Tugas Besar 1 IF3070 Dasar Inteligensi Artifisial

Pencarian Solusi Diagonal Magic Cube dengan Local Search

Disusun oleh:

Kelompok 4

Adinda Khairunnisa I	/ 18222104
Alfaza Naufal Zakiy	/ 18222126
Chairul Nur Wahid	/ 18222132
M. Maulana Firdaus R	/ 18222140

Program Studi Sistem dan Teknologi Informasi

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

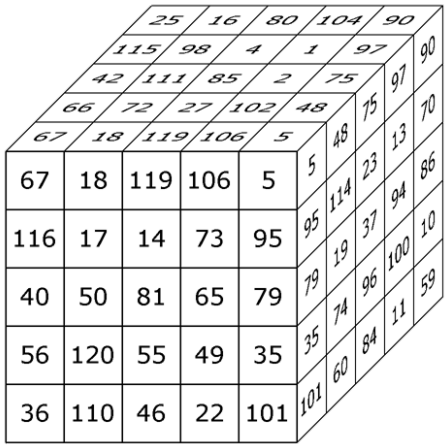
2024

Daftar Isi

Daftar Tabel

Daftar Gambar

1. Deskripsi Persoalan



Gambar 1. Magic cube dimensi 5x5x5

Dalam konsep matematika, magic cube merupakan bentuk 3 dimensi dari persegi ajaib yang berisi kumpulan bilangan bulat yang disusun dengan pola n^3 sehingga jumlah angka-angka pada setiap baris, pada setiap kolom, pada setiap tiang, dan pada setiap diagonal ruang adalah sama yang disebut sebagai magic number. Magic number dilambangkan dengan $M_3(n)$. Magic number memiliki rumus :

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

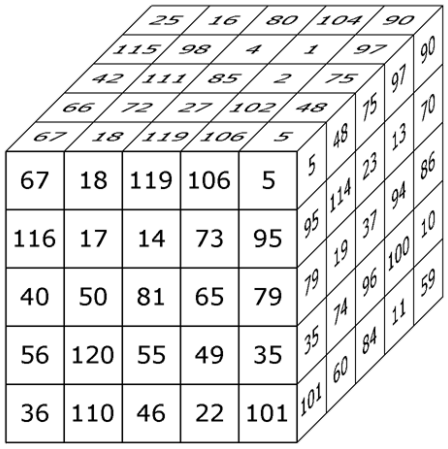
Dalam tugas ini, kami akan menyelesaikan permasalahan Diagonal Magic Cube berukuran 5x5x5. Initial state dari suatu kubus adalah susunan angka 1 hingga 5^3 secara acak. Magic number yang didapat dari ukuran 5x5x5 adalah 315.

Dari persoalan ini diharapkan mampu menjelaskan objective function, menjelaskan proses pencarian dengan memakai algoritma Local Search untuk mencari solusi Diagonal Magic Cube, menjelaskan algoritma local search yang terbaik untuk masalah ini, menjelaskan kelebihan dan kekurangan pendekatan local search dibandingkan dengan complete search, mampu menuliskan rencana kelas/fungsi.

2. Teori Dasar

a. Diagonal Magic Cube

Diagonal Magic Cube adalah sebuah susunan unik dari kubus berukuran $n \times n \times n$ tanpa pengulangan dengan n adalah panjang sisi pada kubus tersebut.



Gambar 2.1 Magic cube dimensi 5x5x5

Angka-angka tersusun sedemikian rupa sehingga poin-poin 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

b. Objective Function

Objective function adalah sebuah fungsi matematika yang digunakan untuk mengukur seberapa baik sebuah solusi memenuhi kriteria atau syarat dari suatu permasalahan.

c. Algoritma Local Search

Algoritma Local Search adalah metode optimasi umum yang digunakan dalam ilmu komputer untuk mengoptimalkan fungsi skalar. Algoritma ini mengeksplorasi *search space* dengan meningkatkan solusi secara bertahap melalui perubahan lokal, tanpa harus memeriksa seluruh *search space*. Algoritma ini sangat efektif untuk mengoptimalkan suatu fungsi, asalkan solusi yang diperoleh memenuhi tingkat konsistensi yang telah ditetapkan. Berikut merupakan contoh algoritma local search.

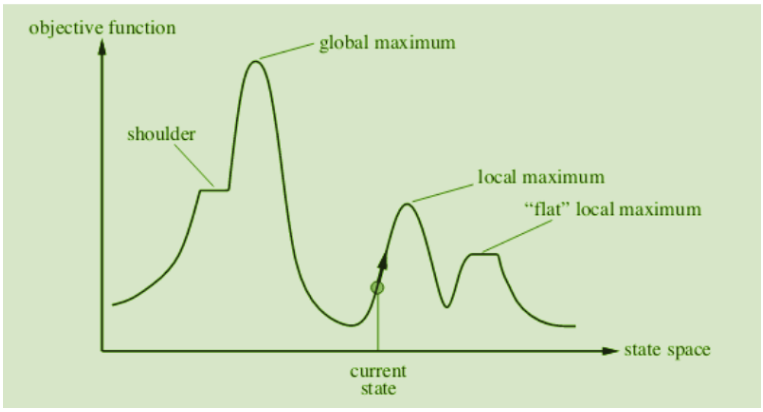
i. Hill-climbing Search

Algoritma ini dimulai dengan initial state yang di generate secara random. Lalu *current state* dilakukan pendakian yakni pindah ke *neighbor* yang terbaik dari

current state, ini dilakukan terus menerus sampai tidak ada lagi *neighbor* yang memiliki nilai yang lebih baik yaitu pada *peak* (puncak). *Peak* di sini bisa saja di g*lobal maximum, shoulder, local maximum,* dan *flat local maximum*.
Berikut merupakan fitur dalam Hill-climbing Search:

- Generate and Test variant: Memberikan umpan balik yang membantu memutuskan arah mana yang harus dituju dalam ruang pencarian.
- Greedy approach: Dalam state space apapun, pencarian bergerak ke arah yang mengoptimalkan cost dengan harapan menemukan solusi optimal di akhir.
- No backtracking: Tidak menelusuri kembali ruang pencarian, karena tidak mengingat status sebelumnya.
- Deterministic Nature: Dengan kondisi awal dan masalah yang sama, hasilnya akan selalu sama. Tidak ada keacakan atau ketidakpastian dalam operasinya.
- Local Neighborhood: Beroperasi dalam area kecil di sekitar solusi saat ini. Algoritma ini mengeksplorasi solusi yang terkait erat dengan kondisi saat ini dengan membuat perubahan kecil dan bertahap. Cara ini memungkinkan untuk menemukan solusi yang lebih baik daripada solusi saat ini, walaupun mungkin bukan solusi yang optimal secara global.

Berikut merupakan State-space Diagram Hill-climbing Search. Diagram ini menunjukkan grafik antara berbagai state algoritma dan Objective function/Cost.



Gambar 2.2. State-space Diagram Hill-climbing Search

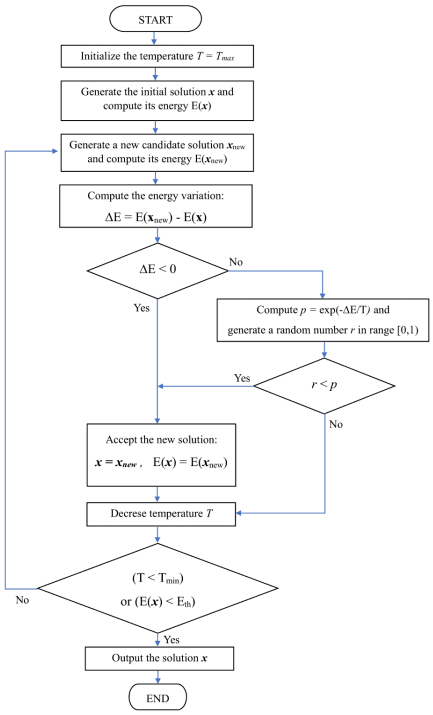
Sumbu-x : menunjukkan *state space*, yaitu keadaan atau konfigurasi yang dapat dicapai oleh algoritma kita
Sumbu-y : menunjukkan nilai objective function yang sesuai dengan keadaan tertentu
Solusi pencarian terbaik ada di global maximum yang memiliki nilai yang paling tinggi.
Macam-macam daerah di State-space Diagram:

- Local Maximum: Sebuah state yang lebih baik dari *neighbor states*, tetapi masih ada state yang lebih tinggi dari state ini.

- Global Maximum: Sebuah state terbaik yang ada di State-space Diagram. State ini memiliki nilai objective function tertinggi.
- Current state: Ini adalah state yang hadir saat ini di State-space Diagram
- Flat local maximum: Sebuah flat space yang semua *neighbor states* dari *current state* memiliki nilai yang sama.
- Shoulder: Sebuah wilayah plateau yang mempunyai uphil edge.

Berikut ini merupakan varian dari Hill-climbing Algorithm.

1. Steepest Ascent Hill-climbing
   Ini merupakan algoritma Hill-climbing yang umum digunakan. Cara kerja algoritma ini adalah mengevaluasi *initial state*, jika sudah di *goal state* maka selesai. Jika tidak di *goal state*, maka melakukan sebuah *loop* yang terus-menerus bergerak ke peningkatan nilai. Algoritma ini selesai saat mencapai *peak* termasuk *flat* ketika *neighbor* tidak memiliki nilai yang lebih tinggi.
2. Hill-climbing with Sideways Move
   Cara kerja algoritma ini mirip dengan Hill-climbing biasa, namun ada penambahan variasi untuk mencapai goal statenya. Variasi Sideways Move akan bergerak sepanjang solusi yang bernilai sama pada *plateau* sehingga peluang untuk mendapatkan solusi yang lebih baik meningkat. Namun langkah untuk variasi ini perlu dibatasi agar terhindar dari eksplorasi yang tidak berujung. Algoritma ini berakhir saat mencapai *peak*.
3. Random Restart Hill-climbing
   Cara kerja algoritma ini mirip dengan Hill-climbing biasa, namun ada penambahan variasi untuk mencapai goal statenya. Algoritma ini melakukan serangkaian pencarian *hill-climbing* dari *initial state* sampai goal ditemukan. Namun jika percobaan pertama belum sukses, coba terus sampai sukses. Proses ini bisa berulang beberapa kali dan dimulai dari initial states yang berbeda atau *restart* dalam setiap proses pencarian goal statenya.
4. Stochastic Hill-climbing
   Cara kerja Algoritma ini adalah memilih sebuah *successor* yang dibangkitkan secara random, kemudian langsung di assign sebagai *neighbor*. Lalu dilakukan pengecekan, jika neighbor valuenya > current. Maka neighbor di assign sebagai current. Jika tidak, neighbor diabaikan lalu loop dilanjutkan kembali.
5. Simulated Annealing
   Simulated Annealing adalah algoritma optimasi yang menggabungkan elemen dari Stochastic Hill-Climbing, di mana state yang lebih buruk dapat dipilih dengan probabilitas tertentu. Algoritma ini bekerja berdasarkan prinsip penurunan temperature T secara bertahap. Pada awal pencarian, nilai T tinggi memungkinkan algoritma untuk mengeksplorasi lebih banyak solusi yang lebih buruk dengan probabilitas lebih besar, mencegah algoritma terjebak di local optima. Seiring waktu, T dikurangi melalui cooling schedule, sehingga semakin sedikit state yang lebih buruk yang diterima. Algoritma ini menjamin bahwa jika nilai T turun cukup lambat (cooling schedule yang tepat), maka algoritma ini akan mencapai global optimum dengan probabilitas mendekati 1.



Gambar 2.3 Flowchart algoritma Simulated Annealing

ii. Genetic Algorithm

Algoritma ini merupakan varian dari stochastic beam search di mana successor states dibuat dengan menggabungkan dua parent state. Dalam algoritma ini, sebuah state direpresentasikan sebagai sebuah string, 1 karakter menyatakan 1 variabel. Cara kerjanya adalah membuat inisialisasi populasi. Kemudian, melakukan fitness function. Lalu melakukan selection point agar mendapatkan parents. Setelah itu dilakukan Cross-over yang menghasilkan 2 individu, tetapi ada versi lain yang menghasilkan 1 individu saja. Langkah selanjutnya adalah mutasi, mutasi dilakukan dengan menentukan mutasi pointnya, lalu dicari bilangan random untuk nilai random mutation. Sesudah itu, menghitung fitness untuk populasi baru. Algoritma ini berhenti ketika individu cukup fit atau waktu yang digunakan sudah cukup.

Berikut merupakan search space dari Genetic Algorithm

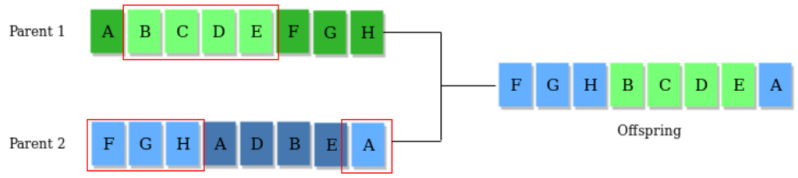
Gambar 2.4 Search space Genetic Algorithm

Populasi individu dipertahankan dalam search space. Setiap individu mewakili solusi dalam search space untuk masalah yang diberikan. Setiap individu dikodekan sebagai vektor panjang terbatas (serupa kromosom) dari komponen-komponen. Komponen-komponen variabel ini seperti Gen. Jadi kromosom (individu) terdiri dari beberapa gen (komponen variabel).

Genetic Algorithm mempunyai Fitness score, Fitness score diberikan kepada setiap individu yang menunjukkan kemampuan individu untuk bersaing. Fitness score sendiri adalah jumlah karakter yang berbeda dari karakter dalam string target pada indeks tertentu. Jadi, individu yang memiliki fitness value lebih rendah diberi lebih banyak preferensi.

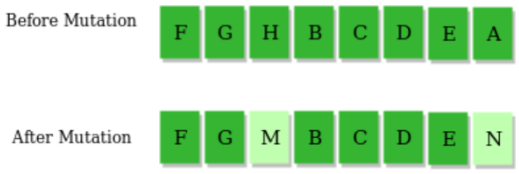
Genetic Algorithm memiliki berbagai macam operator, seperti Selection, Crossover, dan Mutation. Operator berguna untuk mengembangkan generasi. Berikut penjelasan setiap operatornya.

1. Selection Operator: Memberikan preferensi kepada individu dengan fitness score yang baik dan memungkinkan mereka untuk mewariskan gen mereka ke generasi berikutnya.
2. Crossover Operator: Mewakili perkawinan antar individu. Dua individu dipilih menggunakan Selection Operator dan lokasi crossover dipilih secara acak. Kemudian gen di lokasi crossover ini dipertukarkan sehingga menciptakan individu (keturunan) yang sama sekali baru.



Gambar 2.5 Crossover Operator

3. Mutation Operator: Memasukkan gen *random* pada keturunan untuk menjaga keberagaman dalam populasi guna menghindari konvergensi prematur.



Gambar 2.6 Mutation operator

3. Implementasi

a. Objective Function

Objective function dapat dirumuskan sebagai berikut:

$$\text{Objective Function} = \sum_{\text{Komponen}} \sum_{i=1}^{n} \left| \sum_{\text{Elemen} \in \text{Komponen}_i} e - MN \right|$$

Dimana:

- $\sum_{Komponen}$: Penjumlahan dilakukan untuk seluruh jenis komponen yang ada dalam kubus, yaitu baris, kolom, tiang, diagonal ruang, dan diagonal bidang.
- $\sum_{i=1}^{n}$: Penjumlahan dari setiap komponen ke-i pada jenis komponen yang sedang dihitung. Dengan demikian, nilai i bervariasi untuk setiap jenis komponen:
  - Untuk baris: i = 1..25
  - Untuk kolom: i = 1..25
  - Untuk tiang: i = 1..25
  - Untuk diagonal ruang: i = 1..4
  - Untuk diagonal bidang: i = 1..30
- $\sum_{\text{Elemen} \in \text{Komponen}_i}$: Penjumlahan semua elemen di dalam komponen ke-i tersebut.
- $\left| \sum_{\text{Elemen} \in \text{Komponen}_i} e - MN \right|$: perbedaan absolut antara jumlah elemen dalam suatu komponen ke-i dengan Magic Number (MN).

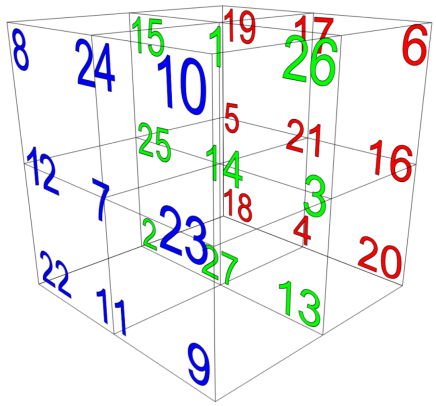
Rumus ini didapat dari penjumlahan penyimpangan absolut antara jumlah elemen dalam setiap komponen kubus dengan Magic Number. Pertama, kita menjumlahkan semua elemen dalam suatu komponen tertentu (baris, kolom, tiang, atau diagonal) untuk mendapatkan jumlah total elemen dalam komponen tersebut. Setelah itu, kita menghitung perbedaan antara jumlah tersebut dan Magic Number. Penyimpangan ini diukur dengan mengambil nilai absolutnya agar kita bisa melihat seberapa jauh komponen tersebut dari nilai yang diinginkan tanpa memperhatikan apakah selisihnya positif atau negatif.

