	82	43	114	15

Final State

61	48	78	113	16
117	58	21	74	45
1	106	64	87	57
69	84	40	24	98
67	19	112	18	99

Final State

110	122	27	11	41
50	34	60	116	52
33	28	62	88	105
119	101	77	7	8
3	30	89	92	109

Final State

9	12	86	102	107
72	121	59	31	32
104	35	47	36	93
22	23	120	90	38
108	118	4	56	46

Final State

80	5	111	63	68
2	97	20	81	115
85	95	39	79	17
91	54	49	55	71
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

103	28	13	54	91
23	32	56	101	40
58	100	39	47	24
112	123	52	75	113
55	57	43	31	4

Initial State

18	64	84	1	85
63	7	12	73	21
90	70	119	108	59
117	41	42	96	111
61	50	49	105	29

Initial State

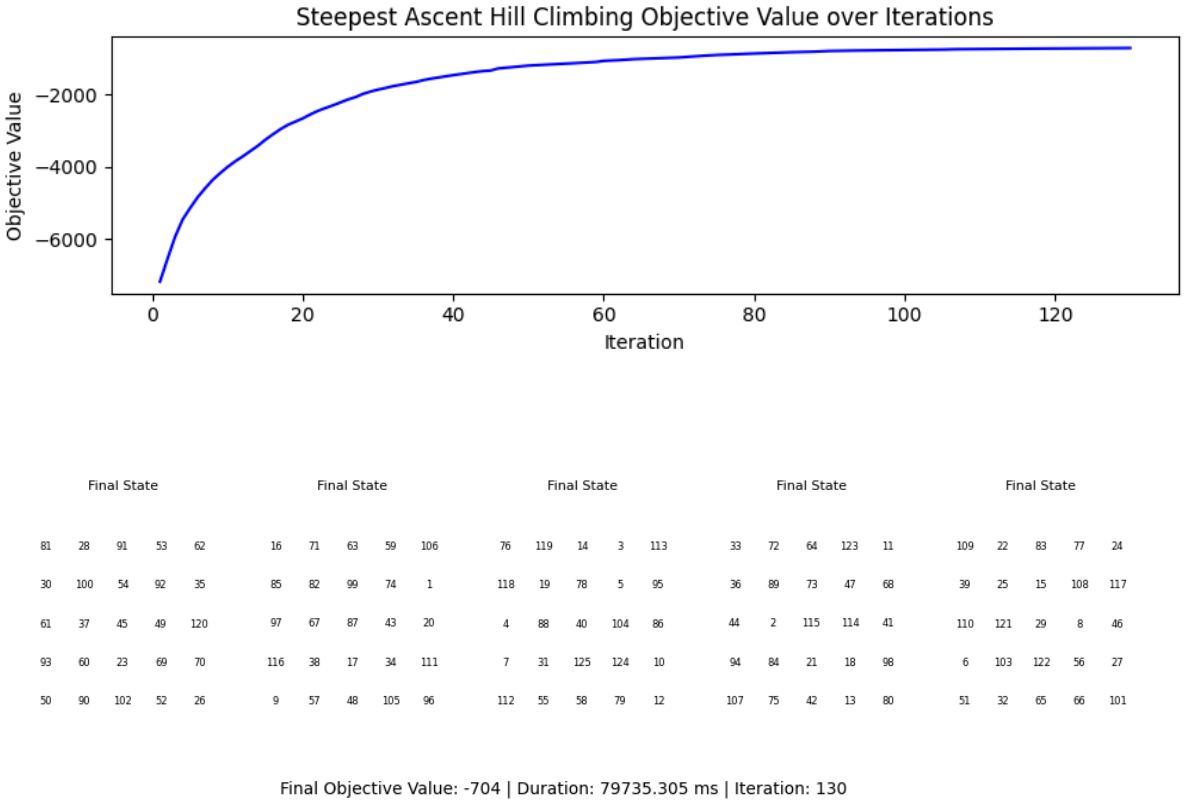
69	115	15	26	122
65	20	88	30	95
5	104	6	60	116
3	37	71	86	44
118	79	92	83	14

Initial State

33	48	9	89	11
16	98	74	10	53
67	124	8	114	38
107	35	25	19	87
80	121	46	22	51

Initial State

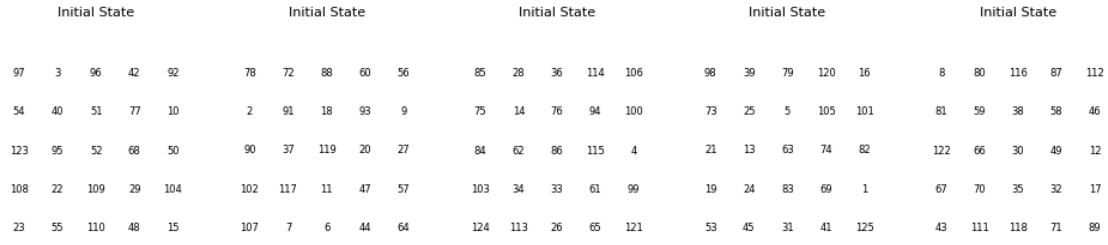
109	94	36	77	62
27	72	2	106	97
81	120	17	99	45
82	76	93	110	34
78	102	68	66	125

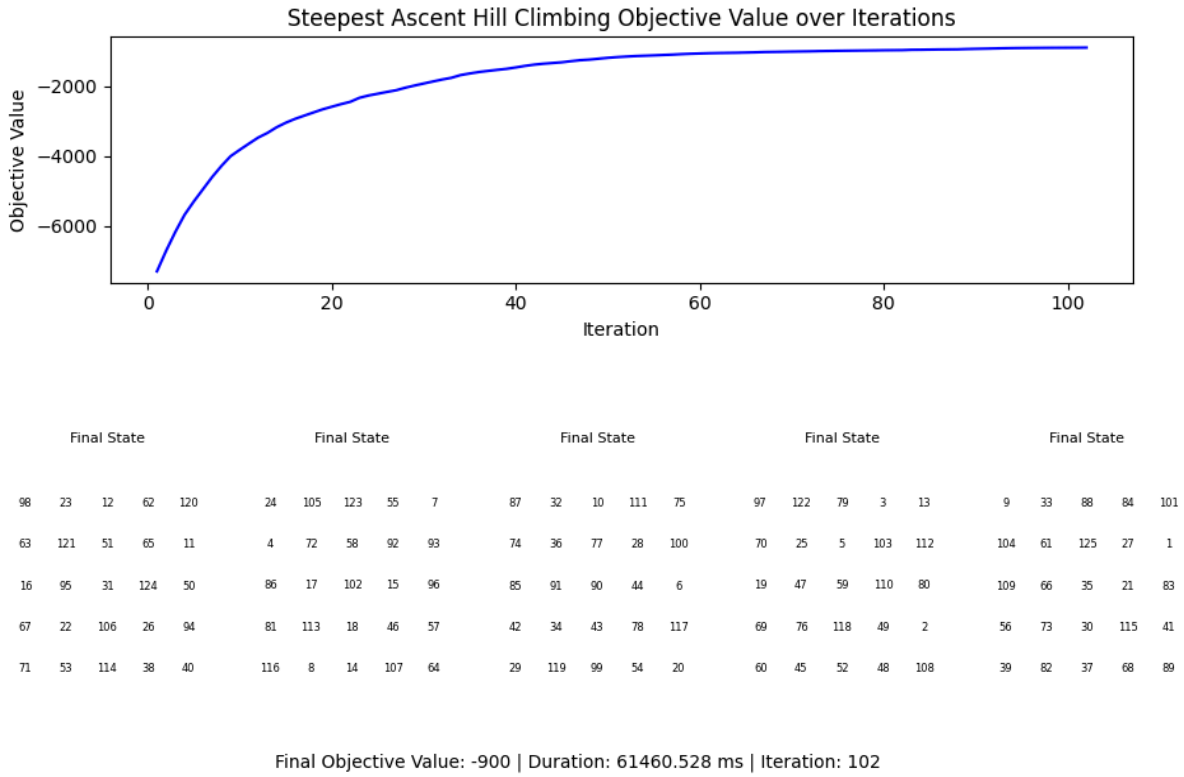


Gambar 2.3.1.1.2. Hasil dari Test Case 2 untuk Algoritma Steepest Ascent Hill Climbing

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





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

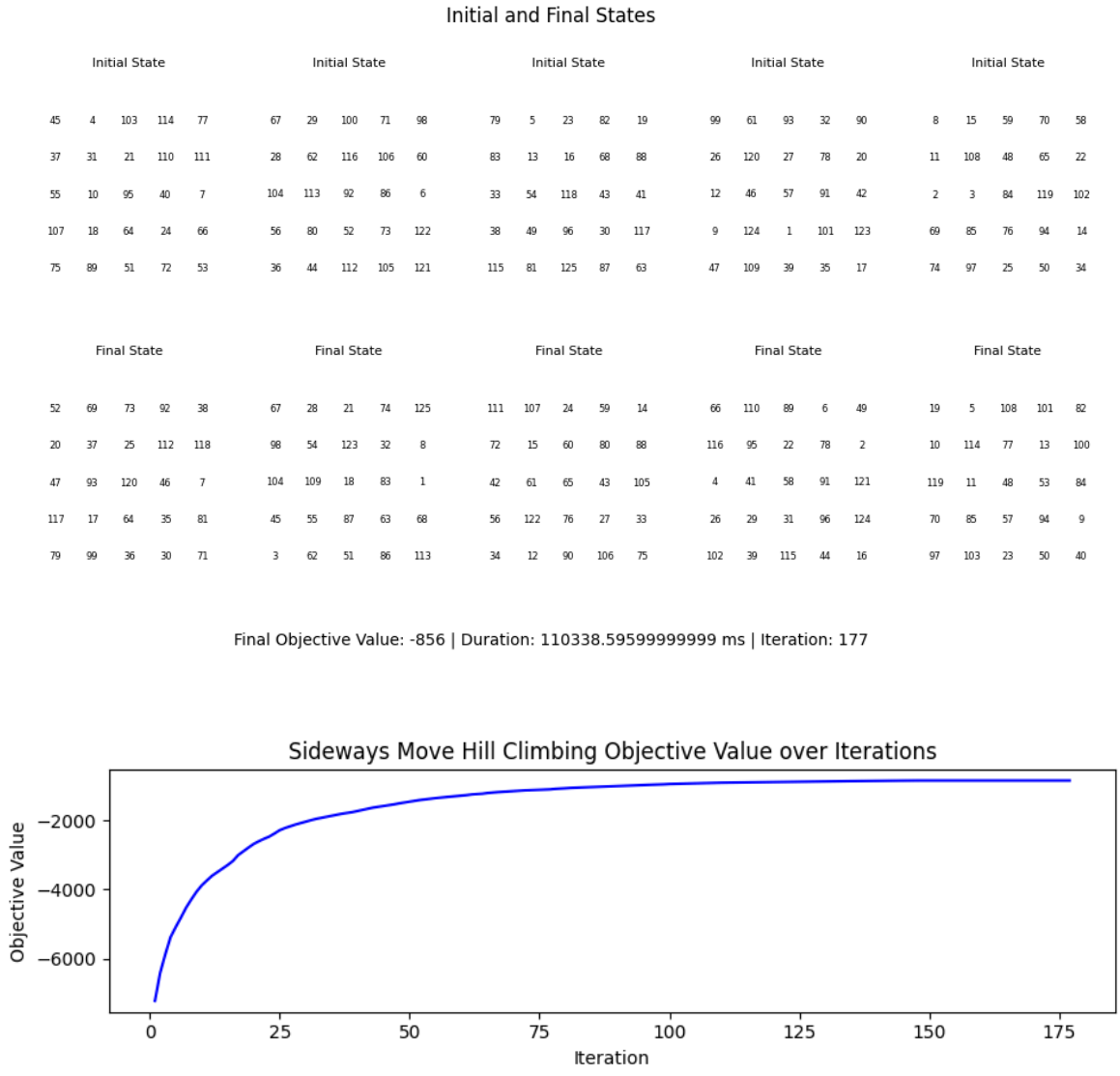
Algoritma ini bergantung pada jumlah iterasi dalam mencari hasil yang paling optimal. Dibandingkan dengan algoritma lain pada laporan ini, algoritma *steepest ascent hill climbing* memiliki cara kerja yang lebih terus terang. Algoritma ini akan terus membandingkan dirinya dengan *neighbor* terbaik dan 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 dibilang 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)

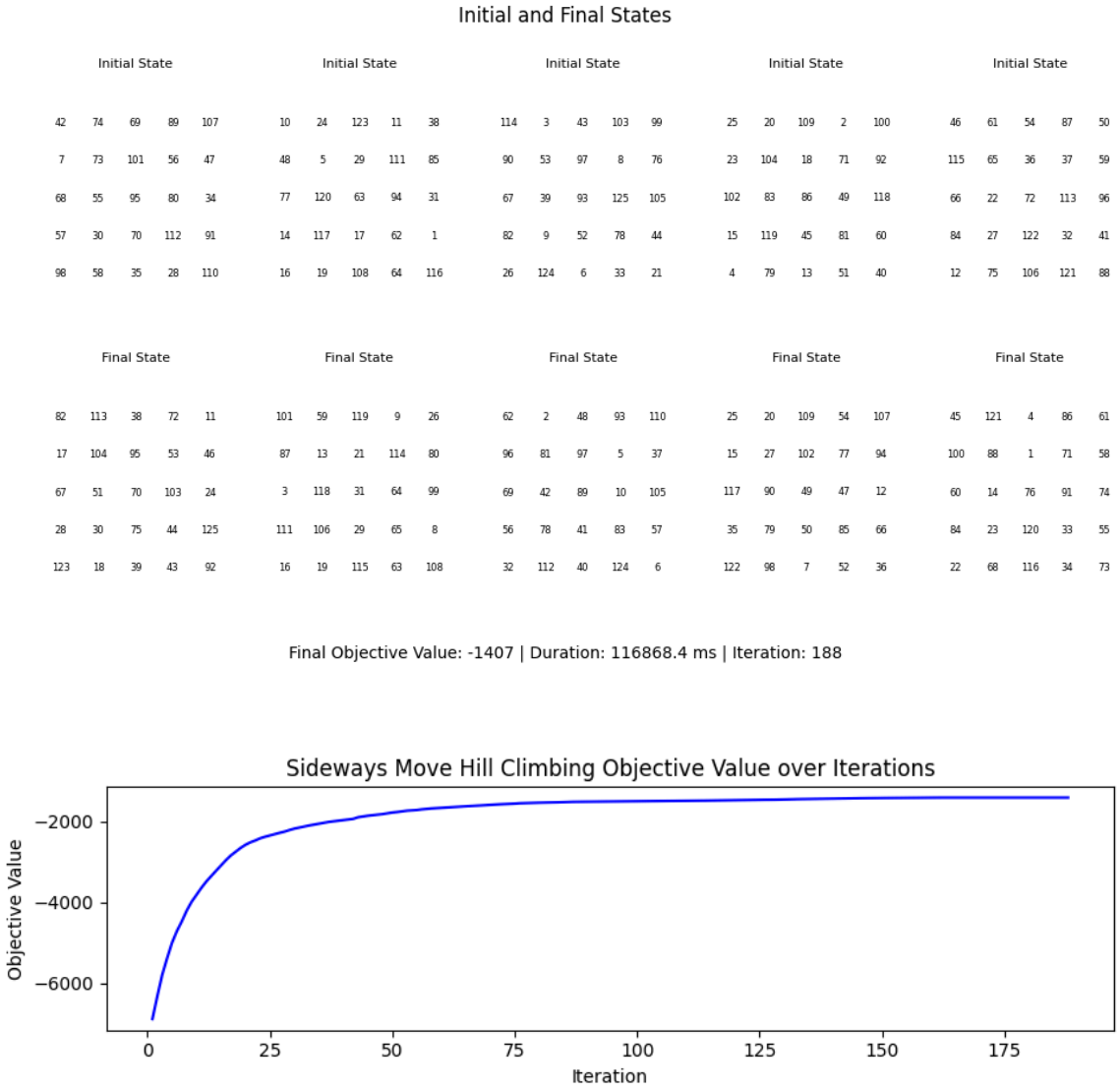


Gambar 2.3.1.2.1. Hasil dari Test Case 1 untuk Metode Hill Climbing With Sideways Move

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.

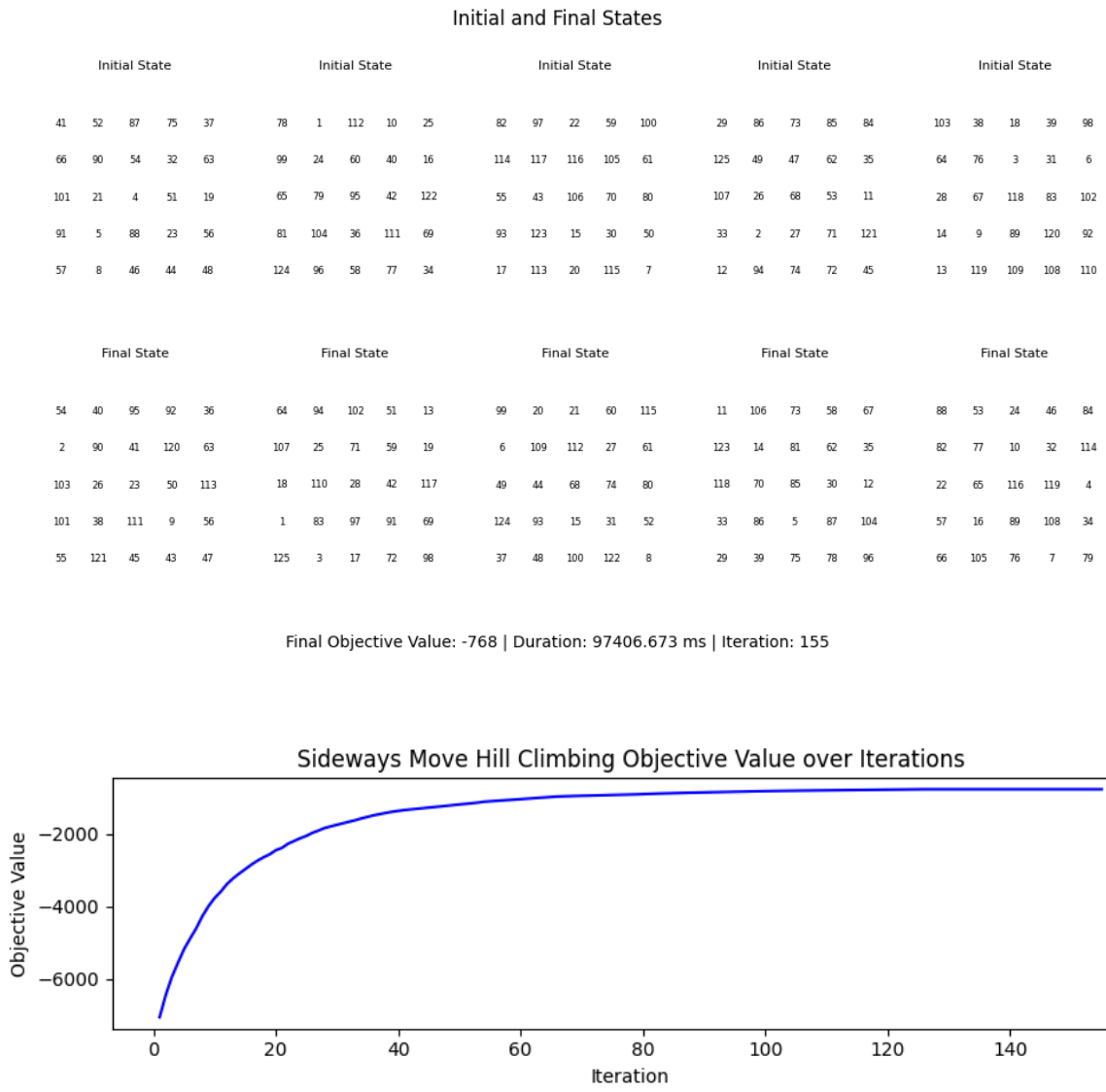
Test Case 2 (max-sideways = 20)



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)



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

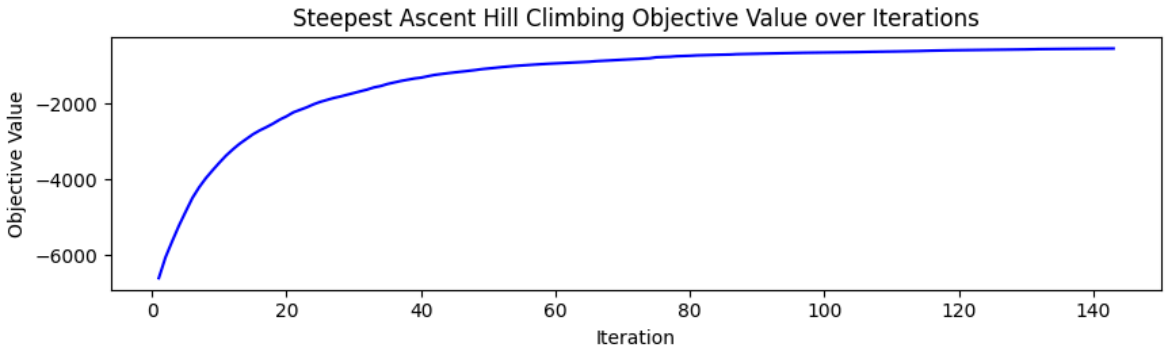
Initial State				
5	58	68	22	12
110	125	27	117	19
76	87	96	15	92
111	6	30	3	37
48	21	35	4	95

Initial State				
23	67	107	53	18
39	120	115	89	72
118	65	9	102	1
82	71	24	104	121
32	34	113	46	84

Initial State				
49	56	57	83	47
74	62	122	38	64
33	73	99	79	17
26	66	36	52	103
45	7	78	119	77

Initial State				
106	25	61	80	88
41	54	14	81	93
11	114	85	20	108
101	94	42	86	75
51	124	123	112	59

Initial State				
63	10	31	90	70
100	60	98	105	28
116	43	50	91	44
2	29	69	55	13
97	8	40	16	109



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

Final State

41	57	113	54	49
56	27	38	91	103
118	115	44	35	4
70	83	9	88	65
30	34	111	46	94

Final State

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

Final State

99	25	40	80	72
39	53	108	12	102
21	14	96	43	125
105	82	48	68	10
51	124	23	112	5

Final State

97	100	24	8	86
31	47	95	110	32
114	18	2	106	76
11	69	123	85	20
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

29	107	22	26	113
19	24	90	86	42
100	66	74	77	112
5	84	97	39	63
58	82	15	124	71

Initial State

99	48	1	122	67
20	12	57	4	23
81	38	75	116	96
37	121	117	2	118
7	95	36	16	62

Initial State

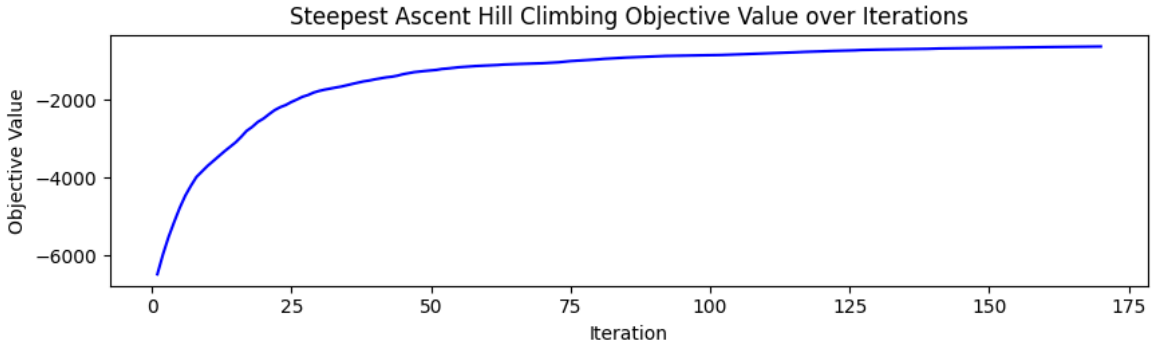
83	44	6	46	21
30	76	10	103	98
78	41	89	27	50
109	108	87	56	70
8	13	105	91	28

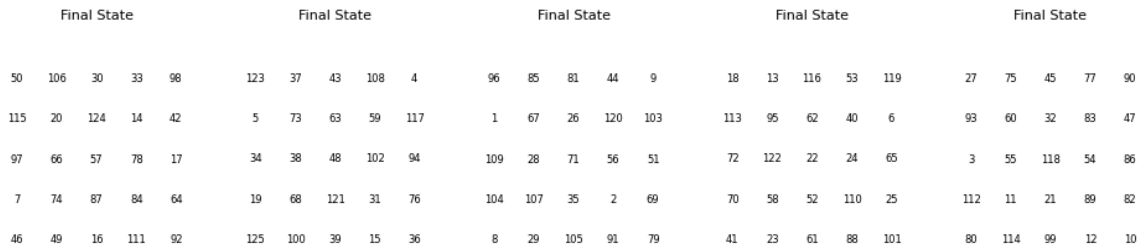
Initial State

123	45	92	51	110
102	119	33	35	115
65	17	64	120	61
68	47	53	104	40
3	9	72	125	101