Proses ini dilakukan untuk semua komponen dalam kubus. Oleh karena itu, rumus ini menggunakan tiga tingkat sigma (∑) untuk menjumlahkan penyimpangan dari semua elemen dalam setiap komponen dan komponen-komponen tersebut. Sigma pertama menjumlahkan penyimpangan dari berbagai jenis komponen dalam kubus (baris, kolom, tiang, diagonal), sigma kedua menjumlahkan setiap komponen ke-i dari jenis komponen tersebut, dan sigma ketiga menghitung jumlah elemen di dalam komponen tersebut. Hasil dari setiap komponen ini kemudian dijumlahkan untuk mendapatkan total penyimpangan dari konfigurasi kubus saat ini terhadap Magic Number. Tujuan dari algoritma Local Search adalah meminimalkan nilai objective function ini, atau idealnya menjadikannya 0.

Rumus tersebut dapat digunakan dengan cara:

1. Hitung jumlah elemen pada setiap baris ke-i dan Kurangi jumlah tersebut dengan Magic Number dan hitung nilai absolut dari selisih tersebut.
2. Lakukan hal yang sama untuk setiap kolom ke-i, dengan menjumlahkan semua elemen di dalam kolom tersebut dan mengukur penyimpangan absolut dari MN.
3. Hitung jumlah elemen pada setiap tiang ke-i, yang melintasi kubus dari lapisan bawah ke atas, dan ukur penyimpangannya dari MN.
4. Hitung jumlah elemen pada setiap diagonal ruang ke-i, yang melintasi seluruh dimensi kubus, dan hitung penyimpangan absolut dari MN.
5. Hitung jumlah elemen pada setiap diagonal bidang ke-i (misalnya, pada bidang xy, xz, atau yz), dan ukur penyimpangan dari MN.
6. Hitung total penyimpangan untuk setiap jenis komponen
7. Semua penyimpangan dari berbagai komponen di atas dijumlahkan untuk mendapatkan total objective function.

Sebagai contoh, dimiliki kubus 3x3x3 sebagai berikut:



Gambar 3.1. Ilustrasi potongan bidang kubus berukuran 3x3x3

Karena n = 3, maka jika dimasukkan ke dalam rumus magic number, diperoleh magic numbernya itu 42, cara perhitungan objective functionnya:

1. Hitung untuk semua baris
   a. Baris 1: 8 + 24 + 10 = |42 - 42| = 0
   b. Baris 2: 12 + 7 + 23 = |42 - 42| = 0
   c. Dst
2. Hitung untuk semua kolom:
   a. Kolom 1: 8 + 12 + 22 = |42 - 42| = 0
   b. Kolom 2: 24 + 7 + 11 = 42 - 42 = 0
   c. Dst
3. Hitung untuk semua tiang:
   a. Tiang 1: 8 + 12 + 22 = |42 - 42| = 0
   b. Tiang 2: 8 + 24 + 10 = |42 - 42| = 0
   c. Dst
4. Hitung untuk semua diagonal ruang:
   a. Diagonal Ruang 1: 8 + 14 + 20 = |42 - 42| = 0
   b. Diagonal Ruang 2: 19 + 14 + 9 = |42 - 42| = 0

c. Dst

5. Hitung untuk semua diagonal bidang
   a. Diagonal Bidang 1: 8 + 7 + 9 = |24 - 42| = 18
   b. Diagonal Bidang 2: 22 + 7 + 10 = |39 - 42| = 3
   c. Dst
6. Hitung untuk total masing masing elemen:
   a. Baris = 0
   b. Kolom = 0
   c. …
   d. Diagonal Bidang = 21
7. Hitung total keseluruhan simpangan:
   a. Total keseluruhan: 0 + 0 +... + 21 = 21

Dengan demikian, objective function yang diperoleh adalah 21, yang berarti kubus ini belum termasuk Magic Cube karena masih terdapat penyimpangan yang tidak sama dengan nol.

Implementasi Objective functionnya yaitu:

```python
def objective_function(self):
        """Calculate the total deviation from the magic number for rows, columns, pillars, and diagonals."""
        total_difference = 0

        # Check rows, columns, and pillars
        for i in range(self.size):
            for j in range(self.size):
                # Sum along each row (yz planes)
                total_difference += abs(sum(self.data[i][j]) - self.magic_number)
                # Sum along each column (xz planes)
                total_difference += abs(sum(self.data[k][i][j] for k in range(self.size)) - self.magic_number)
                # Sum along each pillar (xy planes)
                total_difference += abs(sum(self.data[j][k][i] for k in range(self.size)) - self.magic_number)

        # Check 3D space diagonals
        space_diagonals = [
            sum(self.data[i][i][i] for i in range(self.size)),  # (0,0,0) to (size-1,size-1,size-1)
            sum(self.data[i][i][self.size - 1 - i] for i in range(self.size)),  # (0,0,size-1) to (size-1,size-1,0)
            sum(self.data[i][self.size - 1 - i][i] for i in range(self.size)),  # (0,size-1,0) to (size-1,0,size-1)
            sum(self.data[i][self.size - 1 - i][self.size - 1 - i] for i in range(self.size))  # (0,size-1,size-1) to (size-1,0,0)
        ]
        total_difference += sum(abs(diag - self.magic_number) for diag in space_diagonals)

        # Check 2D plane diagonals in each slice
        for i in range(self.size):
            # Diagonals within each xy plane
            total_difference += abs(sum(self.data[i][j][j] for j in range(self.size)) - self.magic_number)
            total_difference += abs(sum(self.data[i][j][self.size - 1 - j] for j in range(self.size)) - self.magic_number)
            # Diagonals within each xz plane
            total_difference += abs(sum(self.data[j][i][j] for j in range(self.size)) - self.magic_number)
            total_difference += abs(sum(self.data[j][i][self.size - 1 - j] for j in range(self.size)) - self.magic_number)
            # Diagonals within each yz plane
            total_difference += abs(sum(self.data[j][j][i] for j in range(self.size)) - self.magic_number)
            total_difference += abs(sum(self.data[j][self.size - 1 - j][i] for j in range(self.size)) - self.magic_number)

        return total_difference
```

Gambar 3.2. Code Objective function

b. Steepest Ascent Hill-climbing

Algoritma ini dimulai dengan membuat sebuah solusi awal untuk magic cube. Kemudian melakukan generate untuk neighbor dengan memakai objective function. Kemudian dilakukan pengecekan untuk melihat value mana yang lebih baik. Langkah langkah lebih rincinya yaitu.

1. Konfigurasi awal dari Diagonal Magic Cube (solusi awal).
2. Generate semua konfigurasi neighbor dari solusi saat ini yang dihasilkan dengan melakukan swap dua elemen di Magic Cube.
3. Hitung nilai objective function untuk setiap neighbor
4. Pilih *neighbor* yang memiliki nilai objective function terbaik
5. Jika tidak ada neighbor yang lebih baik, berhenti dan kembalikan solusi saat ini sebagai solusi terbaik.
6. Jika ada neighbor yang lebih baik, pindah ke neighbor tersebut dan ulangi langkah 2 hingga 5.

Berikut merupakan source code implementasinya

```
import datetime
import itertools
import time
import copy
import matplotlib.pyplot as plt
from cube.cube import MagicCube

# The Steepest Ascent Hill Climbing class
class SteepestAscentHillClimbing:
    def __init__(self, magic_cube):
        self.magic_cube = magic_cube
        self.iterations = 0
        self.start_time = None
        self.end_time = None
        self.objective_values = []
        self.initial_cube = copy.deepcopy(self.magic_cube.data)
        self.final_cube = None

    def objective_function(self, cube_data):
        """Calculate the objective function value for a given cube
state."""
        original_data = self.magic_cube.data
        self.magic_cube.data = cube_data
        objective_value = self.magic_cube.objective_function()
        self.magic_cube.data = original_data
        return -objective_value

    def find_best_neighbor(self):
        """Find the best neighboring configuration by checking all
possible swaps."""
        indices = [(i, j, k) for i in range(self.magic_cube.size) for j
in range(self.magic_cube.size) for k in range(self.magic_cube.size)]
        best_neighbor = None
        best_value = float('-inf')
        current_data = copy.deepcopy(self.magic_cube.data)  # Copy the
current cube data for evaluation
```

```
        for pos1, pos2 in itertools.combinations(indices, 2):
            # Create a deep copy of the cube data to test the swap
            temp_data = copy.deepcopy(current_data)
            # Perform the swap on the temporary data
            x1, y1, z1 = pos1
            x2, y2, z2 = pos2
            temp_data[x1][y1][z1], temp_data[x2][y2][z2] = temp_data[x2]
[y2][z2], temp_data[x1][y1][z1]

            # Evaluate this neighbor
            neighbor_value = self.objective_function(temp_data)
            if neighbor_value > best_value:
                best_neighbor = (pos1, pos2)
                best_value = neighbor_value

        return best_neighbor, best_value

    def run(self):
        """Run the Steepest Ascent Hill Climbing algorithm with search
log display."""
        self.start_time = time.time()
        current_value = self.objective_function(self.magic_cube.data)
        self.objective_values.append(-current_value)  # Store initial
objective value

        while True:
            best_neighbor, best_value = self.find_best_neighbor()
            self.iterations += 1

            # Track objective function over iterations
            self.objective_values.append(-best_value)

            # Calculate elapsed time for this iteration
            elapsed_time = time.time() - self.start_time

            # Log iteration details to the console
            print(f"Iteration {self.iterations}: Objective = {-
best_value}, Time = {elapsed_time:.4f} seconds")
```

```
            # Stop if no better neighbor is found
            if best_neighbor is None or best_value <= current_value:
                break

            # Apply the best swap to the actual cube
            pos1, pos2 = best_neighbor
            self.magic_cube.swap(pos1, pos2)
            current_value = best_value

        self.end_time = time.time()
        # Capture the final state of the cube
        self.final_cube = copy.deepcopy(self.magic_cube.data)

    def report(self):
        """Display the results and plot the progress."""
        print("\nExperiment Report:")
        print(f"Initial State: {self.initial_cube}")
        print(f"Final State: {self.final_cube}")
        print(f"Final Objective Value: {self.objective_values[-1]}")
        print(f"Total Iterations: {self.iterations}")
```

```
        print(f"Duration: {self.end_time - self.start_time:.4f} seconds")

        # Generate the current timestamp in the desired format (e.g.,
YYYYMMDD_HHMMSS)
        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

        # Plotting
        plt.figure(figsize=(10, 6))
        plt.plot(self.objective_values, label='Objective Function')
        plt.xlabel('Iterations')
        plt.ylabel('Objective Function Value')
        plt.title('Objective Function Value over Iterations')
        plt.legend()
        plt.grid(True)

        # Save the plot with a timestamp in the filename

 plt.savefig(f'./data/steepest_ascent_hill_climbing_plot_{timestamp}.png'
, format='png')

        # Display the plot
        plt.show()


if __name__ == "__main__":
    cube = MagicCube(size=5)
    hill_climber = SteepestAscentHillClimbing(cube)
    hill_climber.run()
    hill_climber.report()
```

Gambar 3.3. Code Steepest Ascent Hill-climbing

c. Hill-climbing with Sideways Move

Mirip dengan Steepest Ascent Hill-Climbing, tetapi algoritma ini memungkinkan perpindahan ke neighbor yang memiliki nilai objective function yang sama (sideways move), dengan batasan jumlah gerakan samping yang bisa ditentukan. Hal ini memberikan fleksibilitas ketika algoritma terjebak dalam "flat local optima". Langkah langkah lebih rincinya yaitu.

1. Konfigurasi awal dari Diagonal Magic Cube (solusi awal).
2. Generate semua konfigurasi neighbor dari solusi saat ini yang dihasilkan dengan melakukan swap dua elemen di Magic Cube.
3. Hitung nilai objective function untuk setiap neighbor
4. Pilih neighbor yang memiliki nilai objective function terbaik
5. Jika nilai neighbor sama dengan nilai solusi saat ini, lakukan perpindahan ke neighbor tersebut (sideways move), dengan batasan jumlah gerakan samping.
6. Jika tidak ada neighbor yang lebih baik dan/ batas sideways move telah tercapai, berhenti dan kembalikan solusi saat ini.

   Berikut merupakan source code implementasinya

```python
import datetime
import itertools
import time
import copy
import matplotlib.pyplot as plt
from cube.cube import MagicCube

class HillClimbingWithSidewaysMove:
    def __init__(self, magic_cube, max_sideways):
        self.magic_cube = magic_cube
        self.max_sideways = max_sideways
        self.iterations = 0
        self.sideways_moves = 0
        self.start_time = None
        self.end_time = None
        self.objective_values = []
        self.initial_cube = copy.deepcopy(self.magic_cube.data)
        self.final_cube = None

    def objective_function(self, cube_data):
        original_data = self.magic_cube.data
        self.magic_cube.data = cube_data
        objective_value = self.magic_cube.objective_function()
        self.magic_cube.data = original_data
        return -objective_value

    def find_best_neighbor(self):
        indices = [(i, j, k) for i in range(self.magic_cube.size) for j
in range(self.magic_cube.size) for k in range(self.magic_cube.size)]
        best_neighbor = None
        best_value = float('-inf')
        current_data = copy.deepcopy(self.magic_cube.data)

        for pos1, pos2 in itertools.combinations(indices, 2):
            temp_data = copy.deepcopy(current_data)
            x1, y1, z1 = pos1
            x2, y2, z2 = pos2
            temp_data[x1][y1][z1], temp_data[x2][y2][z2] = temp_data[x2]
[y2][z2], temp_data[x1][y1][z1]

            neighbor_value = self.objective_function(temp_data)
            if neighbor_value > best_value:
                best_neighbor = (pos1, pos2)
                best_value = neighbor_value

        return best_neighbor, best_value

    def run(self):
        self.start_time = time.time()
        current_value = self.objective_function(self.magic_cube.data)
        self.objective_values.append(-current_value)

        while True:
            best_neighbor, best_value = self.find_best_neighbor()
            self.iterations += 1
```

```
            # Laporan tiap iterasi
            elapsed_time = time.time() - self.start_time
            print(f"Iteration {self.iterations}: Objective = {-
best_value}, Sideways Moves = {self.sideways_moves}, Time Elapsed =
{elapsed_time:.4f} seconds")

            if best_neighbor is None:
                break

            if best_value > current_value:
                self.sideways_moves = 0
            elif best_value == current_value and self.sideways_moves <
self.max_sideways:
                self.sideways_moves += 1
            else:
                break

            pos1, pos2 = best_neighbor
            self.magic_cube.swap(pos1, pos2)
            current_value = best_value
            self.objective_values.append(-best_value)

        self.end_time = time.time()
        self.final_cube = copy.deepcopy(self.magic_cube.data)

    def report(self):
        print("\nExperiment Report:")
        print(f"Initial State: ")
        print((self.initial_cube))
        print(f"Final State: ")
        print((self.final_cube))
        print(f"Final Objective Value: {self.objective_values[-1]}")
        print(f"Total Iterations: {self.iterations}")
        print(f"Total Sideways Moves: {self.sideways_moves}")
        print(f"Duration: {self.end_time - self.start_time:.4f} seconds")
        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

        # Plotting
        plt.figure(figsize=(10, 6))
        plt.plot(self.objective_values, label='Objective Function')
        plt.xlabel('Iterations')
        plt.ylabel('Objective Function Value')
        plt.title('Objective Function Value over Iterations')
        plt.legend()
        plt.grid(True)

        # Save the plot with a timestamp in the filename

 plt.savefig(f'./data/hill_climbing_with_sideways_move_plot_{timestamp}.p
ng', format='png')
        plt.show()

if __name__ == "__main__":
    cube = MagicCube(size=5)
    hill_climber = HillClimbingWithSidewaysMove(cube, max_sideways=100)
    hill_climber.run()
    hill_climber.report()
```

Gambar 3.4. Code Hill-climbing with Sideways Move

d. Random Restart Hill-climbing

Algoritma ini menjalankan Steepest Ascent Hill-Climbing berulang kali dari konfigurasi awal yang berbeda. Setiap kali algoritma terjebak di local optima, pencarian diulang dengan konfigurasi awal yang dihasilkan secara acak. Algoritma menyimpan solusi terbaik yang ditemukan selama berbagai restart. Langkah langkah lebih rincinya yaitu.

1. Konfigurasi awal dari Diagonal Magic Cube (solusi awal).
2. Generate semua konfigurasi neighbor dari solusi saat ini yang dihasilkan dengan melakukan swap dua elemen di Magic Cube.
3. Hitung nilai objective function untuk setiap neighbor
4. Pilih neighbor yang memiliki nilai objective function terbaik
5. Jika tidak ada neighbor yang lebih baik dan batas restart belum tercapai, lakukan restart dan cari solusi baru.
6. Ulangi proses restart hingga batas jumlah restart tercapai.