Initial State

73	69	49	32	88
93	60	25	106	34
54	11	79	59	80
111	14	31	43	52
114	85	94	55	18

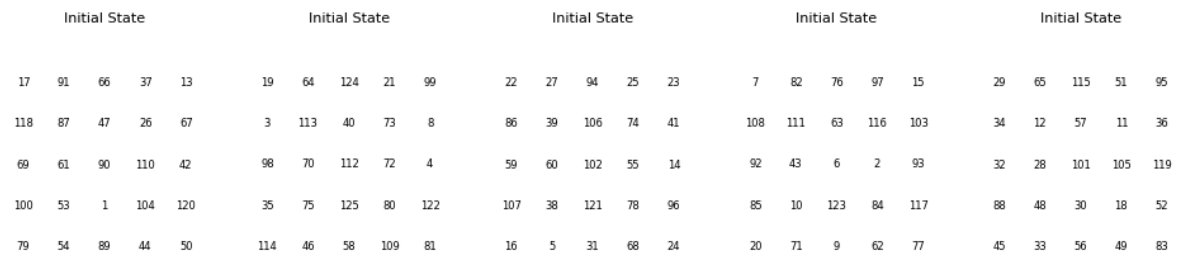


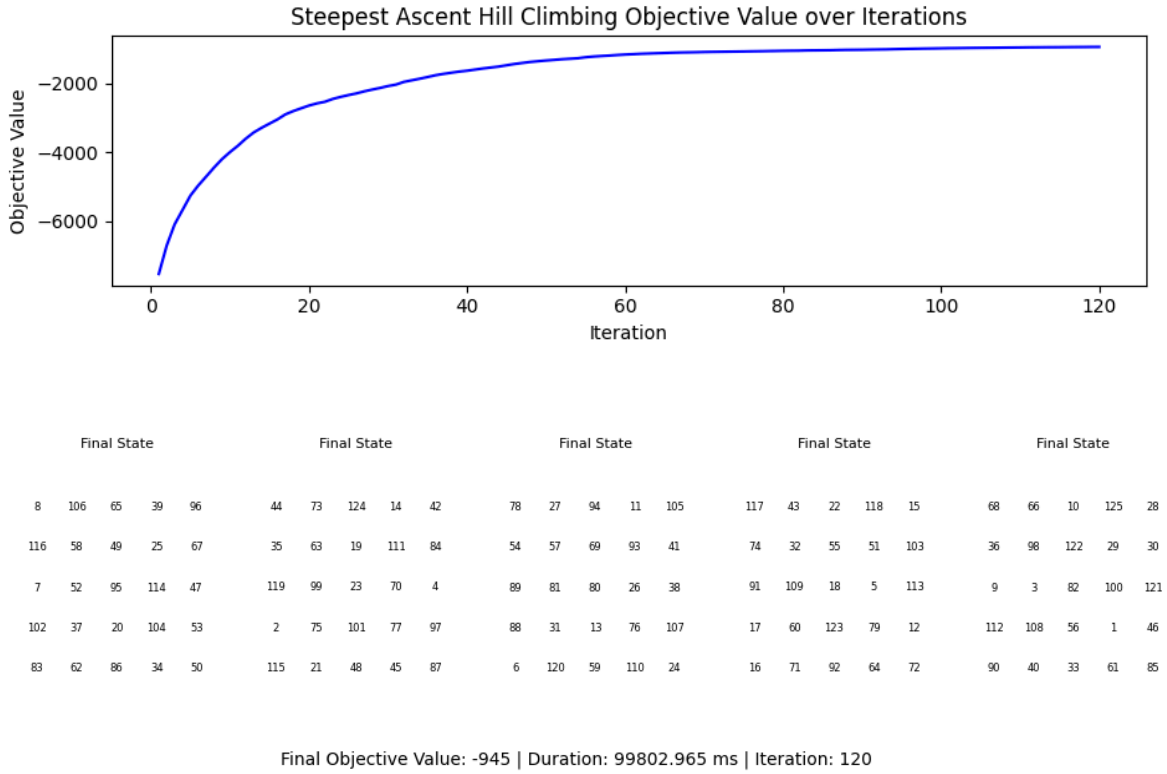


Final Objective Value: -624 | Duration: 152920.17200000002 ms | Iteration: 170

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

Restart 3





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
```

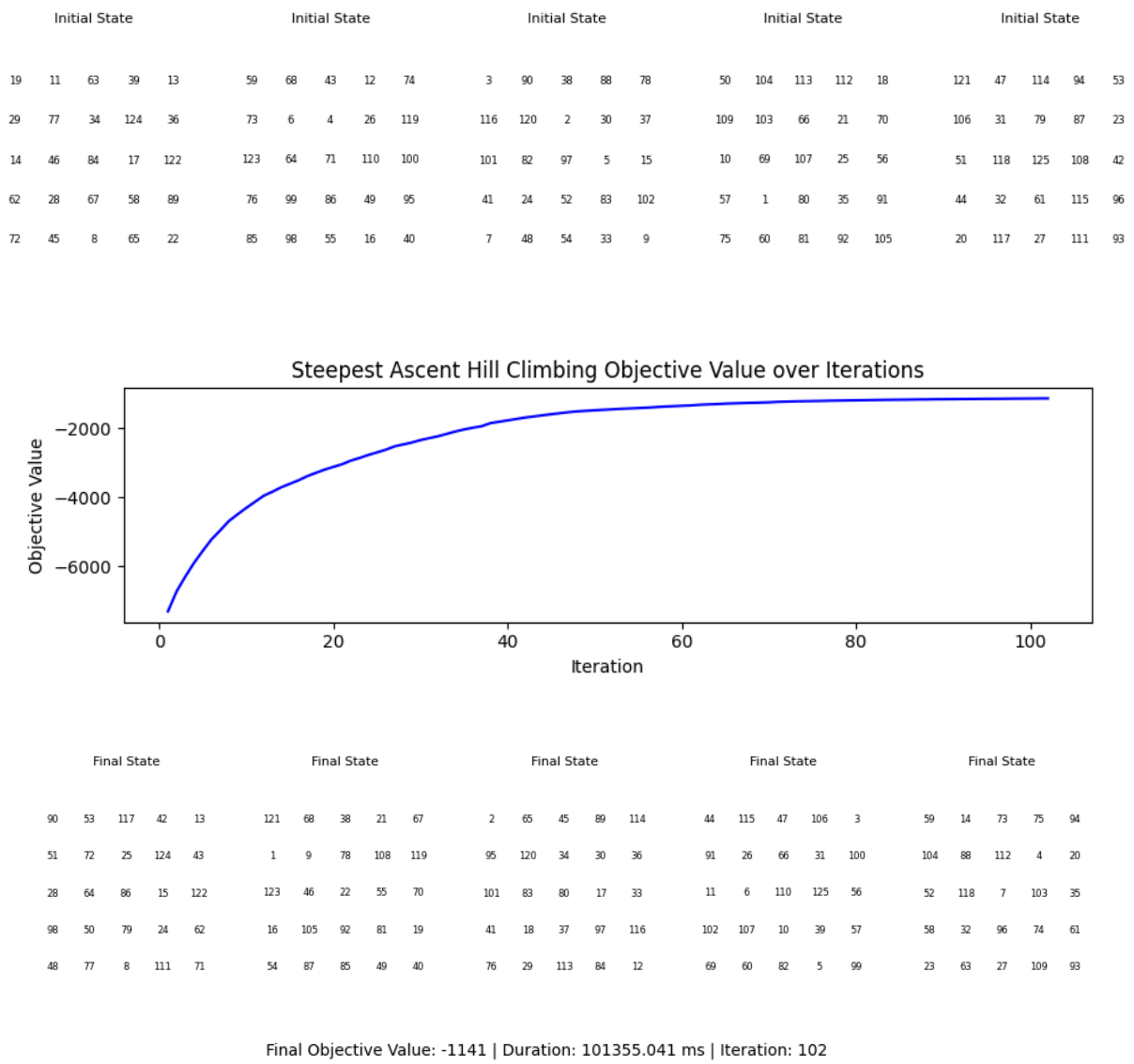
Gambar 2.3.1.3.4. Hasil Akhir pada Test Case 1 untuk Algoritma Random Restart Hill Climbing

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 .

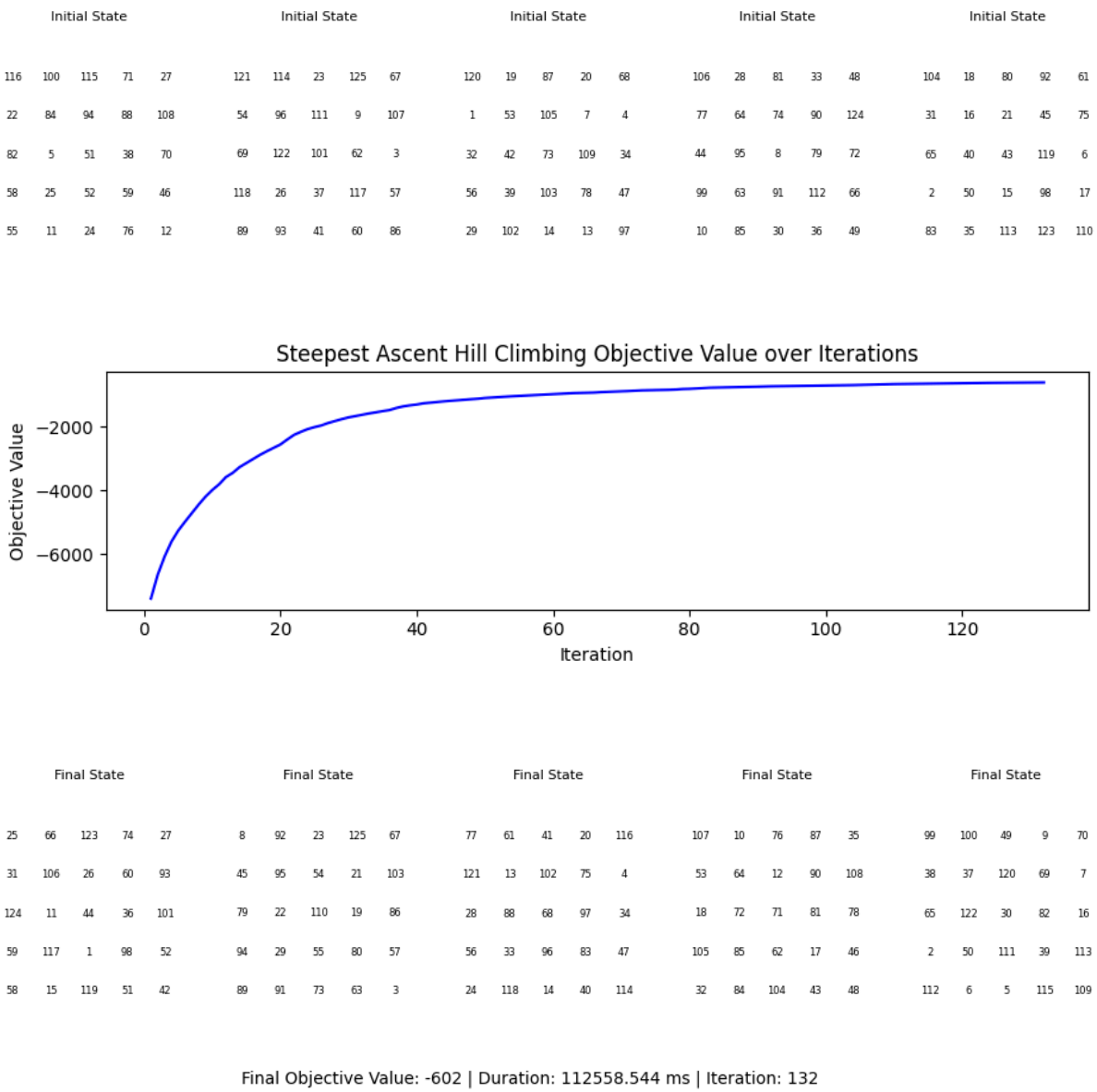
Test Case 2 (Max Restart = 4)

Restart 1



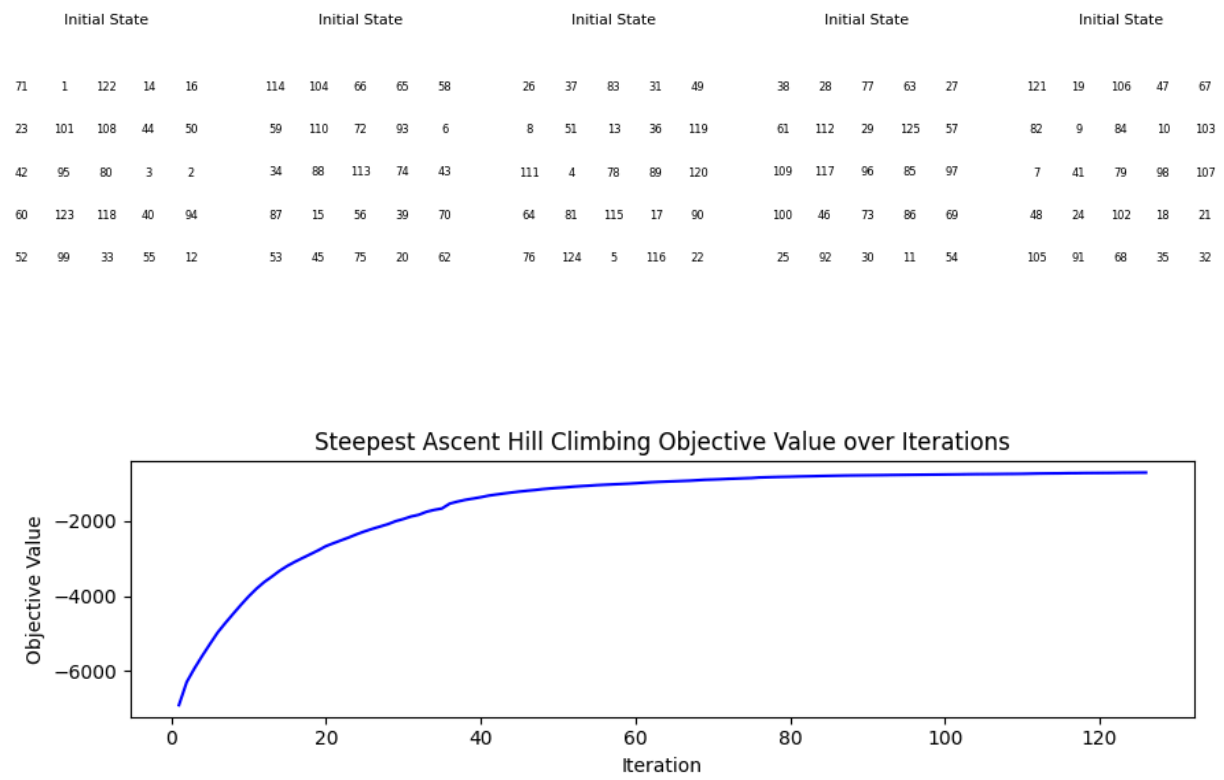
Gambar 2.3.1.3.5. Hasil dari test case 2 pada restart 1 untuk algoritma Random Restart Hill Climbing.

Restart 2



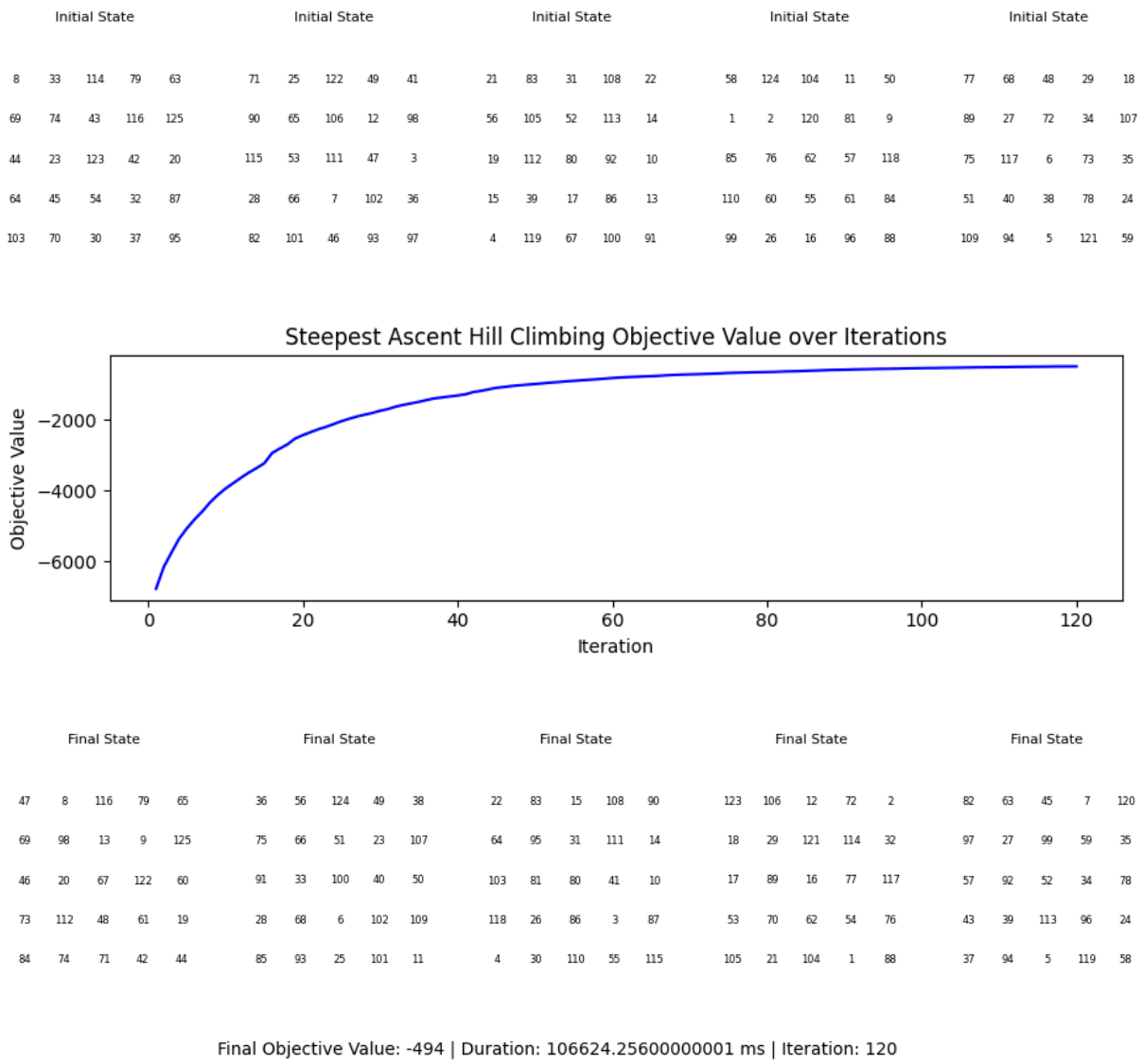
Gambar 2.3.1.3.6. Hasil dari test case 2 pada restart 2 untuk algoritma Random Restart Hill Climbing.

Restart 3



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

Restart 4



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

97	69	41	52	6
28	98	78	107	26
118	31	95	108	25
44	91	94	70	43
14	16	63	13	39

Initial State

110	45	34	55	88
90	1	23	117	113
100	4	27	20	64
85	120	121	99	8