Berikut merupakan source code implementasinya

```
import copy
import datetime
import time
from matplotlib import pyplot as plt
from algorithms.steepest_ascent_hill_climbing import
SteepestAscentHillClimbing
from cube.cube import MagicCube

class RandomRestartHillClimbing:
    def __init__(self, magic_cube, max_restarts=5):
        self.magic_cube = magic_cube
        self.max_restarts = max_restarts
        self.best_cube_state = None
        self.best_objective_value = float('inf')
        self.all_objective_values_by_restart = []
        self.final_objective_values = []
        self.initial_cube_states = []
        self.final_cube_states = []
        self.restart_durations = []
        self.iterations_per_restart = []

    def run(self):
        overall_start_time = time.time()

        for restart in range(self.max_restarts):
            print(f"\nRestart {restart + 1}/{self.max_restarts}...")

            # Randomize initial state for each restart
            self.magic_cube.data = self.magic_cube.initialize_cube()

 self.initial_cube_states.append(copy.deepcopy(self.magic_cube.data))

            # Copy randomized cube state for the steepest ascent hill
climbing algorithm
            cube_copy = copy.deepcopy(self.magic_cube)
            hill_climber = SteepestAscentHillClimbing(cube_copy)
```

```
            # Track time and iterations for this restart
            restart_start_time = time.time()
            hill_climber.run()
            restart_end_time = time.time()
            self.restart_durations.append(restart_end_time -
restart_start_time)
            self.iterations_per_restart.append(hill_climber.iterations)

            # Append the objective values of this restart

 self.all_objective_values_by_restart.append(hill_climber.objective_value
s)

            # Store the final state and objective value of this restart
            final_value = hill_climber.objective_values[-1]
            self.final_objective_values.append(final_value)

 self.final_cube_states.append(copy.deepcopy(hill_climber.magic_cube.data
))

            # Check if this restart found a better solution
            if final_value < self.best_objective_value:
                self.best_objective_value = final_value
                self.best_cube_state =
copy.deepcopy(hill_climber.magic_cube.data)

        overall_end_time = time.time()
        self.total_duration = overall_end_time - overall_start_time

    def report(self):
        """Generate a report with detailed information and plots."""
        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

        print("\nExperiment Report:")
        total_iterations = sum(self.iterations_per_restart)
        for i in range(self.max_restarts):
            print(f"\nRestart {i + 1}/{self.max_restarts}:")
            print(f"Initial State: {self.initial_cube_states[i]}")
            print(f"Final State: {self.final_cube_states[i]}")
            print(f"Final Objective Value:
{self.final_objective_values[i]}")
            print(f"Iterations: {self.iterations_per_restart[i]}")
            print(f"Duration: {self.restart_durations[i]:.4f} seconds")

        print("\nBest Overall Solution:")
        print(f"Final Objective Value: {self.best_objective_value}")
        print(f"Best Final Cube State: {self.best_cube_state}")
        print(f"Total Iterations Across All Restarts:
{total_iterations}")
        print(f"Total Duration (all restarts): {self.total_duration:.4f}
seconds")

        # Plot 1: Objective Function Value over Iterations (with Random
Restarts)
        plt.figure(figsize=(12, 6))
```

```
        # Subplot 1: Iterations vs. Objective Function Value for each
restart
        plt.subplot(1, 2, 1)
        for restart_index, objective_values in
enumerate(self.all_objective_values_by_restart):
            plt.plot(objective_values, label=f'Restart {restart_index +
1}')

        plt.xlabel('Iterations')
        plt.ylabel('Objective Function Value')
        plt.title('Objective Function Value over Iterations (Random
Restarts)')
        plt.legend(title='Restart')
        plt.grid(True)

        # Subplot 2: Final Objective Function Value per Restart
        plt.subplot(1, 2, 2)
        plt.plot(range(1, self.max_restarts + 1),
self.final_objective_values, 'o-', color='red')
        plt.xlabel('Restart Number')
        plt.ylabel('Final Objective Function Value')
        plt.title('Final Objective Function Value per Restart')
        plt.grid(True)

        # Adjust layout to prevent overlap
        plt.tight_layout()

        # Save the combined plot with a timestamp in the filename

 plt.savefig(f'./data/random_restart_hill_climbing_combined_plot_{timesta
mp}.png', format='png')
        plt.show()

if __name__ == "__main__":
    cube = MagicCube(size=5)
    random_restart_hill_climber = RandomRestartHillClimbing(cube,
max_restarts=5)
    random_restart_hill_climber.run()
    random_restart_hill_climber.report()
```

Gambar 3.5. Code Random Restart Hill-climbing

e. Stochastic Hill-climbing

Berbeda dari Steepest Ascent Hill-Climbing, algoritma ini tidak mengevaluasi semua neighbor. Sebaliknya, ia memilih satu neighbor secara acak pada setiap langkah, dan jika nilai neighbor tersebut lebih baik daripada solusi saat ini, algoritma pindah ke neighbor tersebut. Algoritma terus berulang hingga tidak ada neighbor acak yang lebih baik. Langkah langkah lebih rincinya yaitu.

1. Konfigurasi awal dari Diagonal Magic Cube (solusi awal).
2. Pilih salah satu konfigurasi neighbor acak dari solusi saat ini yang dihasilkan dengan melakukan swap dua elemen di Magic Cube.
3. Hitung nilai objective function untuk neighbor tersebut
4. Jika nilai neighbor lebih baik dari solusi saat ini, pindah ke neighbor tersebut. Jika tidak, tetap di solusi saat ini.

5. Jika tidak ada perbaikan setelah beberapa iterasi, berhenti dan kembalikan solusi saat ini.

Berikut merupakan source code implementasinya

```python
import random
import copy
import time
import datetime
import matplotlib.pyplot as plt
from cube.cube import MagicCube

class StochasticHillClimbing:
    def __init__(self, magic_cube, max_trials=10000):
        self.magic_cube = magic_cube
        self.max_trials = max_trials
        self.iterations = 0
        self.start_time = None
        self.end_time = None
        self.objective_values = []
        self.initial_cube = copy.deepcopy(self.magic_cube.data)
        self.final_cube = None

    def run(self):
        # Simpan waktu mulai
        self.start_time = time.time()

        # Nilai objektif dari kondisi awal kubus
        current_cube = copy.deepcopy(self.initial_cube)
        current_cost = self.objective_function(current_cube)
        self.objective_values.append(current_cost)

        for _ in range(self.max_trials):
            self.iterations += 1

            # Buat tetangga dengan menukar dua posisi acak
            neighbor_cube = copy.deepcopy(current_cube)
            pos1 = self._random_position()
            pos2 = self._random_position(different_from=pos1)
            self.swap(neighbor_cube, pos1, pos2)

            # Hitung nilai objektif dari tetangga
            neighbor_cost = self.objective_function(neighbor_cube)

            # Jika tetangga lebih baik, pindah ke state tersebut
            if neighbor_cost < current_cost:
                current_cube = neighbor_cube
                current_cost = neighbor_cost

            # Simpan nilai objektif untuk setiap percobaan
            self.objective_values.append(current_cost)

            # Berhenti jika solusi optimal (biaya 0) ditemukan
            if current_cost == 0:
                print(f"Solusi optimal ditemukan pada iterasi
{self.iterations}")
                break
```

```
        # Simpan waktu selesai
        self.end_time = time.time()
        self.final_cube = current_cube

    def objective_function(self, cube_data):
        original_data = self.magic_cube.data
        self.magic_cube.data = cube_data
        cost = self.magic_cube.objective_function()
        self.magic_cube.data = original_data
        return cost

    def _random_position(self, different_from=None):
        size = self.magic_cube.size
        while True:
            position = (random.randint(0, size - 1),
                        random.randint(0, size - 1),
                        random.randint(0, size - 1))
            if position != different_from:
                return position

    def swap(self, cube_data, pos1, pos2):
        x1, y1, z1 = pos1
        x2, y2, z2 = pos2
        cube_data[x1][y1][z1], cube_data[x2][y2][z2] = cube_data[x2][y2]
[z2], cube_data[x1][y1][z1]

    def report(self):
        duration = self.end_time - self.start_time
        print("===== Laporan Hasil Stochastic Hill Climbing =====")
        print(f"Durasi Pencarian       : {duration:.4f} detik")
        print(f"Total Iterasi          : {self.iterations}")
        print(f"Nilai Objective Awal   : {self.objective_values[0]}")
        print(f"Initial State: ")
        print((self.initial_cube))
        print(f"Nilai Objective Akhir  : {self.objective_values[-1]}")
        print(f"Final State: ")
        print((self.final_cube))
```

```
        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

        # Plot hasil
        plt.figure(figsize=(10, 5))
        plt.plot(self.objective_values)
        plt.xlabel('Iterasi')
        plt.ylabel('Nilai Objective Function')
        plt.title('Performa Stochastic Hill Climbing')
        plt.grid(True)

 plt.savefig(f'./data/stochastic_hill_climbing_plot_{timestamp}.png',
format='png')
        plt.show()


if __name__ == "__main__":
    cube_size = 5
    magic_cube = MagicCube(size=cube_size)
    shc = StochasticHillClimbing(magic_cube, max_trials=10000)
    shc.run()
    shc.report()
```

Gambar 3.6. Code Stochastic Hill-climbing

f. Simulated Annealing

Algoritma ini dimulai dengan solusi awal dan memilih neighbor secara acak. Tidak seperti hill-climbing, Simulated Annealing memungkinkan perpindahan ke solusi yang lebih buruk dengan probabilitas tertentu yang bergantung pada temperatur yang menurun secara bertahap (cooling schedule). Ketika temperatur mendekati nol, algoritma bertindak lebih seperti hill-climbing. Langkah langkah lebih rincinya yaitu.

1. Mulai dengan solusi awal dan tentukan temperatur awal (T).
2. Pilih satu neighbor acak dari solusi saat ini.
3. Pindah ke neighbor:
    a. Jika neighbor lebih baik, maka pindah
    b. Jika neighbor tidak lebih baik, pindah dengan probabilitas $e^{\Delta E/T}$
4. Kurangi temperatur T sesuai cooling schedule.
5. Jika temperatur mendekati nol, berhenti dan kembalikan solusi saat ini sebagai solusi terbaik.
6. Lakukan iterasi langkah 2 hingga 6 hingga solusi optimal ditemukan.

Berikut merupakan source code implementasinya

```python
import random
import math
import copy
import matplotlib.pyplot as plt
from datetime import datetime
from cube.cube import MagicCube

class SimulatedAnnealing:
    def __init__(self, magic_cube, initial_temp, cooling_rate):
        self.magic_cube = magic_cube
        self.initial_temp = initial_temp
        self.cooling_rate = cooling_rate
        self.iterations = 0
        self.start_time = None
        self.end_time = None
        self.objective_values = []
        self.acceptance_probabilities = []  # Store exp(-delta_e /
current_temp)
        self.initial_cube = copy.deepcopy(self.magic_cube.data)
        self.final_cube = None

    def find_neighbor(self):
        pos1 = (random.randint(0, self.magic_cube.size - 1),
                random.randint(0, self.magic_cube.size - 1),
                random.randint(0, self.magic_cube.size - 1))

        pos2 = (random.randint(0, self.magic_cube.size - 1),
                random.randint(0, self.magic_cube.size - 1),
                random.randint(0, self.magic_cube.size - 1))

        # Ensure pos1 and pos2 are different
        while pos1 == pos2:
            pos2 = (random.randint(0, self.magic_cube.size - 1),
                    random.randint(0, self.magic_cube.size - 1),
                    random.randint(0, self.magic_cube.size - 1))
```

```
        neighbor_cube = copy.deepcopy(self.magic_cube)
        neighbor_cube.swap(pos1, pos2)

        return neighbor_cube

    def run(self):
        current_temp = self.initial_temp
        current_objective = self.magic_cube.objective_function()
        best_objective = current_objective
        best_cube = copy.deepcopy(self.magic_cube.data)
        self.start_time = datetime.now()

        while True:
            # Find neighbor
            neighbor_cube = self.find_neighbor()
            new_objective = neighbor_cube.objective_function()
            delta_e = new_objective - current_objective

            # Compute acceptance probability
            if delta_e < 0:
                acceptance_prob = 1.0
            else:
                acceptance_prob = math.exp(-delta_e / current_temp)

            # Accept the neighbor based on the probability
            if delta_e < 0 or random.uniform(0, 1) < acceptance_prob:
                self.magic_cube = neighbor_cube
                current_objective = new_objective

                if current_objective < best_objective:
                    best_objective = current_objective
                    best_cube = copy.deepcopy(self.magic_cube.data)

            # Store values
            self.objective_values.append(current_objective)
            self.acceptance_probabilities.append(acceptance_prob)
            self.iterations += 1

            print(f"Iteration {self.iterations}: Temp =
{current_temp:.4f}, Objective = {current_objective}, Acceptance
Probability = {acceptance_prob:.4f}")

            # Update temperature
            current_temp *= self.cooling_rate

            # Termination condition
            if current_temp < 1e-10 or best_objective == 0:
                break

        # Final state
        self.final_cube = best_cube
        self.end_time = datetime.now()

    def report(self):
        # Execution time
        duration = (self.end_time - self.start_time).total_seconds()
```

```
        print("=== Simulated Annealing Report ===")
        print(f"Initial Objective Value: {self.objective_values[0]}")
        print(f"Final Objective Value: {self.objective_values[-1]}")
        print(f"Total Iterations: {self.iterations}")
        print(f"Execution Time: {duration:.4f} seconds")
        print("\nInitial State:")
        print((self.initial_cube))
        print("\nFinal State:")
        print((self.final_cube))
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

        # Plot changes in objective function and acceptance probability
over iterations
        plt.figure(figsize=(12, 5))

        # Plot objective function values over iterations
        plt.subplot(1, 2, 1)
        plt.plot(self.objective_values, label='Objective Value')
        plt.xlabel("Iterations")
        plt.ylabel("Objective Function Value")
        plt.title("Objective Function vs. Iterations")
        plt.legend()

        # Plot acceptance probabilities over iterations
        plt.subplot(1, 2, 2)
        plt.plot(self.acceptance_probabilities, label='Acceptance
Probability')
        plt.xlabel("Iterations")
        plt.ylabel("Acceptance Probability")
        plt.title("Acceptance Probability vs. Iterations")
        plt.legend()
        plt.tight_layout()

        # Save the plot
        plt.savefig(f'./data/simulated_annealing_plot_{timestamp}.png',
format='png')
        plt.show()


if __name__ == "__main__":
    n = 5
    initial_temp = 1000
    cooling_rate = 0.95

    magic_cube = MagicCube(n)
    sa = SimulatedAnnealing(magic_cube, initial_temp, cooling_rate)

    sa.run()
    sa.report()
```

Gambar 3.7. Code Simulated Annealing

g. Genetic Algorithm

Algoritma ini berbeda dari yang lain karena berbasis populasi, bukan berbasis satu solusi. Sebuah populasi dari solusi awal dihasilkan, dan pada setiap iterasi, solusi baru (anak) dihasilkan dari kombinasi (crossover) dua solusi (orang tua) yang dipilih berdasarkan fitness (objective function). Selain itu, mutasi acak dapat diterapkan untuk menjaga keberagaman solusi. Proses seleksi, crossover, dan mutasi ini terus diulang sampai solusi optimal ditemukan atau batas waktu tercapai. Langkah langkah lebih rincinya yaitu

1. Buat populasi awal berupa sejumlah konfigurasi acak dari Diagonal Magic Cube.
2. Hitung nilai objective function (fitness) dari setiap individu dalam populasi.
3. Pilih dua individu dari populasi berdasarkan fitness-nya untuk dijadikan orang tua.
4. Hasilkan individu baru (anak) dengan melakukan crossover antara dua orang tua.
5. Terapkan mutasi acak pada individu baru (swap elemen secara acak) untuk menjaga keberagaman solusi.
6. Hitung nilai fitness individu baru dan tambahkan ke populasi.
7. Buat populasi baru dengan memilih individu-individu terbaik dari populasi lama dan individu baru (generasi berikutnya).
8. Ulangi langkah 3 hingga 7 hingga batas iterasi tercapai atau solusi terbaik ditemukan.