22	123	29	119	15

Initial State

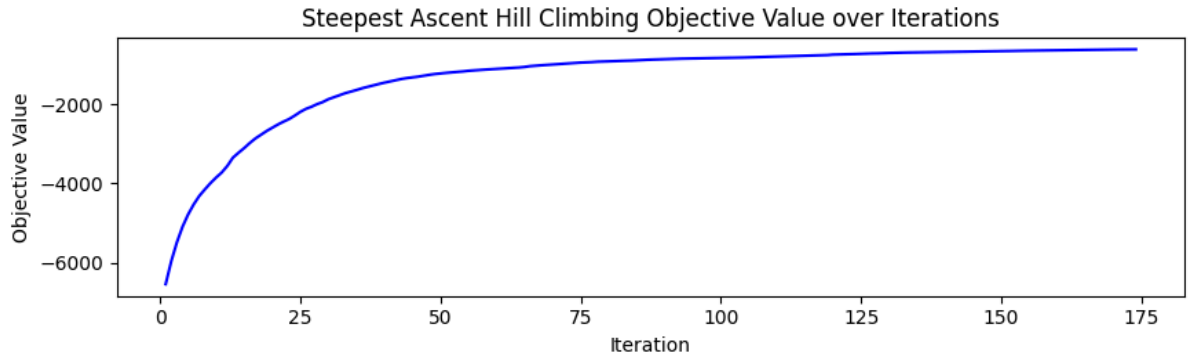
86	47	5	21	59
40	96	2	74	19
51	72	105	87	103
75	77	42	92	112
68	38	57	73	37

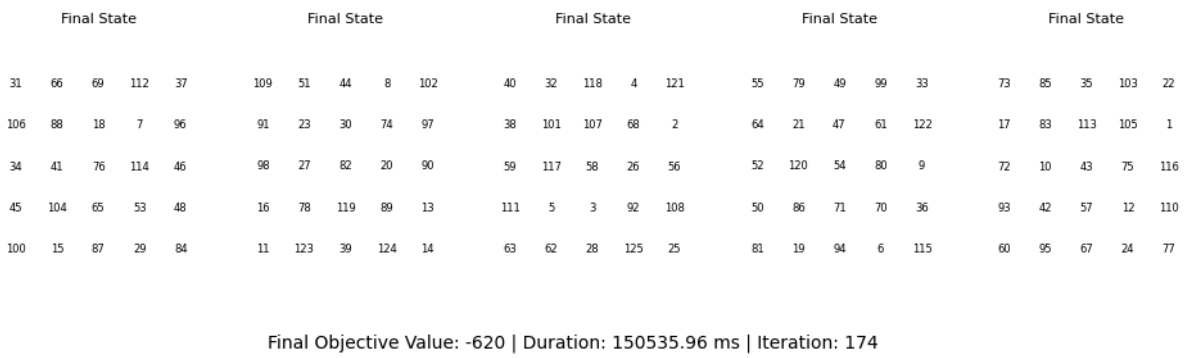
Initial State

54	80	49	124	11
62	24	122	60	10
84	56	116	79	7
50	89	71	32	58
93	18	81	65	115

Initial State

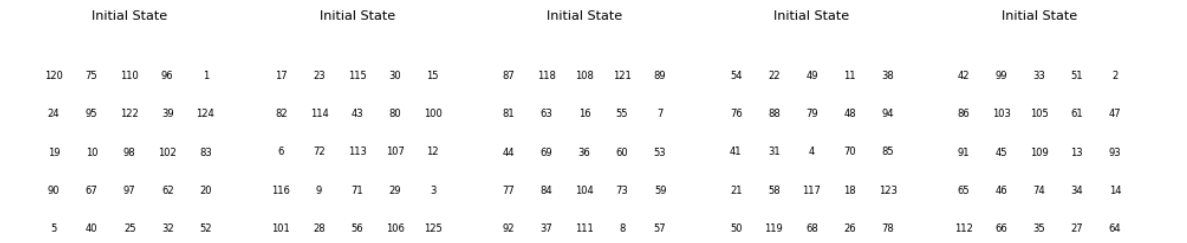
66	82	46	106	12
30	83	104	61	17
76	53	33	36	109
102	48	67	125	114
3	101	9	35	111

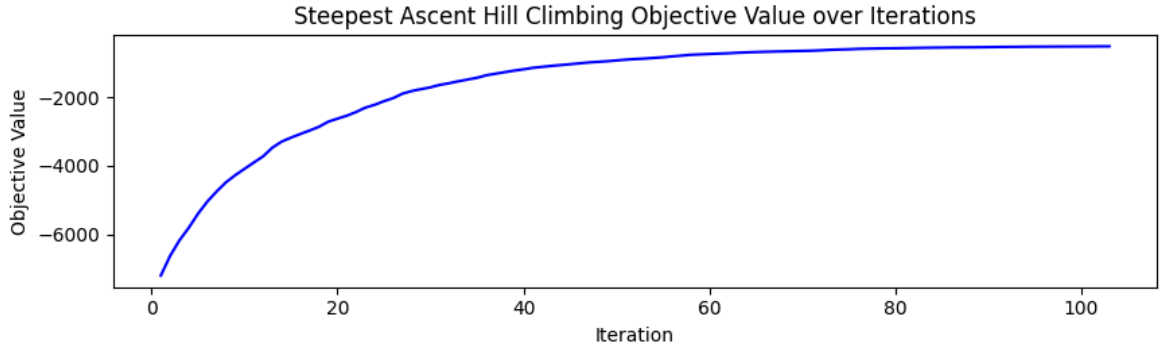


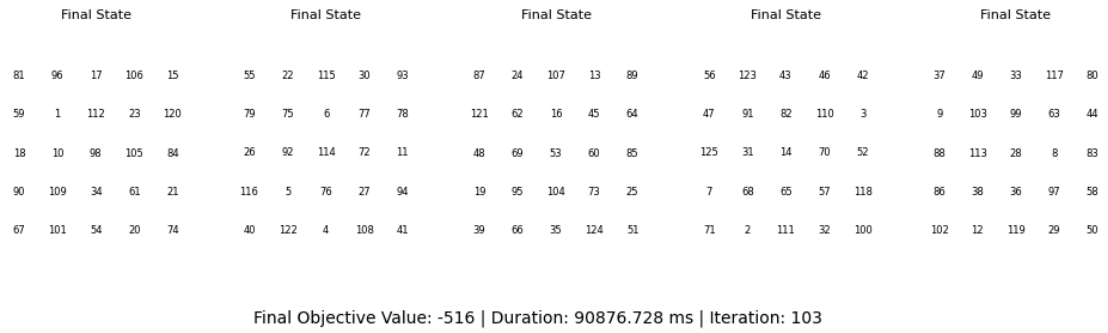


Gambar 2.3.1.3.10. Hasil dari test case 3 pada restart 1 untuk algoritma Random Restart Hill Climbing.

Restart 2

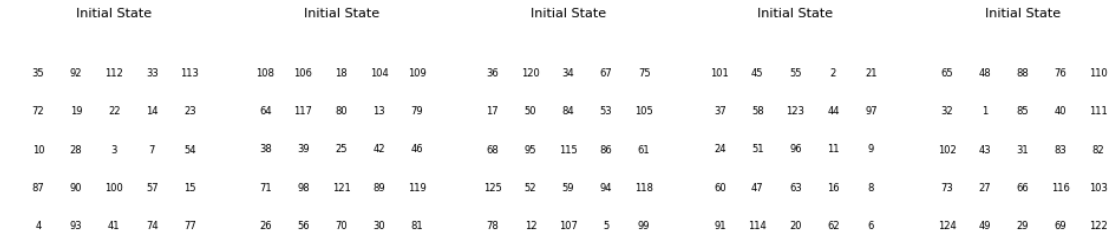


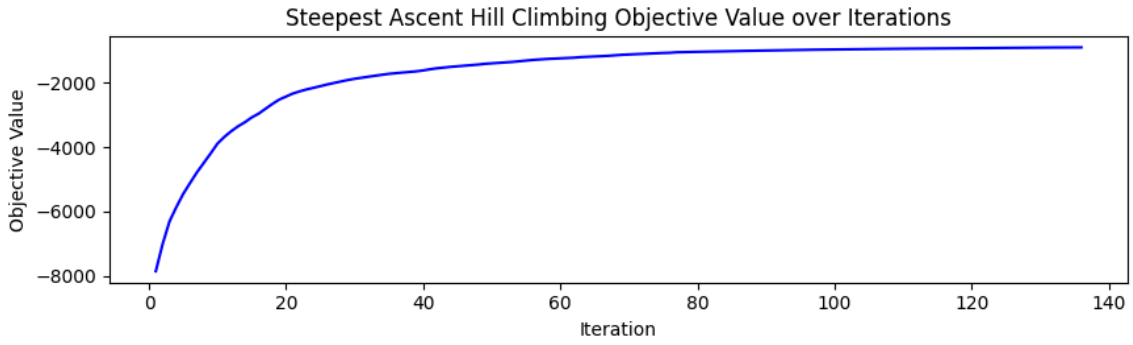




Gambar 2.3.1.3.11. Hasil dari test case 3 pada restart 2 untuk algoritma Random Restart Hill Climbing.

Restart 3





Final State

59	18	118	66	55
74	82	22	122	17
112	28	40	21	114
62	90	99	4	60
8	97	35	102	69

Final State

100	14	20	115	61
56	107	37	45	72
51	30	95	42	98
75	96	44	88	12
33	68	119	25	67

Final State

3	121	34	81	76
23	50	84	53	106
73	85	57	86	11
125	52	36	93	9
89	7	104	2	113

Final State

110	77	31	91	6
41	48	78	39	109
16	38	111	87	63
47	58	71	15	123
101	94	24	83	13

Final State

46	43	108	1	117
120	70	92	26	10
64	124	19	79	29
5	27	65	116	105
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

43	51	110	47	20
40	83	123	65	79
19	24	1	109	108
105	48	59	124	22
88	18	50	17	15

Initial State

35	55	103	122	125
54	77	98	117	26
63	68	101	41	4
115	31	8	21	69
5	87	49	81	100

Initial State

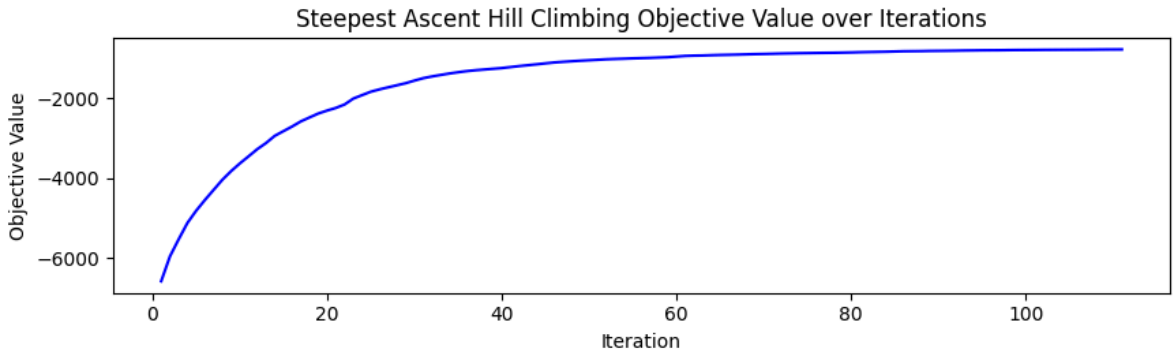
62	61	67	107	57
42	66	94	39	46
118	96	29	99	76
92	114	14	2	72
112	45	85	44	11

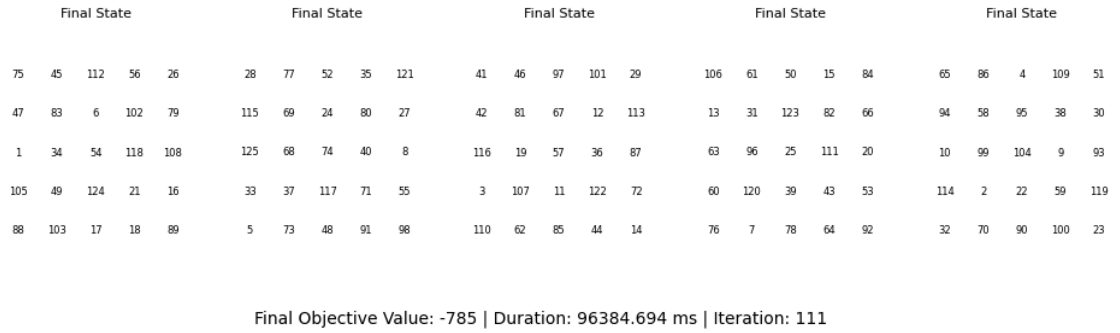
Initial State

106	56	23	121	91
12	34	7	80	74
37	84	73	111	3
58	38	6	32	60
78	9	75	64	28

Initial State

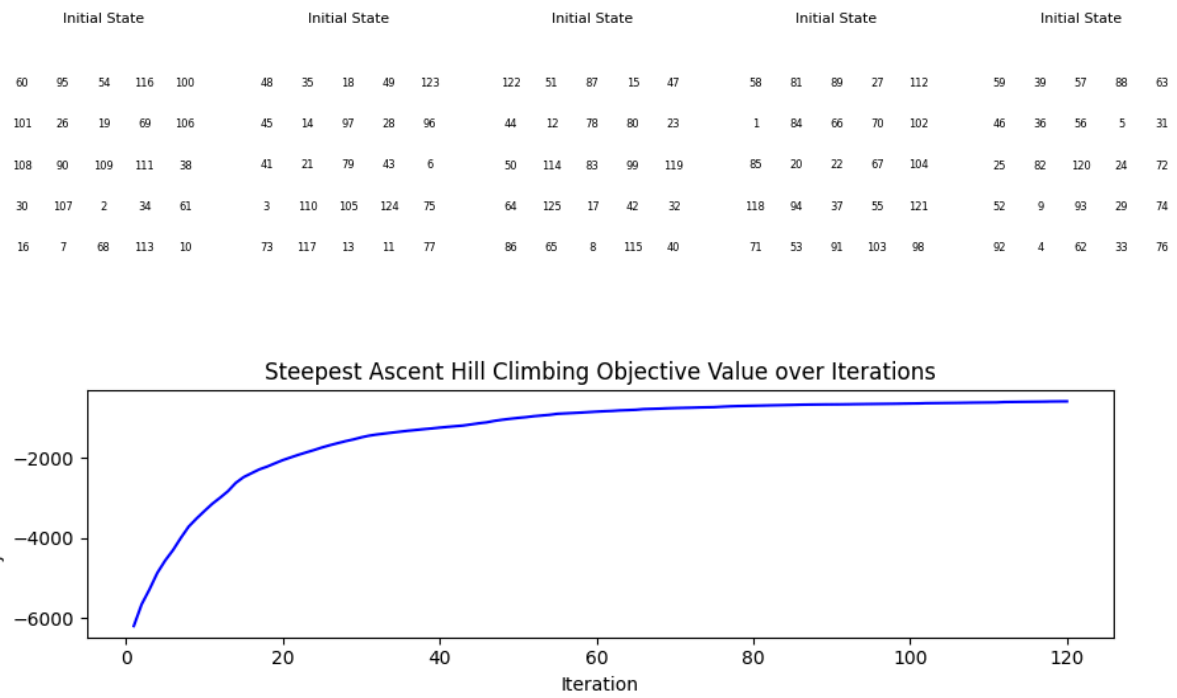
104	86	82	113	97
90	53	93	36	30
27	120	95	10	102
119	33	16	71	116
13	70	89	52	25

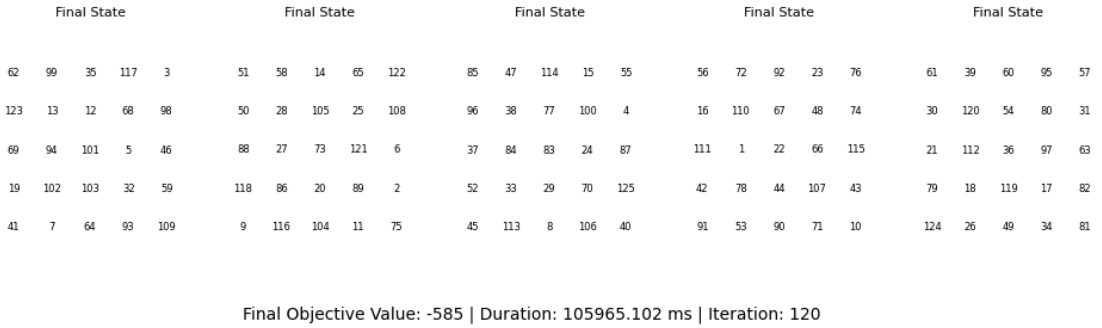




Gambar 2.3.1.3.13. Hasil dari test case 3 pada restart 4 untuk algoritma Random Restart Hill Climbing.

Restart 5





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, 12
4, 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, 1
00, 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

121	105	9	53	97
78	30	111	52	17
3	92	83	112	41
39	107	68	93	99
85	49	106	54	25

Initial State

108	95	28	124	12
123	18	118	45	73
34	26	65	82	113
7	8	104	80	35
119	50	58	15	75

Initial State

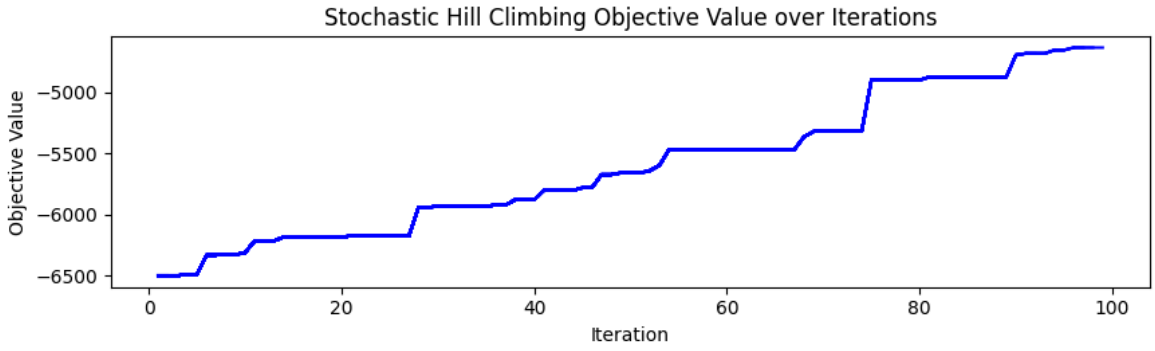
81	56	77	86	43
67	16	36	21	89