Berikut ini merupakan code implementasinya.

```
import matplotlib.pyplot as plt
import time
import random
import copy
import datetime
from cube.cube import MagicCube

class GeneticAlgorithm:
    def __init__(self, magic_cube, amount_iteration, population_size,
mutation_rate=0.01):
        self.magic_cube = magic_cube
        self.amount_iteration = amount_iteration
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.population = []
        self.best_solution = None
        self.best_objective_value = float('inf')
        self.objective_values_history = []
        self.start_time = None
        self.end_time = None
        self.initial_state = None
        self.final_state = None

    def initialize_population(self):
        """Generate initial population with random cube states."""
        self.population = [self.magic_cube.initialize_cube() for _ in
range(self.population_size)]

    def calculate_fitness(self, individual):
        """Use the objective function directly as fitness, aiming for
lower values."""
```

```
        self.magic_cube.data = individual  # Set cube state
        return self.magic_cube.objective_function()  # Positive values,
lower is better

    def select_parents(self):
        """Select two parents using tournament selection, aiming for
lower fitness values."""
        tournament_size = min(4, self.population_size)  # Safe tournament
size
        parents = []
        for _ in range(2):  # Need two parents
            candidates = random.sample(self.population, tournament_size)
            parents.append(min(candidates, key=self.calculate_fitness))
        return parents

    def crossover(self, parent1, parent2):
        """Perform crossover on two parents, ensuring unique values in
each row."""
        child = []
        for layer1, layer2 in zip(parent1, parent2):
            child_layer = []
            for row1, row2 in zip(layer1, layer2):
                combined_row = list(set(row1 + row2))
                random.shuffle(combined_row)
                child_layer.append(combined_row[:self.magic_cube.size])
            child.append(child_layer)
        return child

    def mutate(self, individual):
        """Mutate an individual by shuffling a row within a layer,
maintaining unique values."""
        if random.random() < self.mutation_rate:
            layer = random.choice(individual)
            row = random.choice(layer)
            random.shuffle(row)

    def evolve_population(self):
        """Create a new population through selection, crossover, and
mutation."""
        new_population = []
        while len(new_population) < self.population_size:
            parent1, parent2 = self.select_parents()
            child = self.crossover(parent1, parent2)
            self.mutate(child)
            new_population.append(child)
        self.population = new_population

    def run(self):
        """Run the genetic algorithm to optimize the magic cube."""
        self.start_time = time.time()
        self.initialize_population()
        self.initial_state = copy.deepcopy(self.magic_cube.data)

        for iteration in range(self.amount_iteration):
            fitness_values = [self.calculate_fitness(individual) for
individual in self.population]
            best_index = fitness_values.index(min(fitness_values))
            best_fitness = fitness_values[best_index]
```

Gambar 3.8. Code Genetic Algorithm

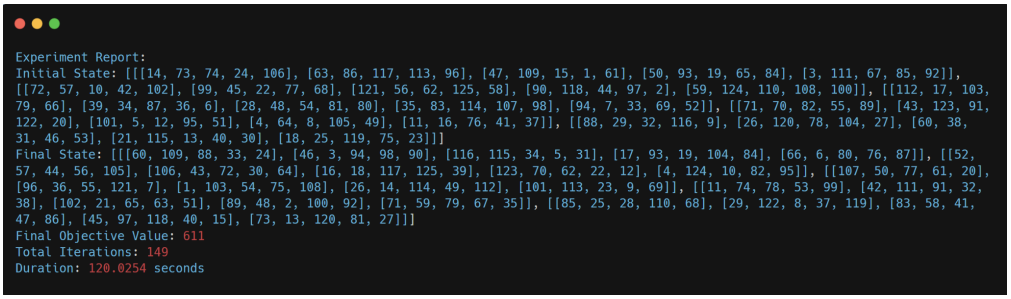
4. Eksperimen

Berikut adalah hasil eksperimen yang dilakukan untuk mengevaluasi kinerja berbagai algoritma optimasi dalam menyelesaikan permasalahan Magic Cube. Setiap eksperimen menggunakan pendekatan algoritma yang berbeda, dengan parameter khusus yang diatur untuk masing-masing algoritma. Berikut adalah hasil yang didapatkan dari masing-masing eksperimen:

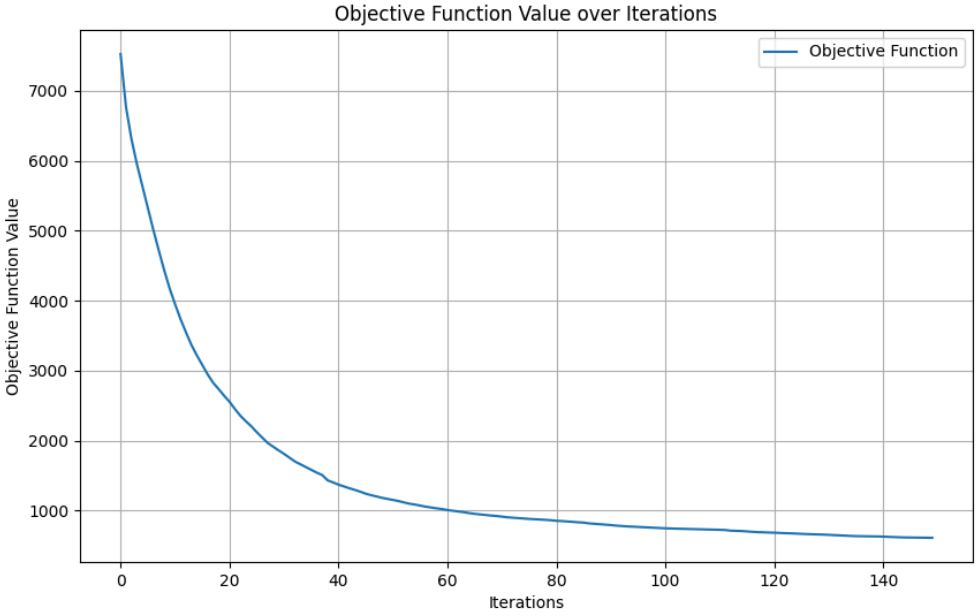
a. Eksperimen 1

i. Steepest Ascent Hill-climbing

Hasil yang didapat dari algoritma ini yaitu:



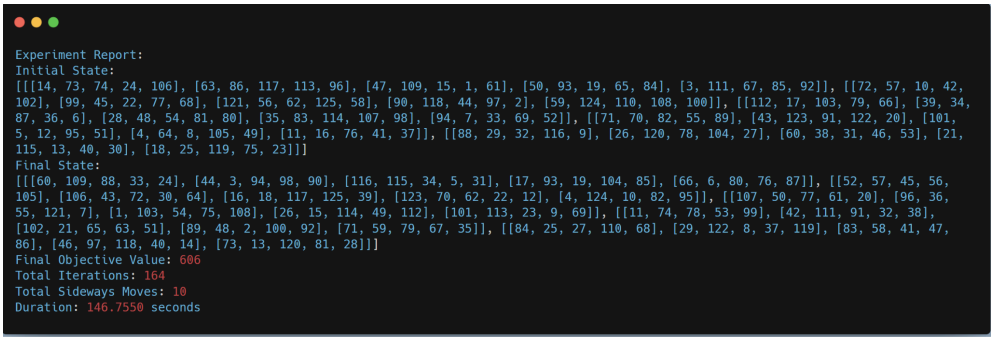
Gambar 4.1.1. Hasil dari algoritma Steepest Ascent Hill-climbing



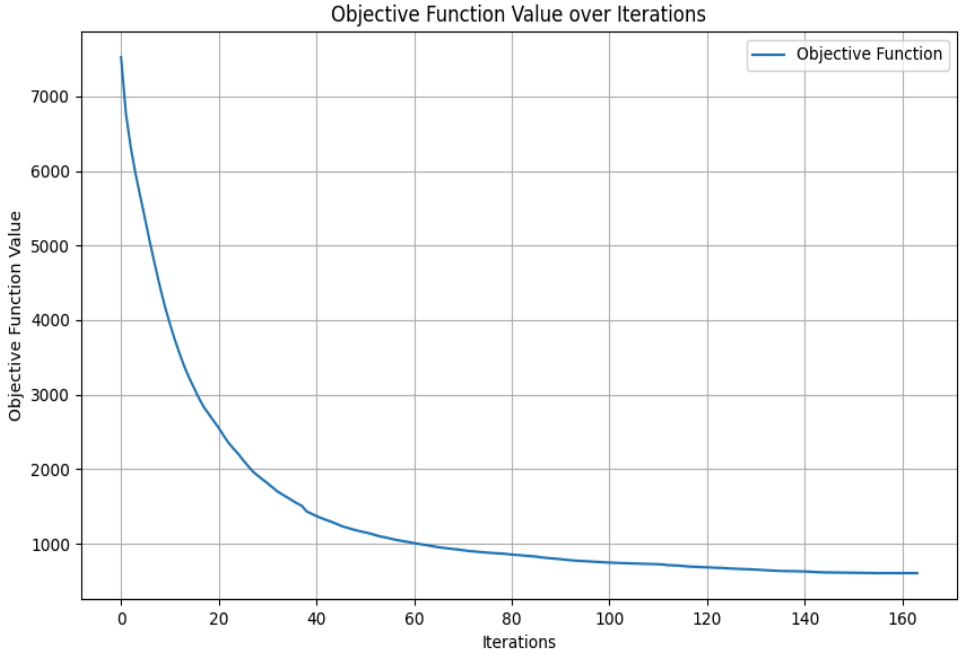
Gambar 4.1.2. Plot dari algoritma Steepest Ascent Hill-climbing

ii. Hill-climbing with Sideways Move

Dalam eksperimen ini, algoritma Hill-climbing with Sideways Move digunakan dengan batas maksimal sideways move sebanyak 10. Hasilnya sebagai berikut.



Gambar 4.1.3. Hasil dari algoritma Hill-climbing with Sideways Move

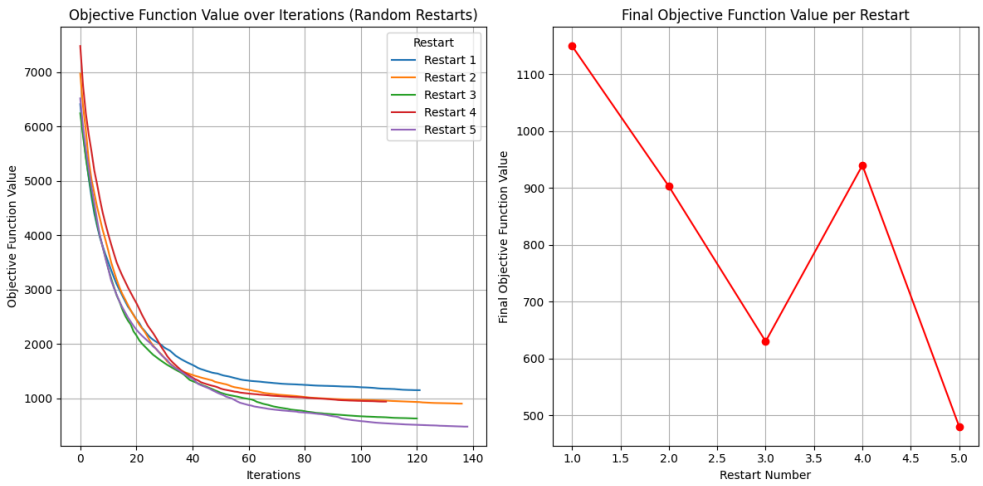


Gambar 4.1.4. Plot dari algoritma Hill-climbing with Sideways Move

iii. Random Restart Hill-climbing

Dalam eksperimen ini, algoritma Random Restart Hill-climbing digunakan dengan batas maksimal Restart sebanyak 5. Hasilnya sebagai berikut.

Gambar 4.1.5. Hasil dari algoritma Random Restart Hill-climbing



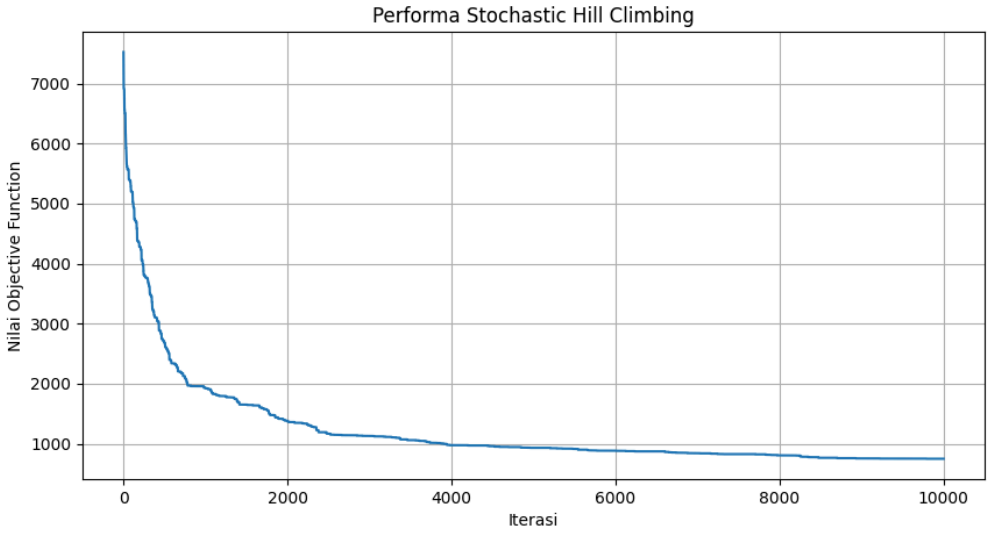
Gambar 4.1.6. Plot dari algoritma Random Restart Hill-climbing

iv. Stochastic Hill-climbing

Dalam eksperimen ini, algoritma Stochastic Hill-climbing digunakan dengan batas maksimal iterasi sebanyak 10.000. Hasilnya sebagai berikut.

```
===== Laporan Hasil Stochastic Hill Climbing =====
Durasi Pencarian       : 1.2981 detik
Total Iterasi          : 10000
Nilai Objective Awal   : 7525
Initial State:
[[[14, 73, 74, 24, 106], [63, 86, 117, 113, 96], [47, 109, 15, 1, 61], [50, 93, 19, 65, 84], [3, 111, 67, 85, 92]], [[72, 57, 10, 42,
102], [99, 45, 22, 77, 68], [121, 56, 62, 125, 58], [90, 118, 44, 97, 2], [59, 124, 110, 108, 100]], [[112, 17, 103, 79, 66], [39, 34,
87, 36, 6], [28, 48, 54, 81, 80], [35, 83, 114, 107, 98], [94, 7, 33, 69, 52]], [[71, 70, 82, 55, 89], [43, 123, 91, 122, 20], [101,
5, 12, 95, 51], [4, 64, 8, 105, 49], [11, 16, 76, 41, 37]], [[88, 29, 32, 116, 9], [26, 120, 78, 104, 27], [60, 38, 31, 46, 53], [21,
115, 13, 40, 30], [18, 25, 119, 75, 23]]]
Nilai Objective Akhir  : 749
Final State:
[[[62, 104, 73, 9, 67], [20, 25, 113, 121, 29], [125, 39, 90, 5, 51], [74, 8, 38, 89, 107], [32, 108, 4, 86, 61]], [[92, 56, 27, 22,
117], [44, 58, 54, 34, 122], [35, 55, 98, 114, 15], [124, 47, 110, 23, 11], [21, 103, 28, 112, 50]], [[52, 77, 59, 123, 6], [109, 37,
69, 45, 40], [26, 72, 60, 68, 88], [14, 81, 43, 78, 100], [118, 66, 83, 1, 80]], [[18, 75, 111, 65, 46], [41, 76, 70, 97, 49], [93,
63, 30, 13, 116], [105, 85, 12, 102, 10], [57, 17, 91, 53, 94]], [[95, 3, 42, 99, 79], [101, 120, 7, 19, 71], [33, 82, 36, 115, 48],
[2, 96, 106, 24, 87], [84, 16, 119, 64, 31]]]
```

Gambar 4.1.7. Hasil dari algoritma Stochastic Hill-climbing



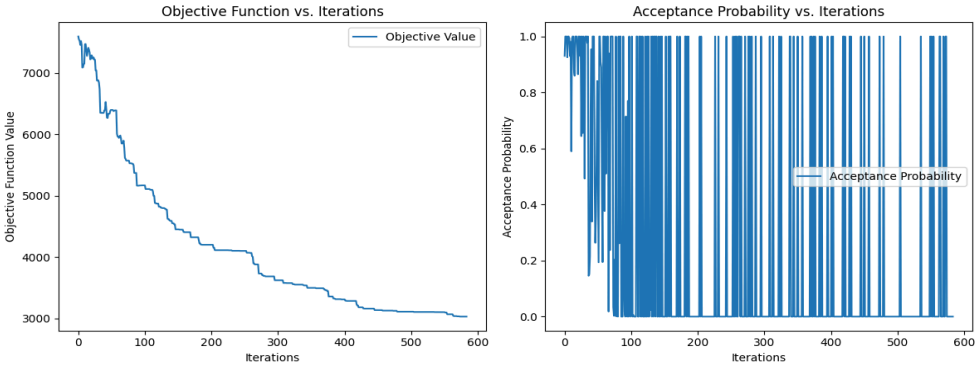
Gambar 4.1.8. Plot dari algoritma Stochastic Hill-climbing

v. Simulated Annealing

Dalam eksperimen ini, algoritma Simulated Annealing digunakan dengan batas temperatur awal 1000 dan cooling rate 0.95. Hasilnya sebagai berikut.

```
=== Simulated Annealing Report ===
Initial Objective Value: 7597
Final Objective Value: 3033
Total Iterations: 584
Execution Time: 0.2289 seconds
Frequency of getting stuck in local optima: 432

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

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

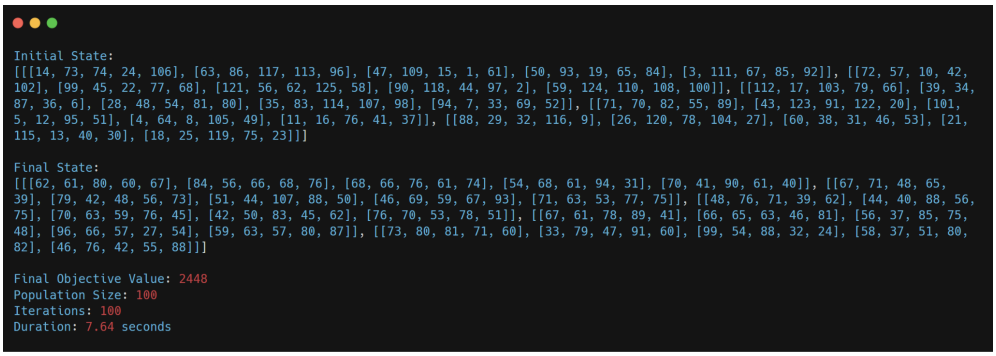
Gambar 4.1.9. Hasil dari algoritma Simulated Annealing



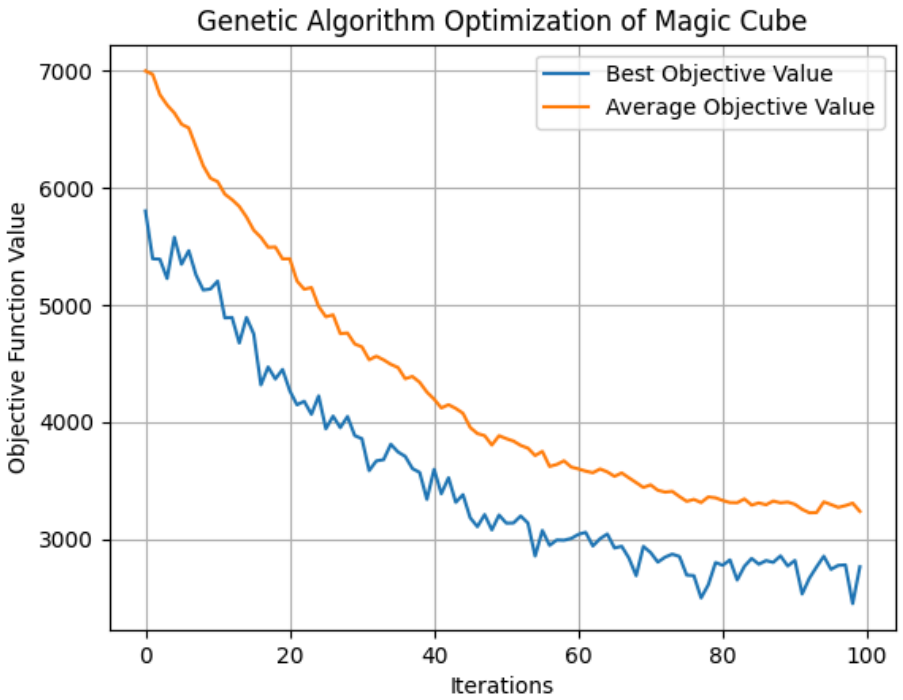
Gambar 4.1.10. Hasil dari algoritma Simulated Annealing

vi. Genetic Algorithm

Dalam eksperimen ini, digunakan beberapa konfigurasi algoritma Genetic Algorithm digunakan. Pada percobaan pertama dipakai dengan population size 100, iterations 100 dan mutation rate 0.01. Hasilnya sebagai berikut.



Gambar 4.1.11. Hasil dari algoritma Genetic Algorithm population size 100 dan iterations 100

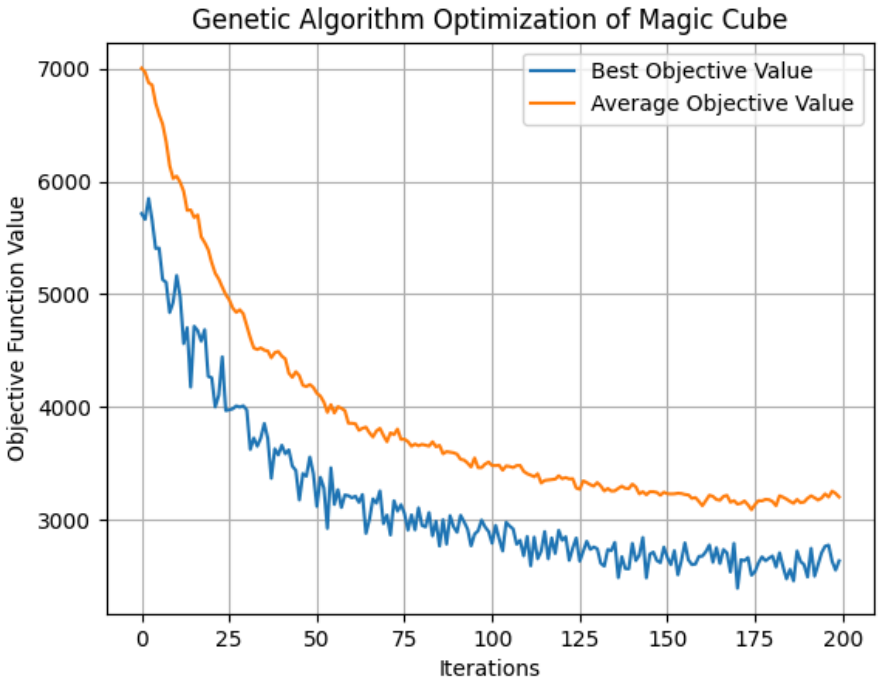


Gambar 4.1.12. Plot dari algoritma Genetic Algorithm population size 100 dan iterations 100

Pada percobaan selanjutnya dipakai dengan population size 100, iterations 200 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[49, 101, 85, 34, 71], [65, 50, 62, 103, 34], [55, 56, 101, 28, 77], [91, 41, 68, 97, 81], [73, 57, 82, 61, 42]], [[83, 72, 60, 47,
70], [74, 59, 73, 46, 85], [72, 78, 41, 75, 38], [55, 60, 75, 76, 57], [60, 64, 63, 68, 65]], [[47, 61, 75, 86, 59], [75, 51, 73, 52,
88], [77, 56, 75, 65, 46], [56, 54, 62, 36, 84], [59, 33, 61, 70, 84]], [[91, 42, 76, 72, 61], [79, 94, 89, 58, 68], [44, 49, 62, 89,
84], [88, 48, 58, 49, 47], [57, 80, 63, 49, 60]], [[70, 69, 47, 76, 54], [44, 73, 84, 41, 55], [48, 52, 57, 77, 88], [64, 39, 80, 71,
35], [96, 67, 33, 69, 56]]]

Final Objective Value: 2397
Population Size: 100
Iterations: 200
Duration: 14.68 seconds
```

Gambar 4.1.13. Hasil dari algoritma Genetic Algorithm population size 100 dan iterations 200

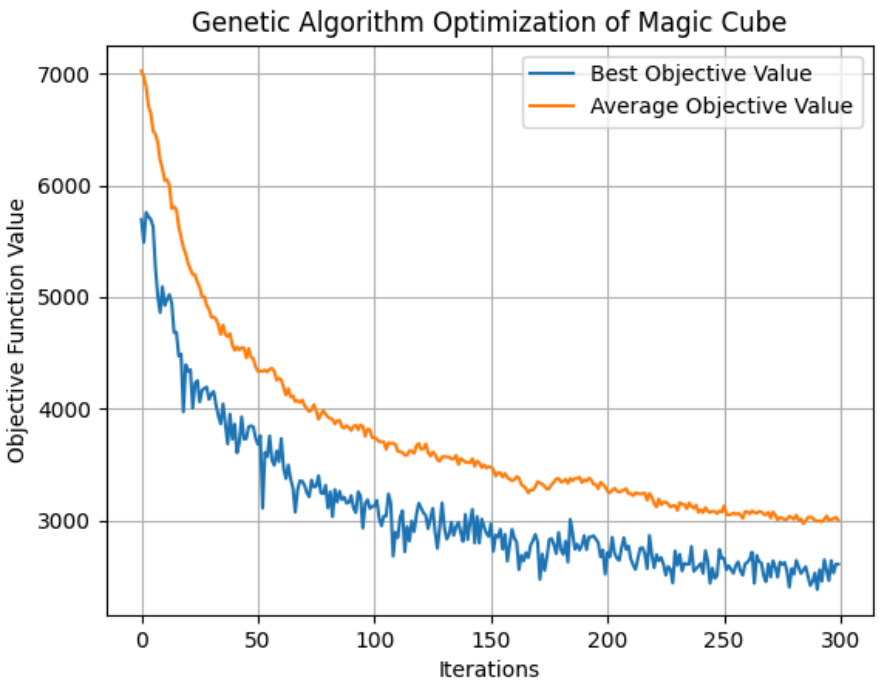


Gambar 4.1.14. Plot dari algoritma Genetic Algorithm population size 100 dan iterations 200

Pada percobaan selanjutnya dipakai dengan population size 100, iterations 300 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[66, 44, 72, 34, 91], [46, 68, 63, 61, 58], [66, 46, 50, 82, 97], [65, 69, 80, 62, 41], [66, 89, 44, 45, 70]], [[64, 38, 82, 35,
61], [48, 72, 80, 60, 53], [64, 81, 68, 84, 58], [66, 74, 45, 73, 63], [76, 70, 69, 72, 56]], [[94, 71, 70, 67, 17], [36, 76, 39, 59,
88], [71, 64, 72, 51, 50], [74, 80, 65, 67, 52], [55, 81, 47, 69, 62]], [[79, 72, 68, 48, 67], [46, 76, 61, 75, 66], [32, 36, 28, 73,
109], [59, 81, 77, 45, 55], [60, 51, 81, 34, 94]], [[43, 61, 70, 88, 79], [76, 71, 85, 51, 80], [55, 73, 81, 47, 50], [72, 41, 38, 47,
82], [61, 70, 47, 80, 84]]]

Final Objective Value: 2382
Population Size: 100
Iterations: 300
Duration: 25.26 seconds
```

Gambar 4.1.15. Hasil dari algoritma Genetic Algorithm population size 100 dan iterations 300

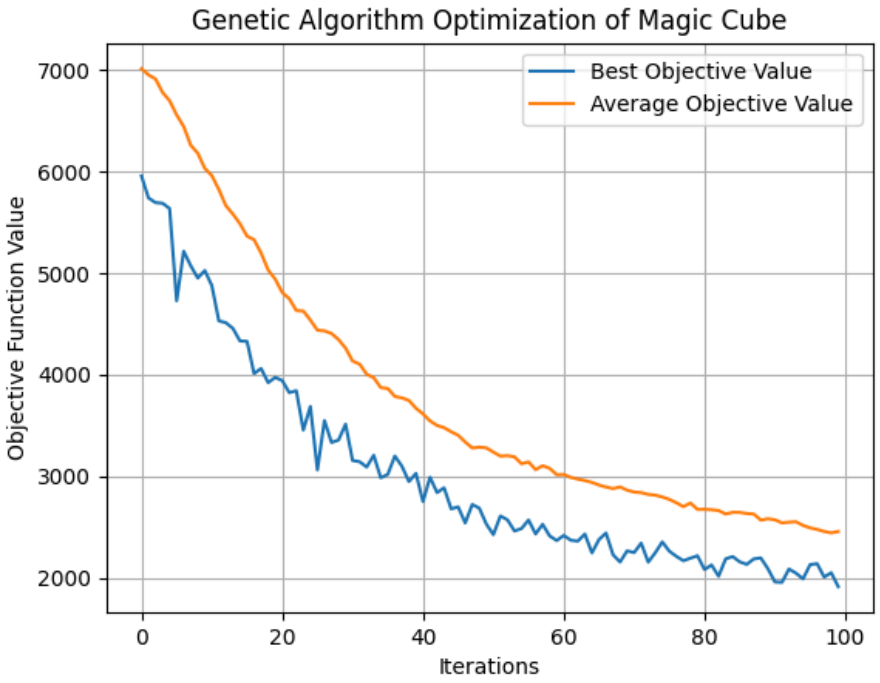


Gambar 4.1.16. Plot dari algoritma Genetic Algorithm population size 100 dan iterations 300

Pada percobaan selanjutnya dipakai dengan population size 200, iterations 100 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[60, 47, 75, 55, 37], [35, 48, 74, 54, 55], [81, 60, 53, 68, 55], [69, 64, 88, 61, 77], [76, 82, 53, 49, 68]], [[68, 69, 63, 66,
67], [67, 77, 52, 53, 60], [39, 81, 77, 70, 72], [67, 72, 69, 70, 71], [47, 44, 58, 75, 77]], [[48, 63, 56, 47, 66], [68, 77, 62, 61,
58], [55, 49, 63, 65, 51], [64, 58, 46, 77, 81], [74, 76, 61, 67, 48]], [[43, 87, 45, 54, 70], [75, 49, 78, 67, 65], [55, 44, 78, 28,
86], [66, 68, 63, 57, 60], [50, 75, 61, 55, 77]], [[71, 65, 72, 66, 67], [74, 81, 41, 63, 50], [62, 61, 56, 42, 60], [70, 58, 63, 66,
56], [62, 56, 76, 78, 54]]]

Final Objective Value: 1911
Population Size: 200
Iterations: 100
Duration: 15.12 seconds
```

Gambar 4.1.17. Hasil dari algoritma Genetic Algorithm population size 200 dan iterations 100

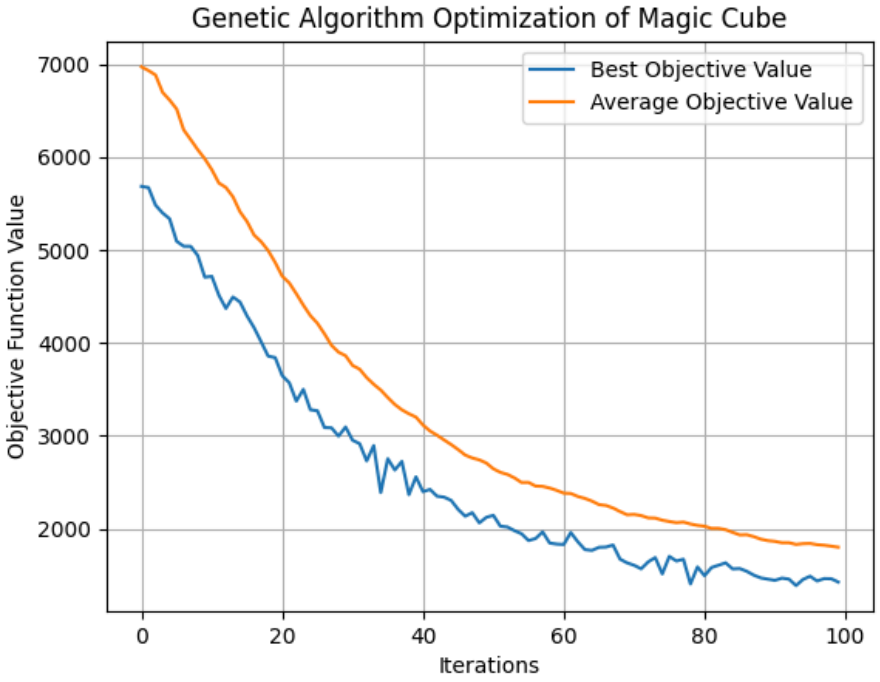


Gambar 4.1.18. Plot dari algoritma Genetic Algorithm population size 200 dan iterations 100

Pada percobaan selanjutnya dipakai dengan population size 300, iterations 100 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[74, 44, 56, 59, 75], [61, 83, 76, 54, 60], [50, 68, 69, 57, 62], [64, 59, 55, 51, 66], [67, 56, 49, 74, 58]], [[75, 51, 71, 76,
65], [43, 72, 57, 46, 56], [66, 80, 62, 60, 55], [70, 74, 68, 56, 51], [61, 53, 81, 46, 72]], [[60, 66, 57, 76, 64], [48, 62, 69, 76,
66], [62, 60, 59, 55, 65], [61, 55, 58, 77, 63], [60, 72, 84, 48, 50]], [[73, 60, 78, 45, 51], [50, 61, 53, 56, 77], [51, 47, 78, 69,
62], [59, 63, 67, 57, 71], [64, 56, 71, 70, 53]], [[62, 69, 72, 68, 64], [84, 74, 69, 56, 76], [66, 61, 67, 62, 52], [61, 67, 68, 52,
77], [66, 65, 55, 75, 52]]]

Final Objective Value: 1390
Population Size: 300
Iterations: 100
Duration: 21.49 seconds
```

Gambar 4.1.19. Hasil dari algoritma Genetic Algorithm population size 300 dan iterations 100



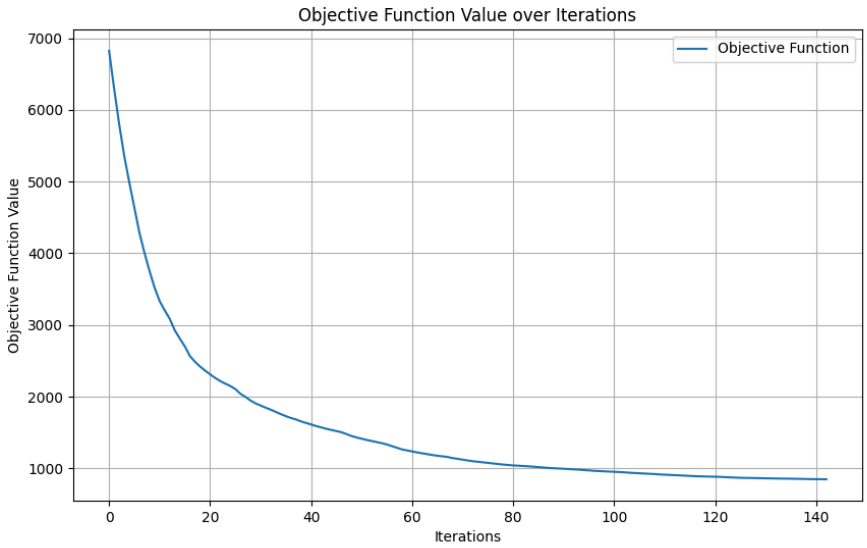
Gambar 4.1.20. Plot dari algoritma Genetic Algorithm population size 300 dan iterations 100

b. Eksperimen 2

i. Steepest Ascent Hill-climbing