116	57	103	31	102
20	87	125	24	90
59	109	84	63	44

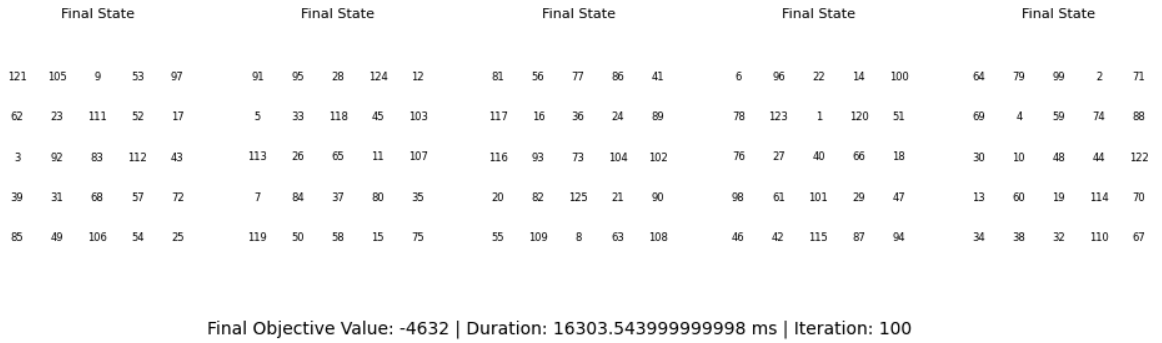
Initial State

6	96	22	14	100
62	11	71	120	51
76	27	69	66	33
98	61	101	29	47
46	42	115	1	94

Initial State

64	79	110	2	37
48	4	55	74	88
23	10	40	91	122
13	60	19	114	70
5	38	32	72	117

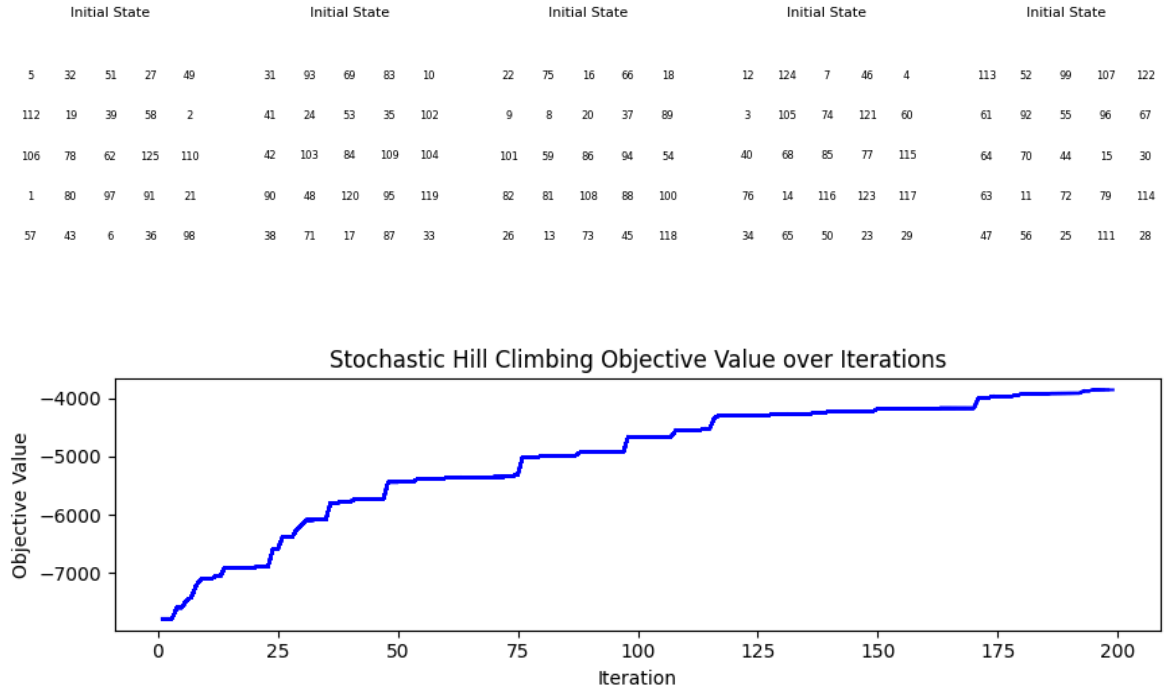


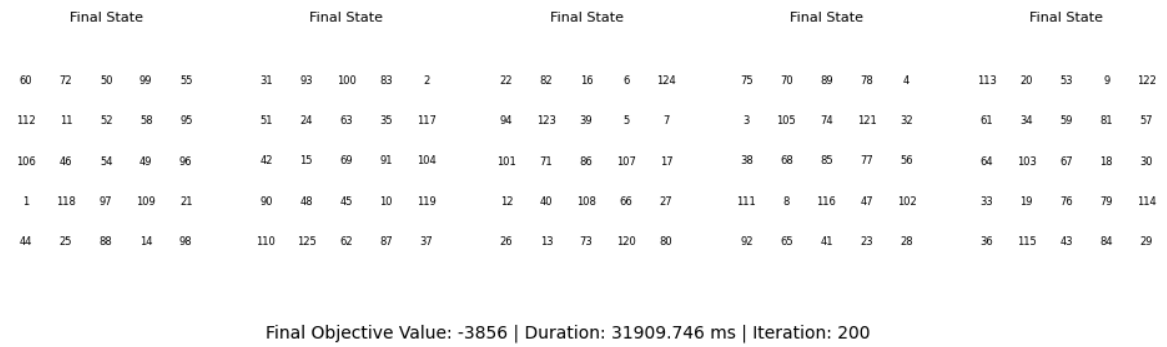


Gambar 2.3.1.4.1. Hasil dari test case 1 untuk algoritma Stochastic Hill Climbing.

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

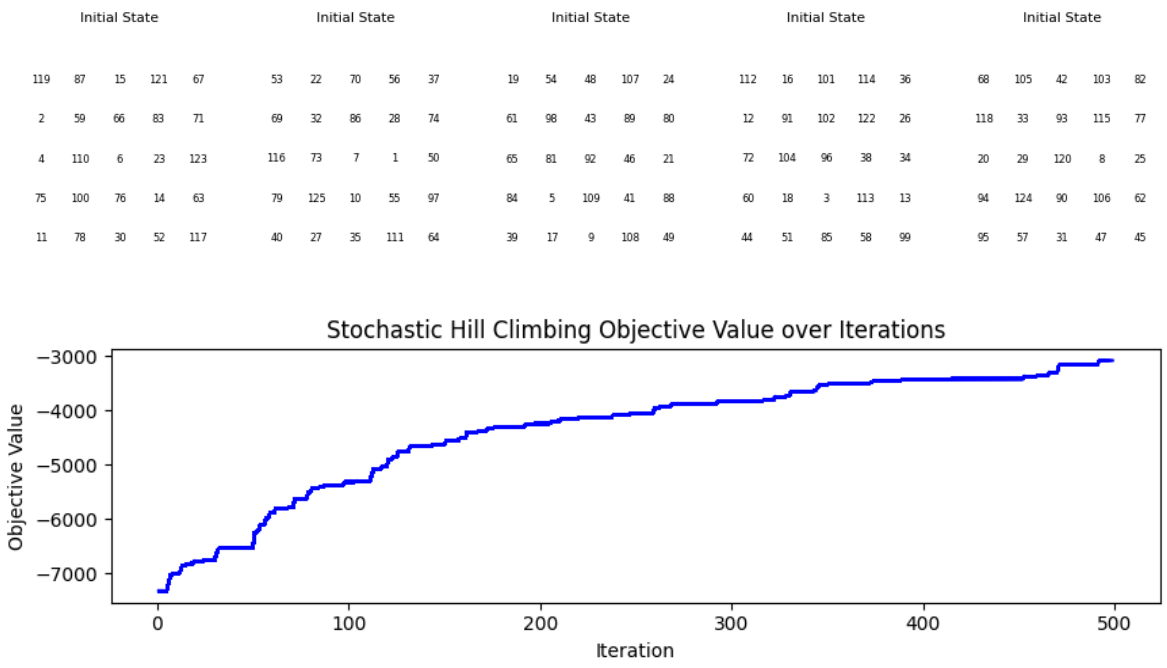


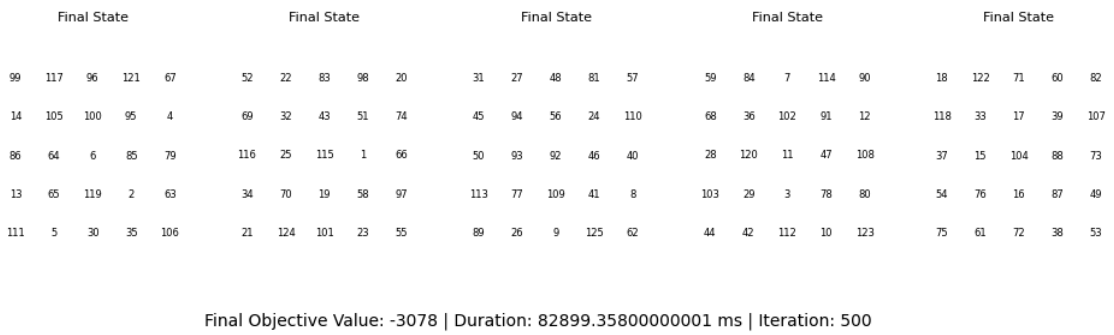


Gambar 2.3.1.4.2. Hasil dari test case 2 untuk algoritma Stochastic Hill Climbing.

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





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

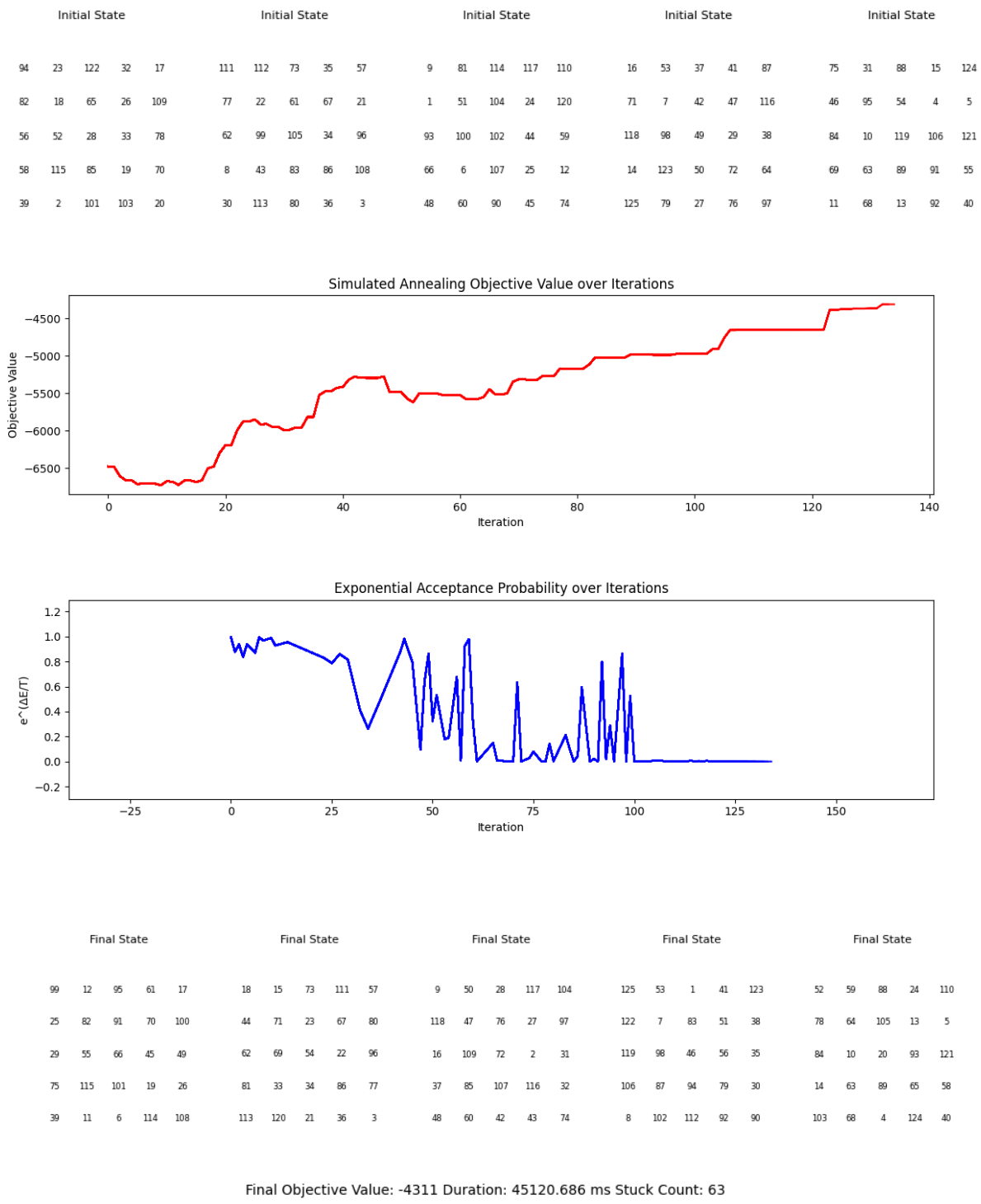
Pencarian *stochastic hill climbing* lebih berfokus pada keringanan iterasi (yang dapat disesuaikan) , namun sangat bergantung dengan jumlah iterasi. Iterasi yang semakin banyak memberikan hasil yang lebih optimal.

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)

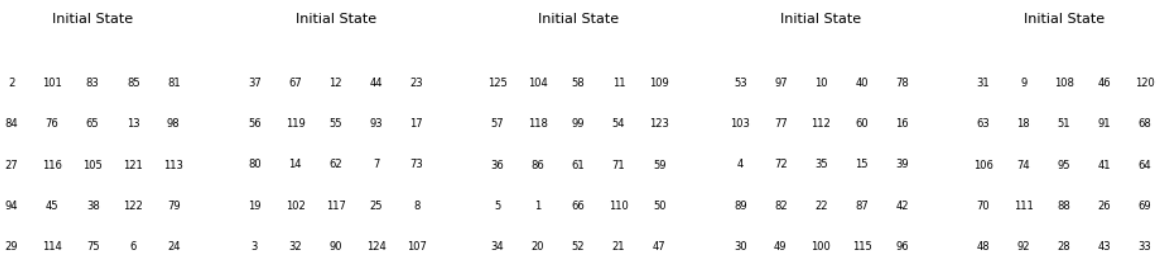


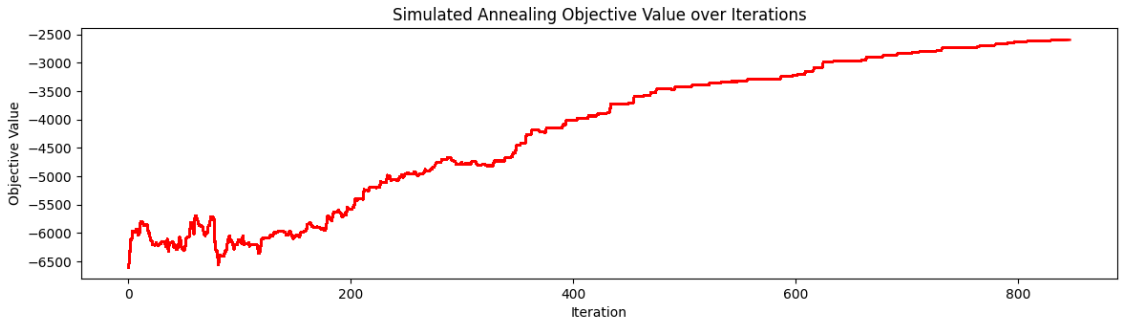
Gambar 2.3.2.1. Hasil dari test case 1 untuk algoritma Simulated Annealing.