```
Experiment Report:
Initial State: [[[50, 104, 115, 28, 44], [75, 36, 57, 113, 77], [34, 30, 80, 35, 83], [54, 7, 16, 26,
73], [18, 62, 64, 37, 74]], [[48, 45, 122, 112, 1], [114, 3, 24, 76, 105], [53, 11, 87, 56, 21], [58, 22,
120, 103, 71], [10, 107, 116, 43, 47]], [[14, 70, 49, 92, 59], [110, 95, 96, 6, 29], [23, 67, 55, 117,
82], [41, 60, 93, 78, 106], [118, 19, 63, 12, 2]], [[20, 108, 99, 79, 84], [98, 91, 125, 85, 100], [8,
38, 86, 33, 123], [25, 13, 66, 88, 15], [109, 121, 27, 39, 40]], [[81, 32, 102, 52, 9], [89, 51, 46, 65,
61], [69, 31, 72, 17, 124], [68, 5, 4, 111, 94], [97, 42, 119, 101, 90]]]
Final State: [[[100, 117, 72, 12, 17], [55, 36, 84, 108, 33], [38, 34, 86, 50, 107], [102, 85, 13, 25,
90], [20, 43, 64, 119, 68]], [[51, 26, 1, 122, 115], [94, 31, 9, 76, 105], [47, 116, 92, 56, 4], [113,
22, 112, 29, 42], [10, 120, 104, 32, 49]], [[41, 59, 53, 88, 78], [114, 96, 98, 5, 6], [23, 83, 39, 95,
77], [14, 60, 54, 106, 81], [123, 19, 71, 27, 74]], [[30, 11, 97, 91, 87], [45, 89, 3, 70, 110], [124,
48, 63, 40, 16], [21, 46, 125, 79, 44], [82, 118, 24, 37, 57]], [[93, 103, 99, 2, 18], [7, 62, 121, 65,
61], [69, 28, 35, 73, 111], [66, 109, 8, 75, 58], [80, 15, 52, 101, 67]]]
Final Objective Value: 849
Total Iterations: 142
Duration: 354.2756 seconds
```

Gambar 4.2.1 Hasil dari algoritma Steepest Ascent Hill-climbing

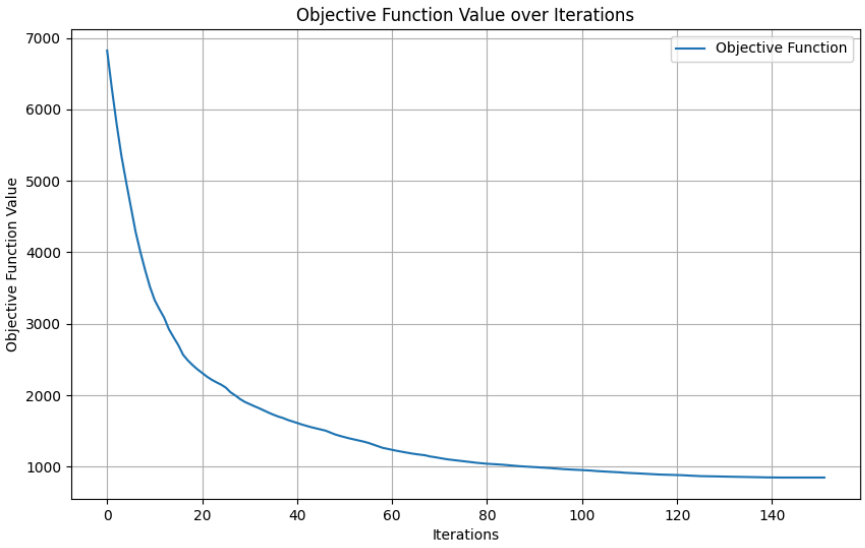


Gambar 4.2.2. Plot dari algoritma Steepest Ascent Hill-climbing

ii. Hill-climbing with Sideways Move

```
Experiment Report:
Initial State:
[[[50, 104, 115, 28, 44], [75, 36, 57, 113, 77], [34, 30, 80, 35, 83], [54, 7, 16, 26, 73], [18, 62, 64,
37, 74]], [[48, 45, 122, 112, 1], [114, 3, 24, 76, 105], [53, 11, 87, 56, 21], [58, 22, 120, 103, 71],
[10, 107, 116, 43, 47]], [[14, 70, 49, 92, 59], [110, 95, 96, 6, 29], [23, 67, 55, 117, 82], [41, 60, 93,
78, 106], [118, 19, 63, 12, 2]], [[20, 108, 99, 79, 84], [98, 91, 125, 85, 100], [8, 38, 86, 33, 123],
[25, 13, 66, 88, 15], [109, 121, 27, 39, 40]], [[81, 32, 102, 52, 9], [89, 51, 46, 65, 61], [69, 31, 72,
17, 124], [68, 5, 4, 111, 94], [97, 42, 119, 101, 90]]]
Final State:
[[[100, 117, 72, 12, 17], [55, 36, 84, 108, 33], [38, 34, 86, 50, 107], [102, 85, 13, 25, 90], [20, 43,
64, 119, 68]], [[51, 26, 1, 122, 115], [94, 31, 9, 76, 105], [47, 116, 92, 56, 4], [113, 22, 112, 29,
42], [10, 120, 104, 32, 49]], [[41, 59, 53, 88, 78], [114, 96, 98, 5, 6], [23, 83, 39, 95, 77], [14, 60,
54, 106, 81], [123, 19, 71, 27, 74]], [[30, 11, 97, 91, 87], [45, 89, 3, 70, 110], [124, 48, 63, 40, 16],
[21, 46, 125, 79, 44], [82, 118, 24, 37, 57]], [[93, 103, 99, 2, 18], [7, 62, 121, 65, 61], [69, 28, 35,
73, 111], [66, 109, 8, 75, 58], [80, 15, 52, 101, 67]]]
Final Objective Value: 849
Total Iterations: 152
Total Sideways Moves: 10
Duration: 200.8300 seconds
```

Gambar 4.2.3. Hasil dari algoritma Hill-climbing with Sideways Move



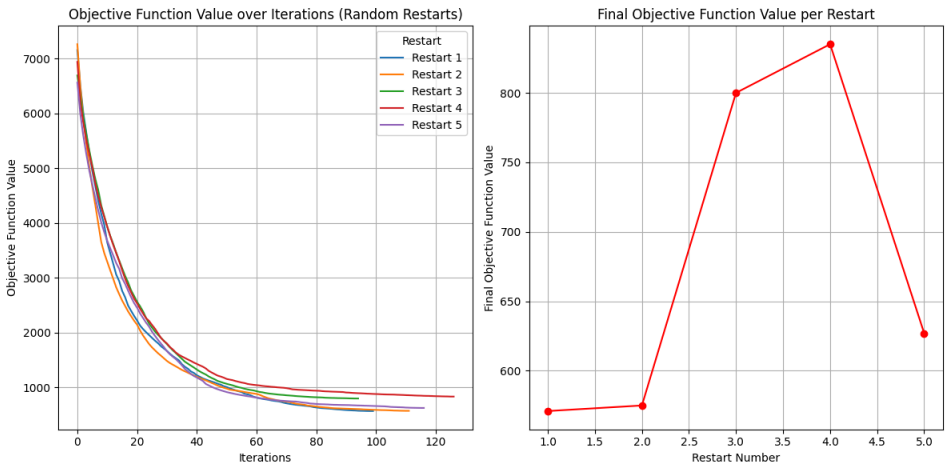
Gambar 4.2.4. Plot dari algoritma Hill-climbing with Sideways Move

iii. Random Restart Hill-climbing

```
Experiment Report:

Restart 1/5:
Initial State: [[[54, 63, 42, 74, 120], [41, 58, 12, 91, 25], [99, 44, 21, 123, 10], [45, 124, 81, 1,
30], [55, 122, 14, 18, 68]], [[112, 89, 26, 117, 35], [104, 69, 73, 11, 75], [51, 85, 84, 114, 111], [79,
94, 8, 24, 47], [106, 125, 22, 76, 67]], [[105, 70, 56, 49, 4], [32, 60, 109, 93, 115], [40, 50, 34, 107,
28], [86, 97, 64, 102, 53], [39, 88, 101, 113, 119]], [[31, 20, 37, 62, 96], [2, 13, 33, 23, 83], [29,
16, 87, 116, 92], [17, 5, 6, 110, 59], [100, 66, 9, 43, 77]], [[57, 52, 78, 48, 82], [65, 103, 38, 90,
19], [27, 80, 7, 61, 3], [98, 46, 36, 15, 71], [108, 118, 121, 95, 72]]]
Final State: [[[76, 49, 67, 3, 120], [40, 58, 102, 82, 33], [98, 48, 21, 123, 24], [36, 39, 75, 96, 69],
[65, 121, 50, 11, 68]], [[37, 100, 38, 106, 34], [104, 62, 47, 23, 81], [42, 85, 84, 16, 89], [18, 60,
119, 73, 45], [114, 8, 25, 95, 72]], [[115, 124, 9, 63, 4], [12, 7, 109, 88, 103], [55, 57, 52, 110, 28],
[86, 117, 44, 15, 53], [46, 2, 101, 41, 125]], [[31, 26, 122, 51, 74], [97, 77, 20, 32, 91], [13, 116,
87, 5, 94], [79, 29, 59, 108, 43], [93, 66, 27, 113, 14]], [[56, 6, 78, 92, 83], [64, 111, 35, 90, 17],
[107, 10, 71, 61, 80], [99, 70, 19, 22, 105], [1, 118, 112, 54, 30]]]
Final Objective Value: 571
Iterations: 99
Duration: 132.2505 seconds

Restart 2/5:
Initial State: [[[49, 55, 40, 112, 121], [3, 109, 22, 111, 50], [73, 31, 96, 63, 68], [23, 32, 74, 20,
71], [115, 53, 57, 123, 91]], [[107, 11, 58, 5, 59], [35, 86, 8, 84, 13], [95, 92, 88, 76, 46], [7, 77,
108, 38, 52], [48, 65, 60, 70, 69]], [[10, 15, 125, 34, 103], [67, 2, 66, 79, 27], [81, 85, 124, 44, 36],
[18, 75, 19, 117, 41], [43, 9, 100, 89, 24]], [[82, 122, 119, 78, 56], [118, 37, 102, 39, 113], [97, 29,
120, 87, 17], [99, 14, 62, 106, 83], [1, 45, 94, 105, 61]], [[21, 101, 64, 98, 12], [4, 80, 26, 30, 28],
[90, 54, 114, 42, 47], [6, 16, 110, 104, 51], [93, 72, 25, 116, 33]]]
Final State: [[[49, 24, 113, 112, 17], [1, 105, 11, 108, 98], [63, 100, 40, 65, 47], [110, 48, 66, 21,
62], [104, 38, 85, 5, 91]], [[107, 19, 58, 7, 124], [52, 56, 102, 77, 27], [96, 92, 2, 81, 44], [23, 76,
94, 83, 42], [36, 72, 60, 68, 78]], [[37, 22, 125, 15, 106], [99, 67, 90, 43, 16], [35, 69, 39, 51, 121],
[64, 75, 41, 117, 18], [79, 82, 10, 89, 55]], [[101, 123, 8, 71, 12], [118, 13, 88, 46, 50], [6, 29, 120,
87, 73], [86, 97, 4, 14, 119], [3, 53, 103, 95, 61]], [[28, 116, 9, 111, 54], [45, 84, 26, 34, 122],
[115, 25, 114, 31, 30], [32, 20, 109, 80, 74], [93, 70, 57, 59, 33]]]
Final Objective Value: 575
Iterations: 111
Duration: 145.3890 seconds
```

```
Restart 3/5:
Initial State: [[[14, 23, 13, 97, 30], [16, 120, 123, 83, 92], [49, 86, 46, 48, 93], [96, 90, 104, 66,
11], [84, 39, 25, 28, 50]], [[114, 47, 85, 69, 9], [52, 68, 91, 89, 61], [87, 119, 33, 124, 2], [103, 24,
51, 38, 7], [59, 35, 73, 100, 1]], [[105, 27, 57, 98, 109], [36, 17, 29, 3, 32], [53, 54, 88, 26, 81],
[65, 43, 37, 80, 31], [94, 45, 112, 74, 108]], [[12, 20, 116, 58, 113], [125, 40, 67, 64, 101], [71, 118,
10, 18, 60], [56, 102, 19, 62, 44], [15, 95, 70, 21, 34]], [[111, 6, 77, 122, 63], [5, 117, 110, 4, 107],
[76, 72, 42, 55, 82], [79, 78, 99, 121, 115], [106, 22, 8, 41, 75]]]
Final State: [[[49, 27, 125, 46, 30], [17, 116, 10, 85, 113], [47, 8, 44, 97, 119], [93, 43, 112, 66,
15], [109, 121, 25, 21, 38]], [[114, 108, 34, 6, 64], [52, 56, 45, 90, 72], [87, 101, 33, 19, 75], [11,
24, 67, 111, 115], [51, 35, 124, 89, 1]], [[69, 53, 39, 98, 57], [118, 16, 91, 48, 32], [23, 123, 76, 12,
81], [65, 94, 41, 78, 37], [40, 29, 70, 74, 106]], [[20, 120, 22, 58, 96], [122, 31, 59, 68, 18], [80,
13, 100, 86, 36], [84, 55, 54, 62, 60], [9, 95, 82, 28, 105]], [[50, 7, 88, 107, 63], [5, 117, 110, 2,
83], [77, 71, 61, 102, 4], [79, 99, 42, 3, 92], [104, 26, 14, 103, 73]]]
Final Objective Value: 800
Iterations: 94
Duration: 121.5784 seconds

Restart 4/5:
Initial State: [[[54, 88, 122, 118, 63], [93, 23, 87, 6, 76], [92, 98, 89, 125, 109], [61, 33, 13, 114,
60], [7, 26, 96, 70, 112]], [[69, 47, 94, 80, 55], [86, 21, 64, 12, 2], [66, 75, 58, 77, 37], [105, 124,
73, 5, 120], [121, 100, 113, 59, 103]], [[91, 107, 71, 52, 28], [111, 22, 35, 53, 117], [99, 9, 57, 1,
62], [81, 19, 45, 90, 51], [82, 16, 106, 95, 27]], [[43, 104, 17, 40, 50], [8, 84, 24, 115, 46], [31, 3,
79, 85, 83], [42, 25, 32, 44, 68], [38, 4, 39, 65, 108]], [[56, 102, 67, 123, 14], [11, 119, 36, 49,
101], [10, 78, 20, 74, 97], [30, 18, 72, 34, 15], [48, 41, 110, 116, 29]]]
Final State: [[[53, 89, 4, 98, 71], [99, 13, 87, 23, 93], [85, 118, 88, 8, 16], [61, 69, 44, 110, 32],
[17, 21, 97, 77, 103]], [[107, 55, 94, 7, 51], [83, 24, 101, 106, 2], [63, 22, 68, 121, 39], [27, 109,
46, 9, 124], [36, 102, 5, 72, 100]], [[48, 18, 90, 47, 111], [10, 91, 35, 58, 120], [92, 66, 60, 30, 59],
[84, 14, 74, 116, 25], [81, 125, 56, 64, 1]], [[42, 104, 114, 43, 12], [112, 70, 54, 76, 3], [6, 31, 79,
86, 113], [38, 95, 29, 45, 108], [115, 15, 40, 65, 80]], [[67, 49, 19, 123, 57], [11, 119, 37, 52, 96],
[82, 78, 20, 73, 62], [105, 28, 122, 34, 26], [50, 41, 117, 33, 75]]]
Final Objective Value: 835
Iterations: 126
Duration: 163.1087 seconds

Restart 5/5:
Initial State: [[[39, 99, 60, 9, 119], [85, 70, 57, 22, 64], [26, 49, 31, 3, 65], [59, 76, 106, 29, 117],
[43, 42, 95, 121, 47]], [[25, 37, 120, 16, 83], [15, 86, 91, 19, 115], [62, 4, 34, 21, 40], [8, 72, 66,
104, 80], [110, 90, 2, 69, 51]], [[63, 28, 20, 32, 1], [54, 100, 13, 116, 30], [5, 98, 36, 45, 101], [11,
122, 93, 35, 24], [112, 118, 75, 97, 78]], [[102, 38, 81, 87, 113], [68, 48, 124, 88, 92], [94, 55, 44,
53, 103], [50, 7, 41, 108, 77], [73, 58, 10, 96, 12]], [[84, 123, 52, 79, 107], [105, 125, 33, 6, 14],
[111, 23, 114, 89, 17], [27, 18, 67, 71, 56], [46, 74, 109, 82, 61]]]
Final State: [[[44, 100, 36, 13, 104], [77, 79, 56, 74, 48], [96, 26, 30, 97, 66], [59, 68, 87, 50, 51],
[39, 42, 106, 82, 46]], [[18, 29, 122, 19, 121], [69, 58, 91, 88, 9], [62, 113, 33, 38, 72], [114, 20,
65, 105, 11], [52, 95, 2, 64, 102]], [[110, 22, 71, 112, 1], [54, 90, 17, 116, 31], [57, 98, 60, 28, 73],
[8, 76, 93, 21, 117], [86, 25, 75, 37, 92]], [[63, 53, 83, 101, 14], [5, 47, 119, 34, 107], [16, 55, 103,
40, 99], [108, 35, 4, 94, 81], [123, 125, 10, 45, 12]], [[80, 109, 3, 70, 78], [115, 41, 32, 6, 120],
[84, 23, 89, 111, 7], [27, 118, 67, 43, 49], [15, 24, 124, 85, 61]]]
Final Objective Value: 627
Iterations: 116
Duration: 179.9395 seconds

Best Overall Solution:
Final Objective Value: 571
Best Final Cube State: [[[76, 49, 67, 3, 120], [40, 58, 102, 82, 33], [98, 48, 21, 123, 24], [36, 39, 75,
96, 69], [65, 121, 50, 11, 68]], [[37, 100, 38, 106, 34], [104, 62, 47, 23, 81], [42, 85, 84, 16, 89],
[18, 60, 119, 73, 45], [114, 8, 25, 95, 72]], [[115, 124, 9, 63, 4], [12, 7, 109, 88, 103], [55, 57, 52,
110, 28], [86, 117, 44, 15, 53], [46, 2, 101, 41, 125]], [[31, 26, 122, 51, 74], [97, 77, 20, 32, 91],
[13, 116, 87, 5, 94], [79, 29, 59, 108, 43], [93, 66, 27, 113, 14]], [[56, 6, 78, 92, 83], [64, 111, 35,
90, 17], [107, 10, 71, 61, 80], [99, 70, 19, 22, 105], [1, 118, 112, 54, 30]]]
Total Iterations Across All Restarts: 546
Total Duration (all restarts): 742.2761 seconds
```

Gambar 4.2.5. Hasil dari algoritma Random Restart Hill-climbing

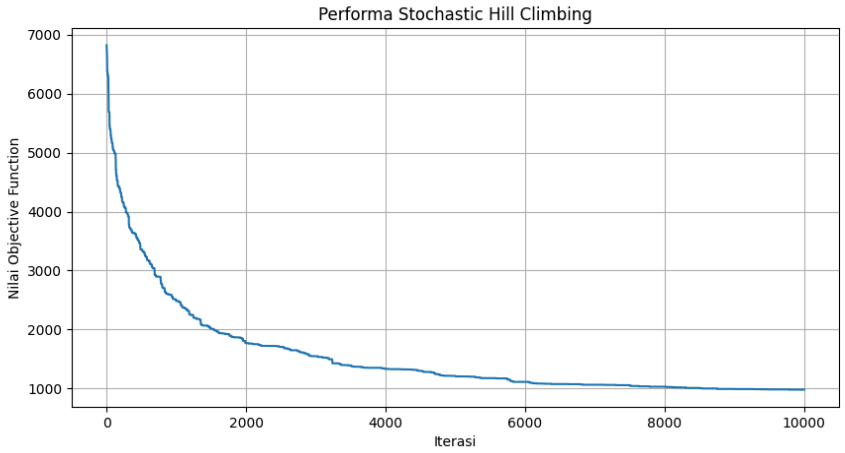


Gambar 4.2.6. Plot dari algoritma Random Restart Hill-climbing

iv. Stochastic Hill-climbing

```
===== Laporan Hasil Stochastic Hill Climbing =====
Durasi Pencarian       : 3.5866 detik
Total Iterasi          : 10000
Nilai Objective Awal   : 6823
Initial State:
[[[50, 104, 115, 28, 44], [75, 36, 57, 113, 77], [34, 30, 80, 35, 83], [54, 7, 16, 26, 73], [18, 62, 64,
37, 74]], [[48, 45, 122, 112, 1], [114, 3, 24, 76, 105], [53, 11, 87, 56, 21], [58, 22, 120, 103, 71],
[10, 107, 116, 43, 47]], [[14, 70, 49, 92, 59], [110, 95, 96, 6, 29], [23, 67, 55, 117, 82], [41, 60, 93,
78, 106], [118, 19, 63, 12, 2]], [[20, 108, 99, 79, 84], [98, 91, 125, 85, 100], [8, 38, 86, 33, 123],
[25, 13, 66, 88, 15], [109, 121, 27, 39, 40]], [[81, 32, 102, 52, 9], [89, 51, 46, 65, 61], [69, 31, 72,
17, 124], [68, 5, 4, 111, 94], [97, 42, 119, 101, 90]]]
Nilai Objective Akhir  : 978
Final State:
[[[51, 114, 92, 14, 56], [66, 59, 3, 102, 83], [76, 63, 101, 22, 54], [96, 1, 19, 87, 109], [25, 75, 103,
95, 17]], [[13, 62, 125, 34, 81], [122, 40, 8, 86, 58], [113, 26, 78, 93, 10], [70, 79, 73, 48, 47], [6,
108, 31, 53, 121]], [[107, 24, 32, 45, 106], [28, 69, 111, 2, 104], [38, 116, 46, 84, 33], [18, 94, 44,
88, 72], [118, 11, 80, 99, 5]], [[61, 82, 43, 117, 15], [65, 27, 71, 91, 60], [35, 68, 74, 36, 100], [49,
29, 120, 67, 50], [105, 110, 9, 4, 89]], [[115, 20, 23, 97, 57], [12, 119, 123, 42, 7], [52, 39, 21, 77,
124], [85, 112, 41, 37, 30], [55, 16, 90, 64, 98]]]
```

Gambar 4.2.7. Hasil dari algoritma Stochastic Hill-climbing

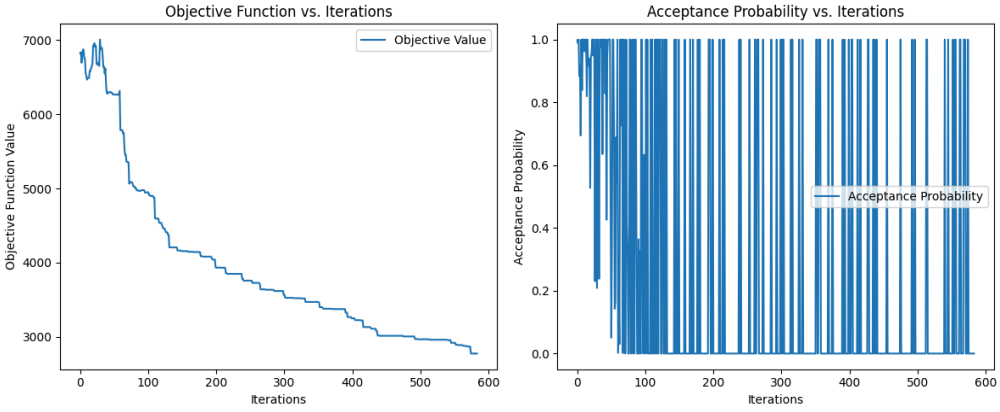


Gambar 4.2.8. Plot dari algoritma Stochastic Hill-climbing

v. Simulated Annealing

```
=== Simulated Annealing Report ===
Initial Objective Value: 6825
Final Objective Value: 2775
Total Iterations: 584
Execution Time: 0.6243 seconds
Frequency of getting stuck in local optima: 435

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

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

Gambar 4.2.9. Hasil dari algoritma Simulated Annealing



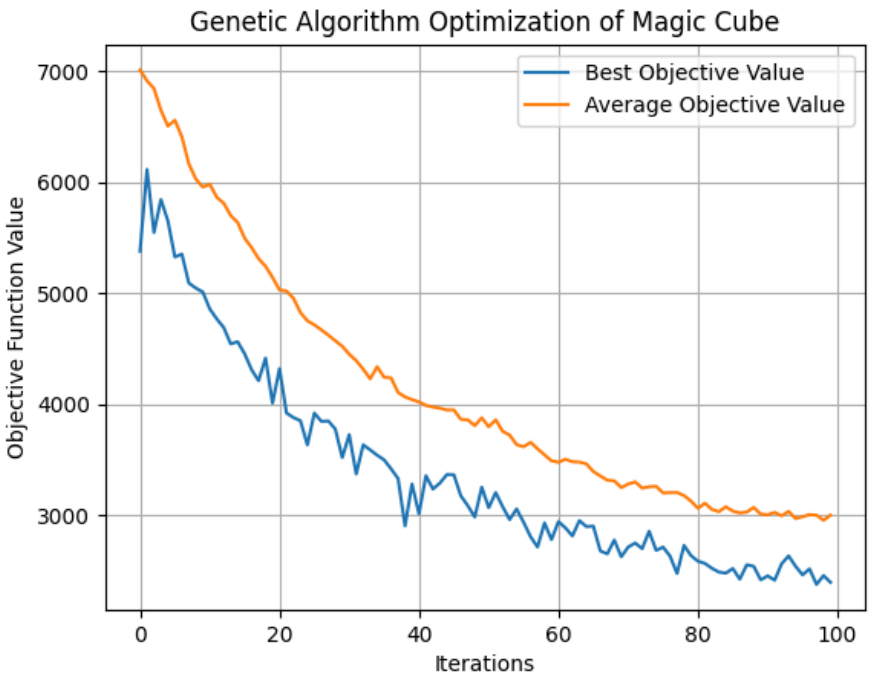
Gambar 4.2.10. Hasil dari algoritma Simulated Annealing

vi. Genetic Algorithm

Pada percobaan pertama dipakai dengan population size 100, iterations 100 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[64, 75, 57, 40, 78], [69, 52, 62, 75, 49], [54, 103, 64, 32, 58], [62, 52, 84, 86, 60], [53, 59, 84,
82, 81]], [[38, 59, 68, 53, 62], [71, 75, 58, 74, 69], [68, 45, 86, 54, 43], [49, 55, 65, 51, 83], [44,
79, 45, 70, 51]], [[69, 39, 70, 50, 82], [94, 45, 48, 62, 61], [71, 76, 70, 66, 85], [74, 79, 43, 68,
70], [38, 79, 50, 45, 67]], [[61, 59, 79, 48, 28], [81, 52, 68, 44, 75], [60, 40, 87, 53, 61], [56, 46,
53, 80, 51], [74, 50, 84, 75, 51]], [[49, 68, 55, 60, 83], [79, 67, 54, 47, 73], [69, 52, 54, 91, 55],
[69, 51, 35, 59, 65], [66, 52, 56, 80, 63]]]

Final Objective Value: 2378
Population Size: 100
Iterations: 100
Duration: 12.62 seconds
```

Gambar 4.2.11. Hasil dari algoritma Genetic Algorithm population size 100 dan iterations 100

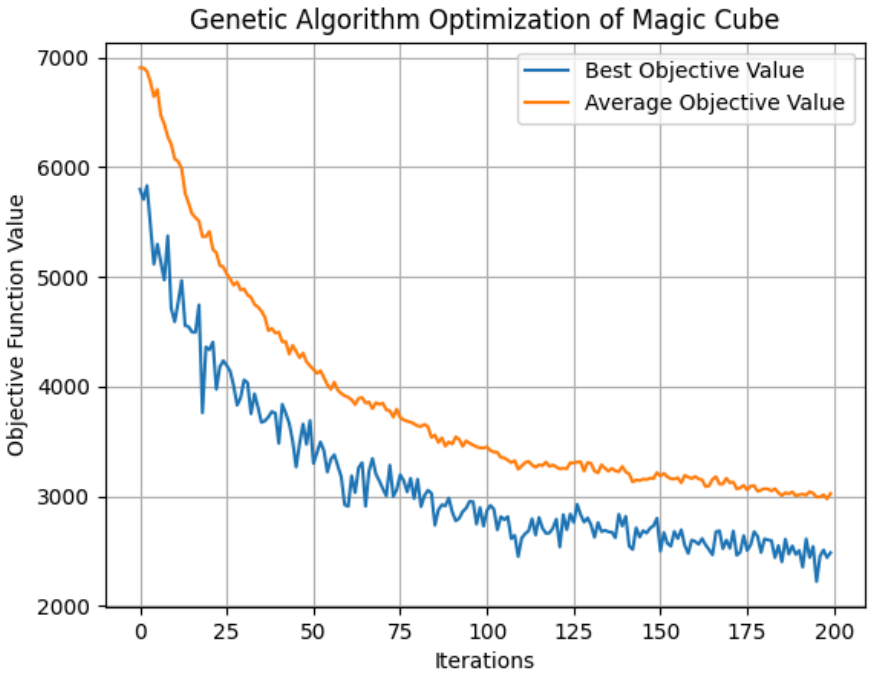


Gambar 4.2.12. Plot dari algoritma Genetic Algorithm population size 100 dan iterations 100

Pada percobaan selanjutnya dipakai dengan population size 100, iterations 200 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[74, 78, 46, 72, 59], [52, 57, 70, 46, 84], [49, 41, 92, 34, 35], [34, 80, 75, 82, 60], [78, 63, 87,
71, 48]], [[86, 49, 66, 29, 77], [56, 67, 75, 92, 42], [78, 61, 42, 89, 64], [79, 74, 65, 44, 77], [48,
54, 64, 66, 51]], [[60, 77, 76, 65, 41], [88, 68, 75, 49, 54], [81, 56, 42, 59, 82], [41, 60, 47, 62,
79], [55, 85, 64, 72, 61]], [[41, 69, 53, 68, 64], [69, 45, 74, 42, 44], [50, 71, 55, 85, 97], [62, 84,
82, 42, 53], [86, 37, 41, 63, 88]], [[54, 53, 89, 84, 58], [47, 76, 60, 75, 53], [66, 77, 62, 47, 44],
[58, 46, 74, 80, 90], [53, 57, 80, 51, 73]]]

Final Objective Value: 2221
Population Size: 100
Iterations: 200
Duration: 20.18 seconds
```

Gambar 4.2.13. Hasil dari algoritma Genetic Algorithm population size 100 dan iterations 200

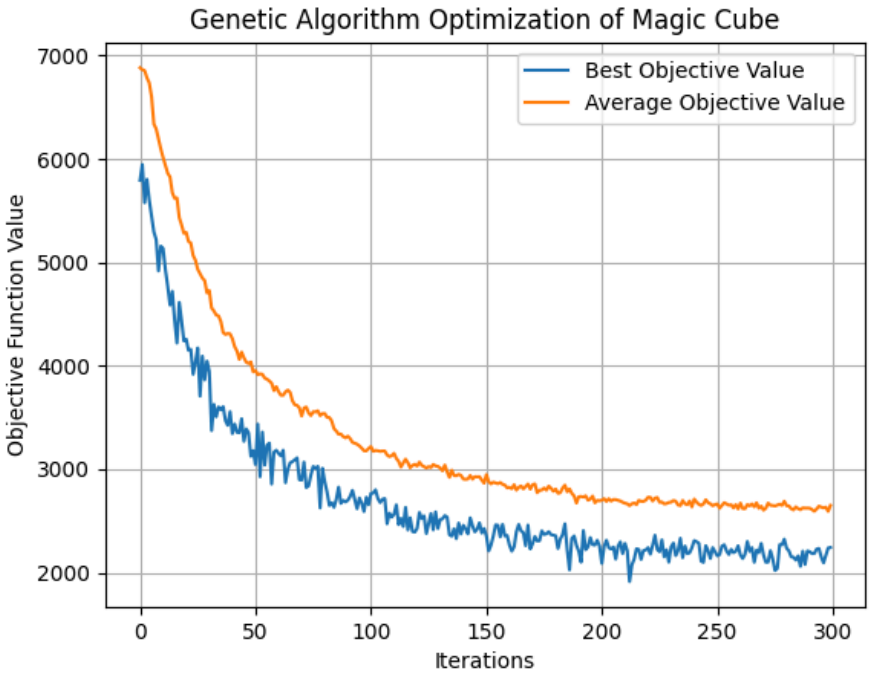


Gambar 4.2.14. Plot dari algoritma Genetic Algorithm population size 100 dan iterations 200

Pada percobaan selanjutnya dipakai dengan population size 100, iterations 300 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[56, 52, 60, 81, 62], [48, 61, 74, 63, 84], [62, 81, 55, 43, 58], [49, 52, 64, 93, 51], [83, 77, 62,
61, 63]], [[48, 87, 68, 41, 49], [44, 71, 57, 76, 67], [48, 45, 56, 47, 78], [81, 57, 50, 46, 60], [62,
50, 94, 43, 75]], [[55, 72, 64, 73, 57], [47, 76, 48, 77, 82], [56, 70, 71, 53, 55], [79, 68, 85, 64,
37], [85, 71, 47, 51, 60]], [[74, 78, 83, 56, 39], [53, 58, 67, 70, 77], [63, 52, 54, 80, 62], [54, 59,
73, 53, 84], [47, 77, 76, 48, 61]], [[51, 50, 49, 74, 68], [61, 43, 82, 56, 78], [72, 43, 76, 68, 49],
[67, 75, 65, 77, 46], [74, 59, 52, 56, 54]]]