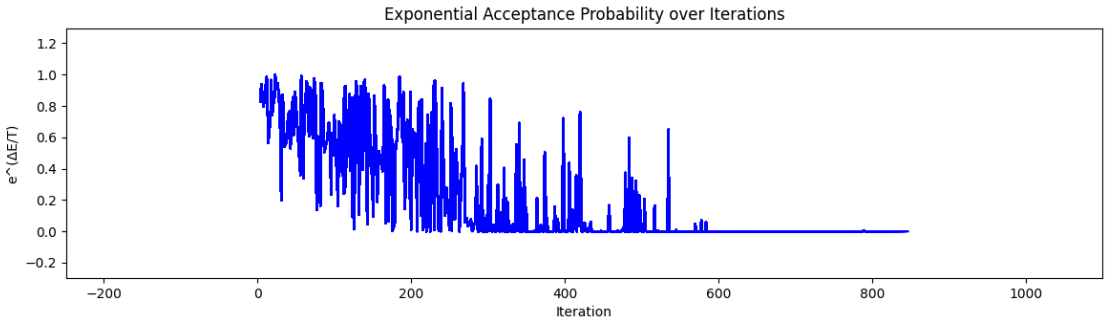
Dari hasil eksperimen pertama yang dilakukan dengan menggunakan algoritma Simulated Annealing, didapatkan bahwa hasil dari pencarian yang dilakukan mendapatkan final state seperti yang terlampir dari final state yang terlampir. Nilai objektif dari final state yang

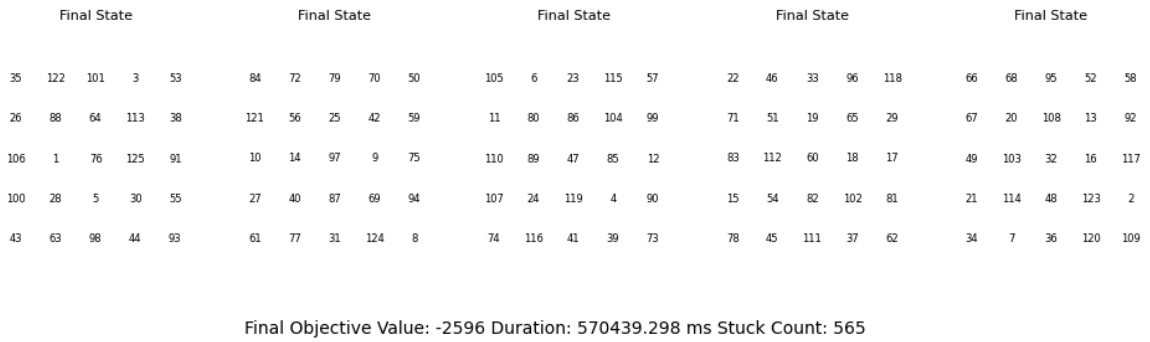
diperoleh bernilai -4311, dengan durasi eksekusi selama kurang lebih 46 detik. Kemudian, untuk banyak kali ditemukan neighbor dengan nilai objektif yang sama, yaitu sebanyak 63 kali, namun algoritma berhasil melepaskan diri. Kemudian, dari grafik Exponential Acceptance Probability, juga dapat dilihat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

Test Case 2 (T0 = 500, CoolingRate = 0.99, T1 = 0.1)





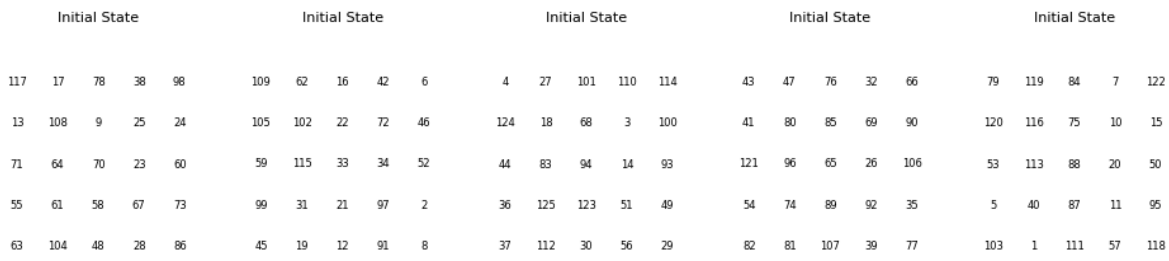


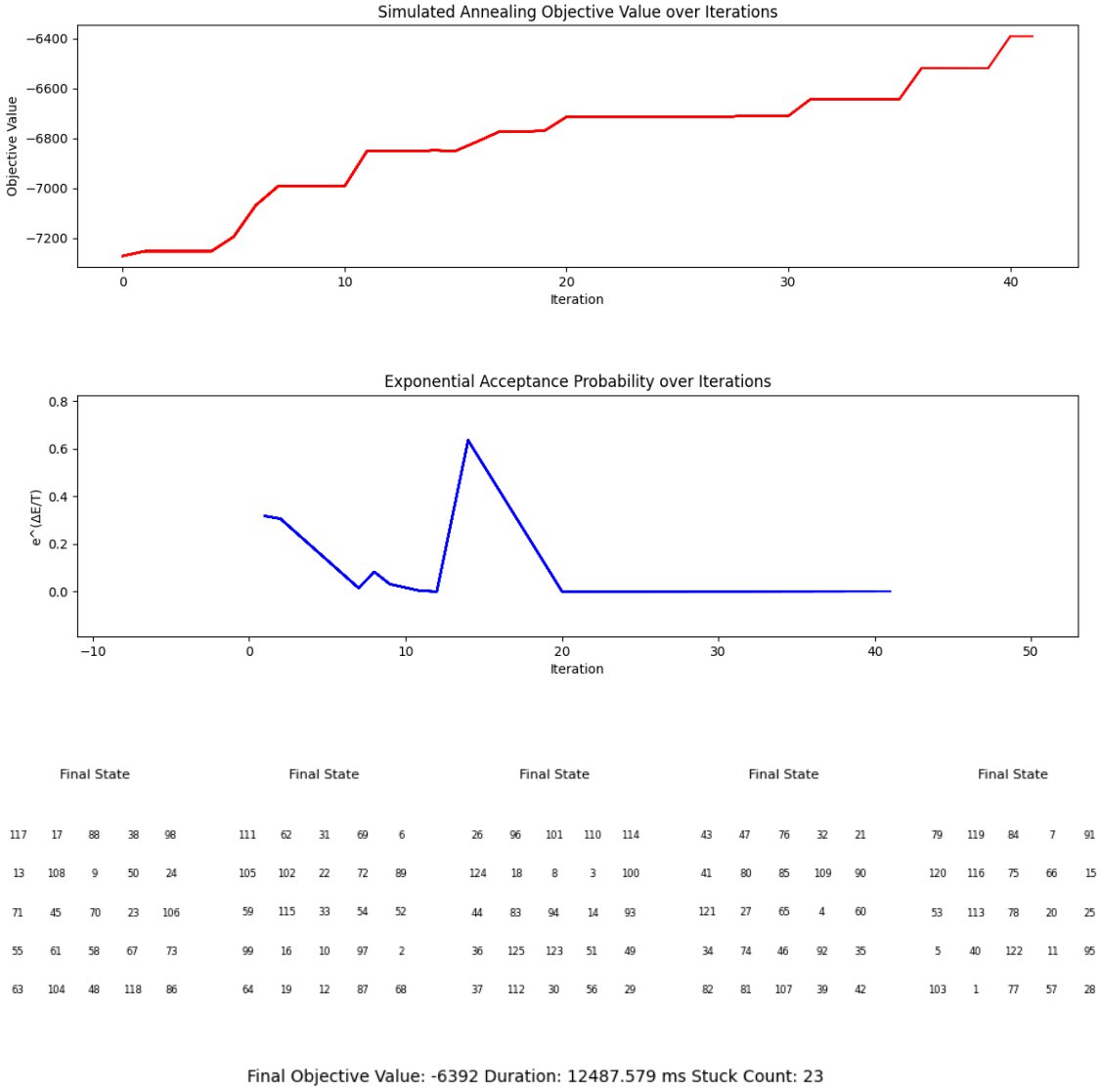


Gambar 2.3.2.1. Hasil dari test case 2 untuk algoritma Simulated Annealing.

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 (T0 = 100, CoolingRate = 0.8, T1 = 0.01)





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

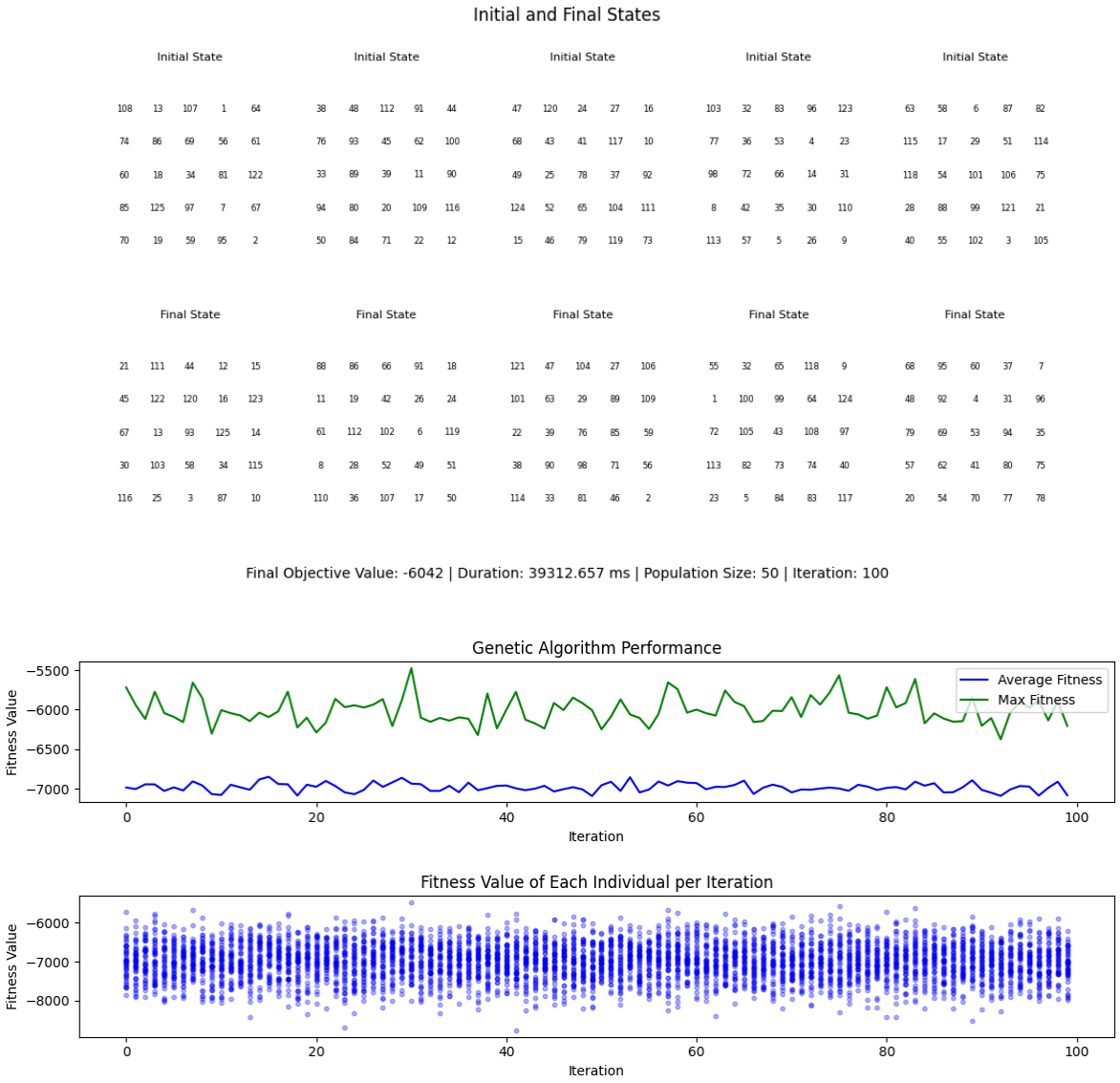
Dari hasil ketiga eksperimen yang telah dilakukan dengan menggunakan simulated annealing, dapat dilihat bahwa nilai objektif final state yang didapatkan bernilai -2596. Namun, hasil terbaik yang didapatkan melewati iterasi sebanyak 833 kali. Sedangkan, untuk yang terburuk adalah dengan nilai objektif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

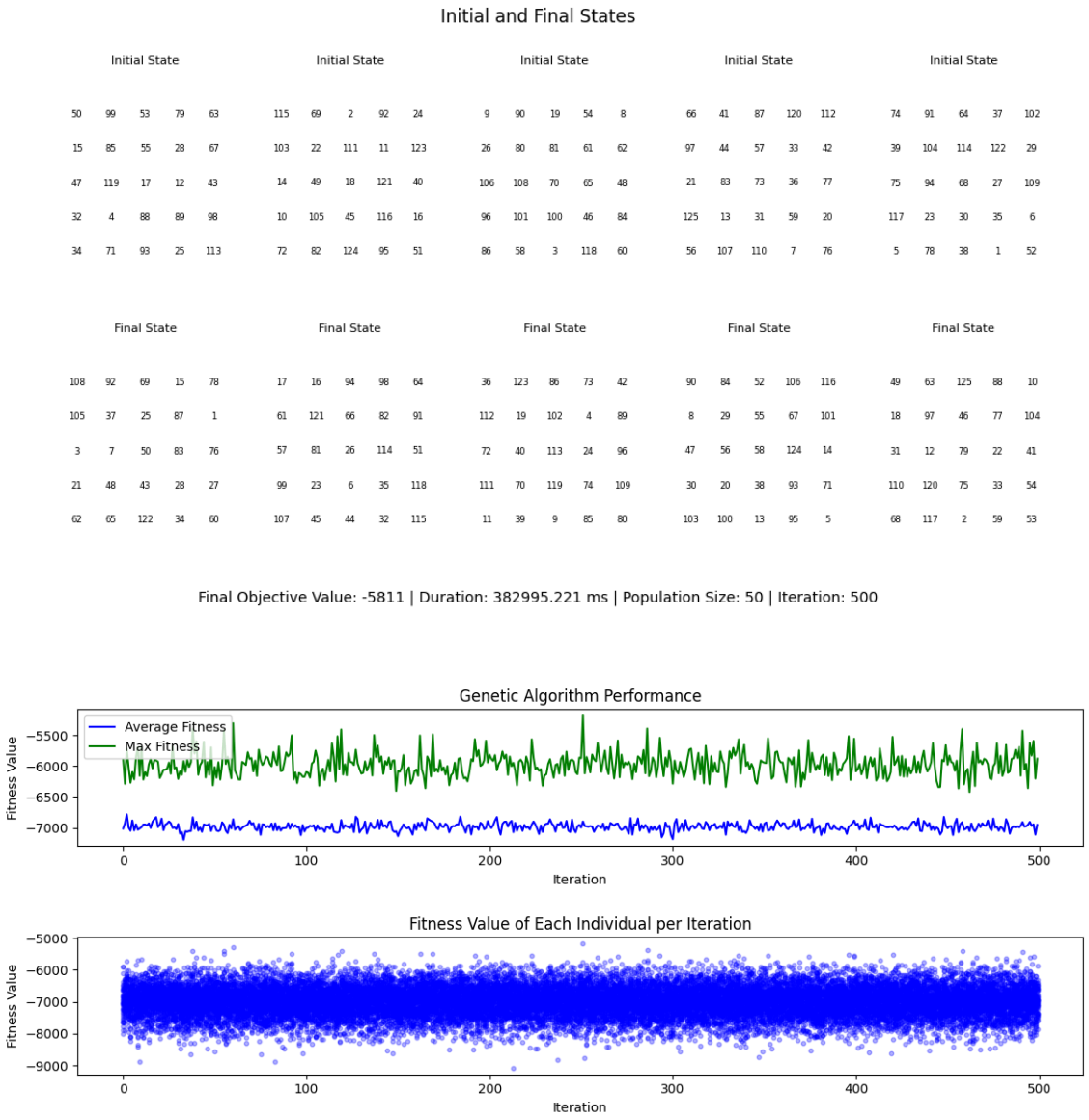
Dari eksperimen ini dilakukan iterasi sebanyak 100 kali dengan jumlah populasi sebanyak 50. Nilai objektif akhir yang didapat yakni adalah -6042 , dan memiliki durasi selama kurang lebih 39 detik.

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

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.

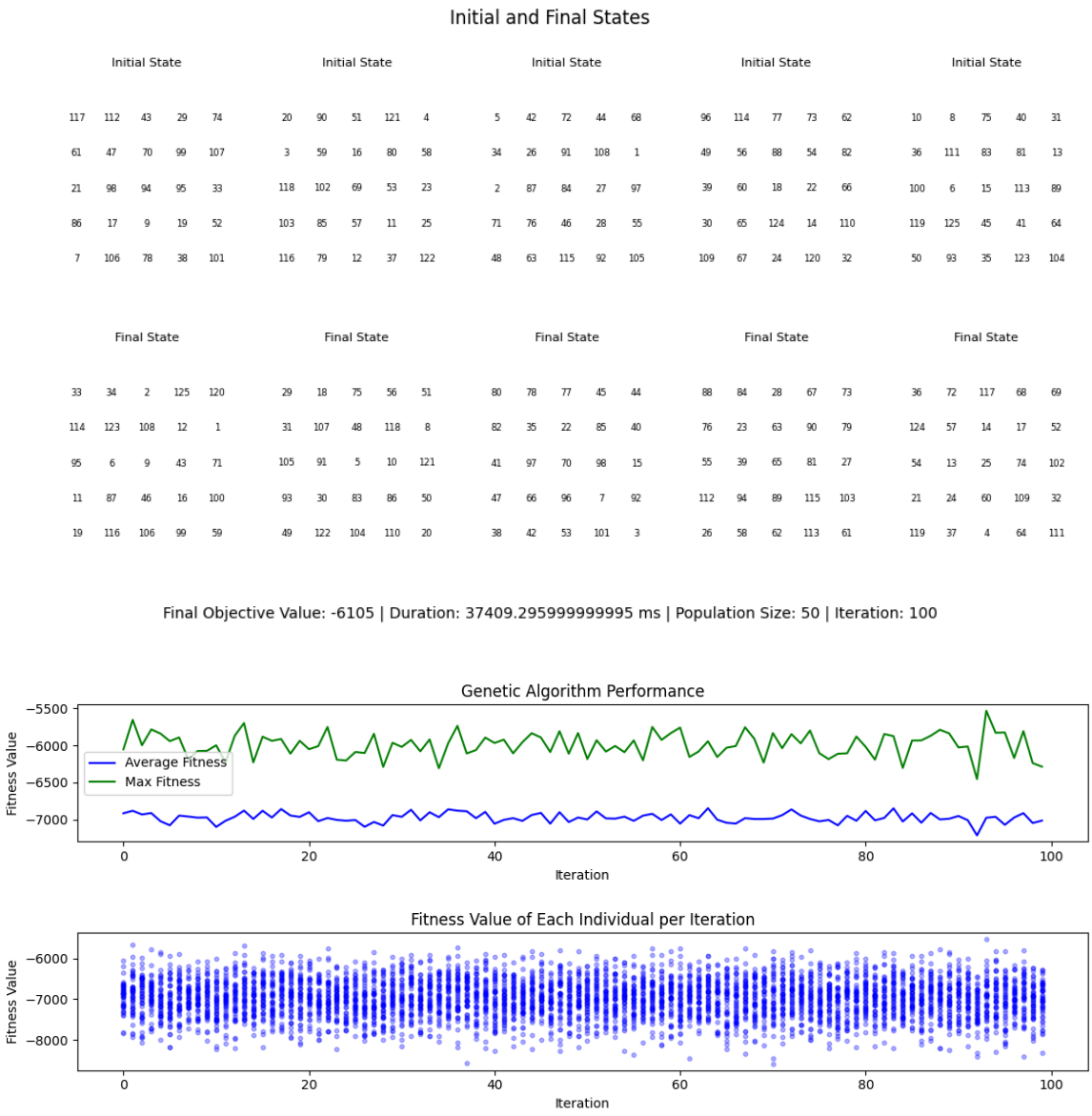
Test Case 3 (Untuk perubahan iterasi terhadap populasi yang sama)



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.

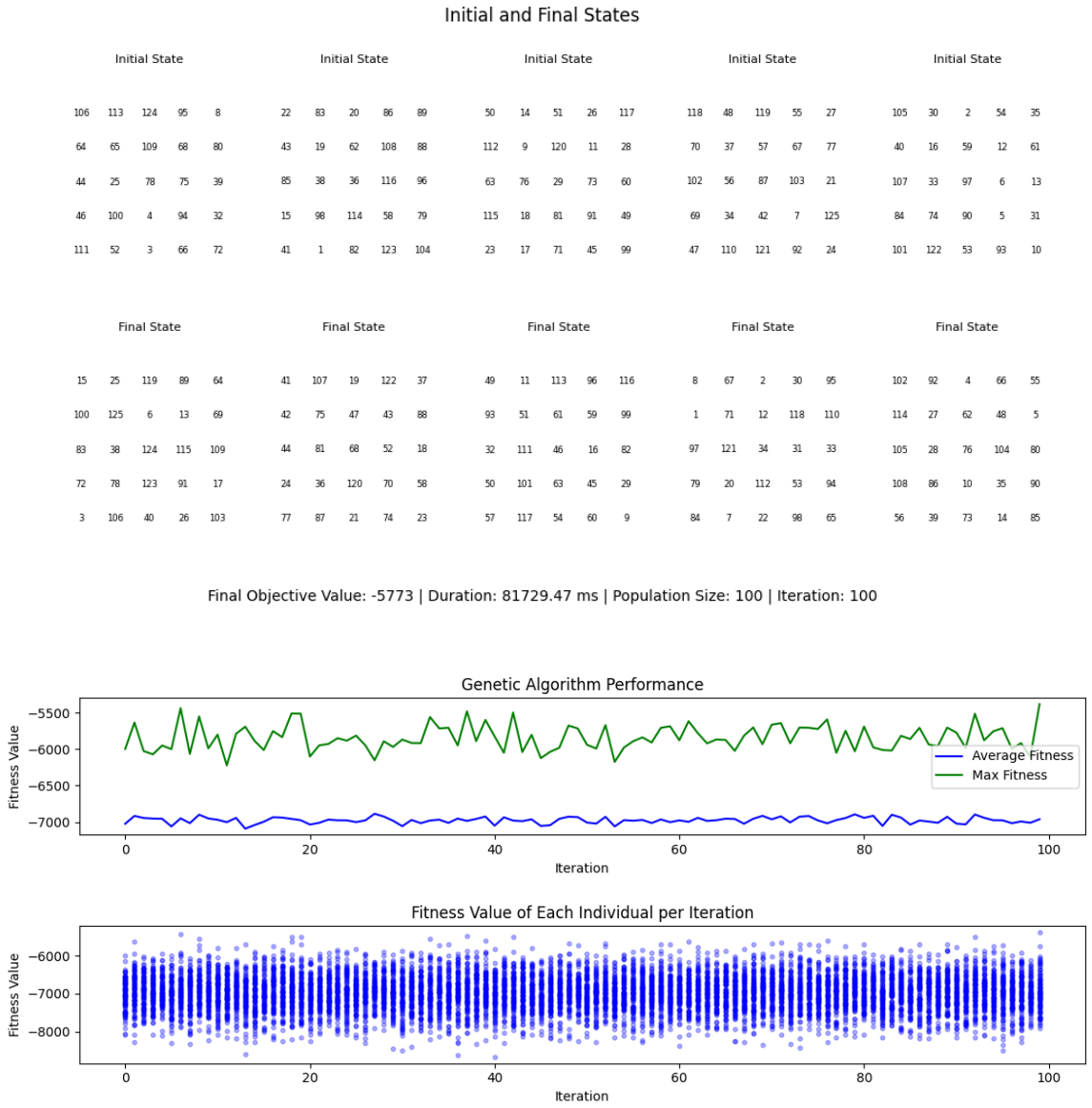
Test Case 4 (Untuk perubahan populasi terhadap iterasi yang sama)



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.

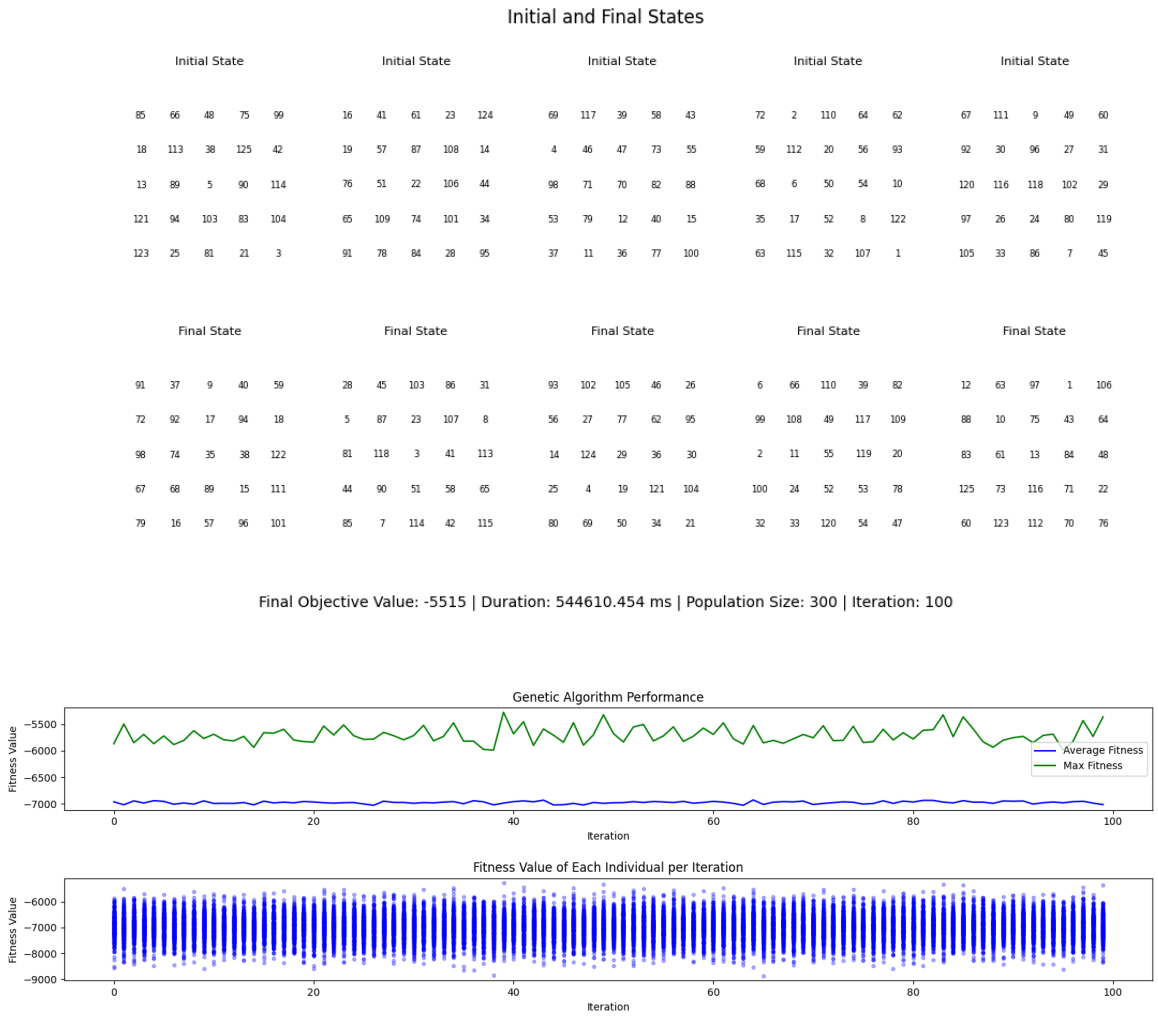
Test Case 5 (Untuk perubahan populasi terhadap iterasi yang sama)



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)



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 membuktikkan 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 T0, *cooling rate*, dan T1, 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