Final Objective Value: 1914
Population Size: 100
Iterations: 300
Duration: 49.04 seconds
```

Gambar 4.2.15. Hasil dari algoritma Genetic Algorithm population size 100 dan iterations 300

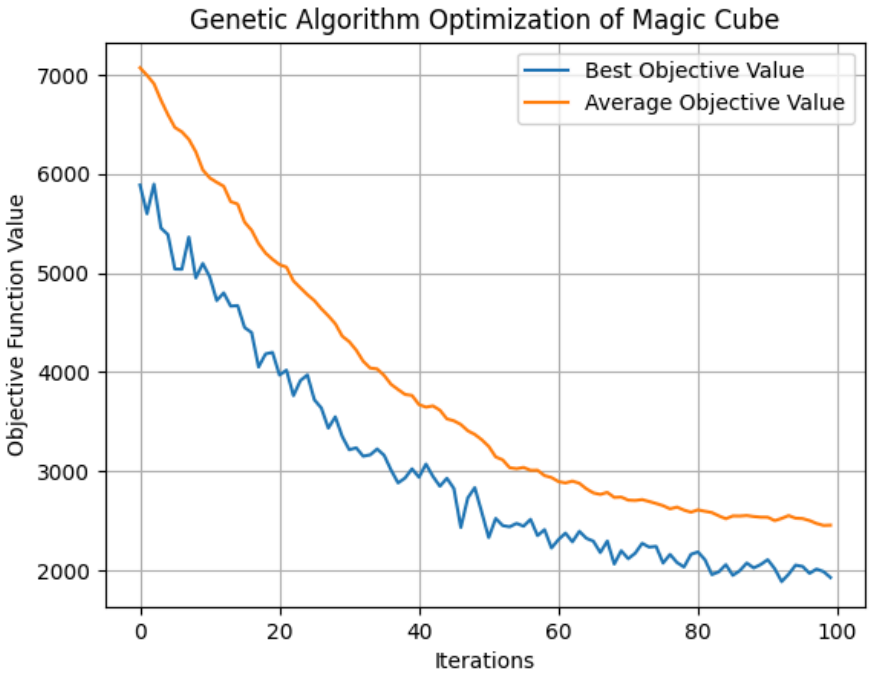


Gambar 4.2.16. Plot dari algoritma Genetic Algorithm population size 100 dan iterations 300

Pada percobaan selanjutnya dipakai dengan population size 200, iterations 100 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[61, 50, 59, 75, 71], [53, 70, 78, 51, 48], [70, 77, 67, 41, 82], [56, 71, 61, 53, 76], [56, 57, 84,
71, 50]], [[59, 71, 45, 63, 65], [46, 54, 74, 45, 51], [57, 65, 69, 61, 64], [79, 57, 45, 64, 77], [65,
85, 61, 94, 83]], [[47, 64, 69, 52, 42], [60, 58, 49, 78, 66], [74, 60, 66, 64, 46], [47, 43, 64, 76,
70], [59, 76, 54, 49, 82]], [[46, 59, 64, 68, 87], [62, 80, 68, 49, 79], [61, 56, 59, 62, 58], [66, 76,
63, 73, 47], [75, 74, 61, 64, 65]], [[54, 80, 38, 46, 74], [70, 50, 80, 48, 54], [50, 65, 72, 76, 55],
[43, 53, 45, 90, 88], [67, 65, 45, 44, 73]]]

Final Objective Value: 1886
Population Size: 200
Iterations: 100
Duration: 23.39 seconds
```

Gambar 4.2.17. Hasil dari algoritma Genetic Algorithm population size 200 dan iterations 100

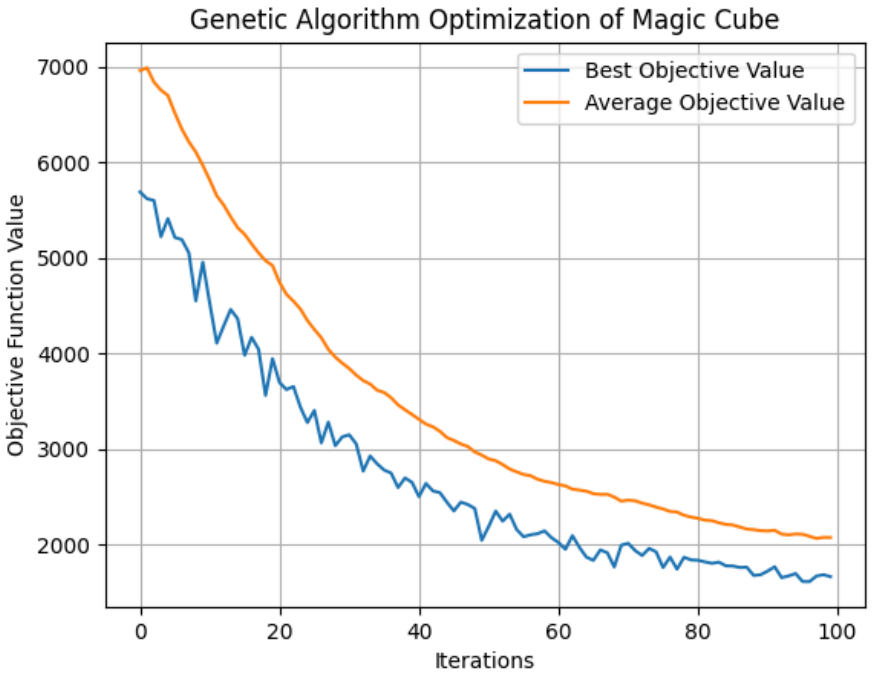


Gambar 4.2.18. Plot dari algoritma Genetic Algorithm population size 200 dan iterations 100

Pada percobaan selanjutnya dipakai dengan population size 300, iterations 100 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[69, 54, 75, 74, 60], [83, 77, 53, 50, 58], [42, 52, 65, 76, 47], [58, 72, 74, 68, 63], [76, 58, 59,
78, 63]], [[75, 70, 53, 54, 64], [59, 45, 66, 47, 54], [73, 80, 56, 41, 67], [67, 62, 78, 75, 55], [42,
51, 59, 62, 68]], [[47, 64, 46, 67, 60], [69, 48, 64, 71, 75], [50, 61, 57, 58, 75], [63, 70, 67, 47,
59], [55, 67, 66, 73, 72]], [[52, 62, 46, 74, 68], [54, 80, 48, 52, 58], [59, 71, 64, 51, 69], [68, 64,
48, 72, 55], [62, 60, 61, 53, 71]], [[55, 72, 63, 61, 60], [69, 47, 82, 49, 73], [78, 48, 67, 56, 62],
[42, 79, 71, 75, 64], [65, 67, 64, 77, 70]]]

Final Objective Value: 1612
Population Size: 300
Iterations: 100
Duration: 33.51 seconds
```

Gambar 4.2.19. Hasil dari algoritma Genetic Algorithm population size 300 dan iterations 100



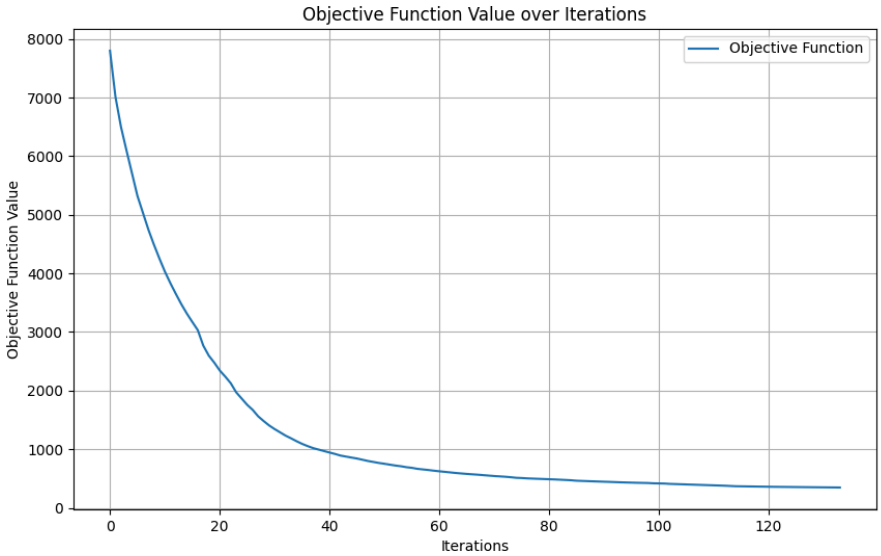
Gambar 4.2.20. Plot dari algoritma Genetic Algorithm population size 300 dan iterations 100

c. Eksperimen 3

i. Steepest Ascent Hill-climbing

```
Experiment Report:
Initial State: [[[27, 59, 73, 118, 67], [121, 72, 89, 25, 1], [68, 9, 33, 63, 55], [51, 13, 93, 119,
84], [49, 100, 82, 78, 10]], [[23, 47, 43, 5, 34], [103, 54, 86, 22, 2], [90, 123, 116, 105, 74], [77,
37, 120, 46, 30], [70, 17, 85, 16, 113]], [[98, 39, 26, 52, 114], [58, 112, 81, 79, 53], [94, 108, 110,
64, 109], [107, 4, 29, 111, 104], [35, 44, 31, 41, 19]], [[40, 7, 101, 45, 36], [80, 62, 122, 96, 88],
[83, 42, 12, 106, 66], [14, 15, 91, 75, 97], [102, 65, 76, 3, 20]], [[125, 115, 69, 92, 117], [32, 56,
38, 28, 6], [24, 99, 18, 124, 95], [11, 87, 48, 60, 61], [8, 21, 71, 50, 57]]]
Final State: [[[34, 58, 74, 116, 32], [68, 72, 88, 30, 56], [121, 9, 92, 1, 100], [28, 75, 12, 93,
107], [64, 101, 50, 78, 20]], [[24, 124, 51, 35, 73], [105, 22, 21, 60, 106], [89, 11, 120, 87, 8],
[81, 44, 118, 47, 25], [17, 114, 5, 85, 103]], [[96, 29, 27, 52, 111], [39, 108, 79, 84, 7], [6, 115,
61, 26, 109], [123, 18, 57, 41, 77], [49, 45, 90, 112, 10]], [[40, 102, 94, 46, 33], [42, 59, 15, 98,
97], [83, 53, 23, 91, 67], [70, 82, 86, 76, 3], [80, 16, 99, 4, 117]], [[122, 2, 69, 66, 63], [62, 55,
113, 36, 48], [14, 125, 19, 119, 31], [13, 95, 43, 54, 110], [104, 38, 71, 37, 65]]]
Final Objective Value: 347
Total Iterations: 133
Duration: 126.6734 seconds
```

Gambar 4.3.1 Hasil dari algoritma Steepest Ascent Hill-climbing

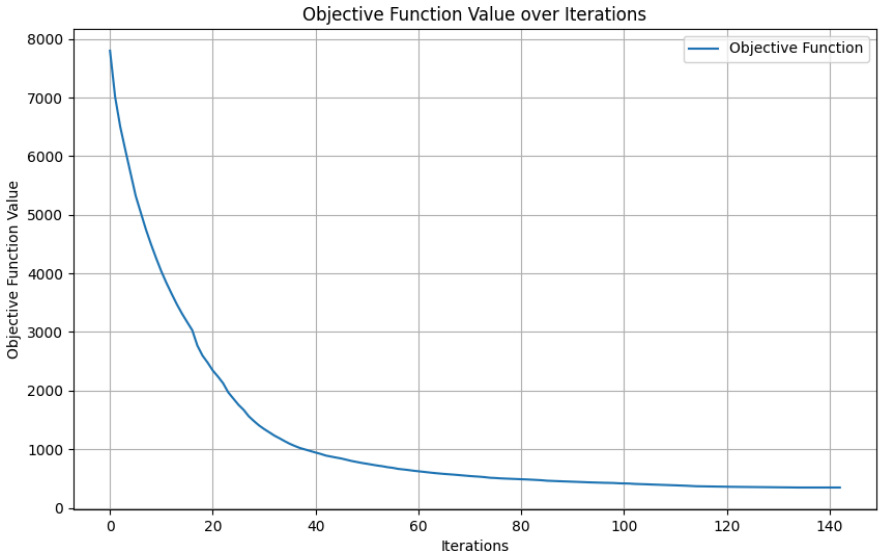


Gambar 4.3.2. Plot dari algoritma Steepest Ascent Hill-climbing

ii. Hill-climbing with Sideways Move

```
Experiment Report:
Initial State:
[[[27, 59, 73, 118, 67], [121, 72, 89, 25, 1], [68, 9, 33, 63, 55], [51, 13, 93, 119, 84], [49, 100,
82, 78, 10]], [[23, 47, 43, 5, 34], [103, 54, 86, 22, 2], [90, 123, 116, 105, 74], [77, 37, 120, 46,
30], [70, 17, 85, 16, 113]], [[98, 39, 26, 52, 114], [58, 112, 81, 79, 53], [94, 108, 110, 64, 109],
[107, 4, 29, 111, 104], [35, 44, 31, 41, 19]], [[40, 7, 101, 45, 36], [80, 62, 122, 96, 88], [83, 42,
12, 106, 66], [14, 15, 91, 75, 97], [102, 65, 76, 3, 20]], [[125, 115, 69, 92, 117], [32, 56, 38, 28,
6], [24, 99, 18, 124, 95], [11, 87, 48, 60, 61], [8, 21, 71, 50, 57]]]
Final State:
[[[34, 58, 74, 116, 32], [68, 72, 88, 30, 56], [121, 9, 92, 1, 100], [28, 75, 12, 93, 107], [64, 101,
50, 78, 20]], [[24, 124, 51, 35, 73], [105, 22, 21, 60, 106], [89, 11, 120, 87, 8], [81, 44, 118, 47,
25], [17, 114, 5, 85, 103]], [[96, 29, 27, 52, 111], [39, 108, 79, 84, 7], [6, 115, 61, 26, 109], [123,
18, 57, 41, 77], [49, 45, 90, 112, 10]], [[40, 102, 94, 46, 33], [42, 59, 15, 98, 97], [83, 53, 23, 91,
67], [70, 82, 86, 76, 3], [80, 16, 99, 4, 117]], [[122, 2, 69, 66, 63], [62, 55, 113, 36, 48], [14,
125, 19, 119, 31], [13, 95, 43, 54, 110], [104, 38, 71, 37, 65]]]
Final Objective Value: 347
Total Iterations: 143
Total Sideways Moves: 10
Duration: 137.1795 seconds
```

Gambar 4.3.3. Hasil dari algoritma Hill-climbing with Sideways Move



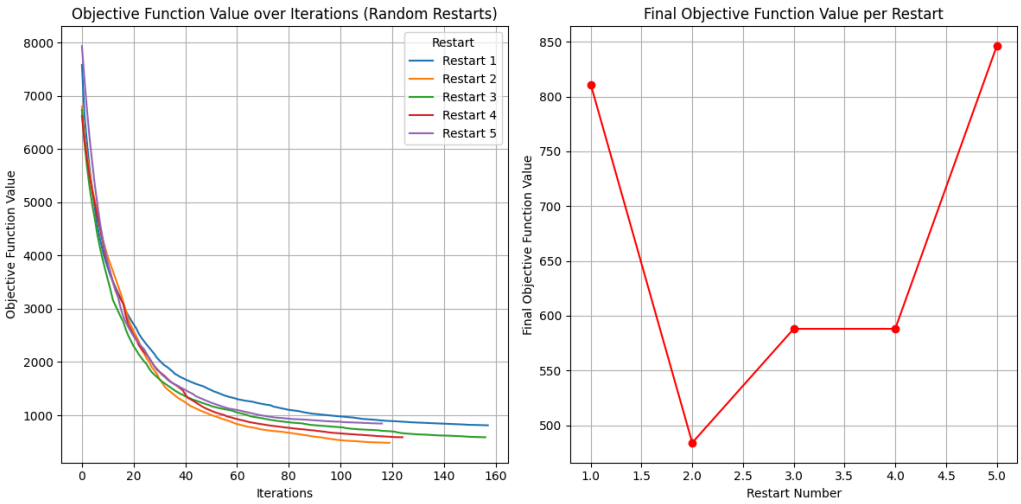
Gambar 4.3.4. Plot dari algoritma Hill-climbing with Sideways Move

iii. Random Restart Hill-climbing

```
Experiment Report:

Restart 1/5:
Initial State: [[[109, 85, 121, 11, 27], [114, 30, 43, 78, 118], [24, 66, 31, 73, 14], [9, 88, 41, 69,
107], [5, 71, 19, 70, 45]], [[15, 54, 46, 101, 53], [110, 119, 92, 125, 20], [4, 56, 12, 48, 103], [47,
34, 95, 35, 33], [68, 64, 7, 22, 37]], [[8, 111, 90, 21, 1], [82, 63, 44, 115, 17], [23, 49, 105, 99,
93], [97, 38, 84, 36, 81], [79, 57, 52, 67, 32]], [[87, 108, 61, 28, 76], [18, 100, 80, 40, 59], [102,
60, 86, 2, 96], [122, 98, 94, 77, 42], [117, 58, 75, 16, 89]], [[113, 10, 123, 65, 25], [83, 29, 26,
50, 124], [39, 3, 74, 91, 55], [112, 116, 106, 13, 120], [104, 62, 72, 51, 6]]]
Final State: [[[108, 80, 81, 11, 34], [25, 6, 113, 77, 91], [88, 73, 37, 103, 14], [4, 74, 31, 106,
109], [90, 83, 52, 20, 70]], [[18, 41, 54, 100, 102], [9, 119, 116, 63, 7], [111, 36, 12, 84, 72],
[101, 62, 95, 44, 13], [76, 57, 38, 23, 121]], [[27, 120, 93, 29, 46], [123, 65, 3, 105, 19], [2, 47,
82, 58, 124], [96, 16, 85, 79, 39], [66, 67, 49, 45, 87]], [[53, 43, 55, 50, 114], [35, 92, 59, 51,
78], [97, 64, 86, 28, 40], [122, 56, 5, 71, 61], [8, 60, 110, 115, 22]], [[104, 30, 32, 125, 24], [117,
33, 26, 21, 118], [17, 98, 89, 42, 69], [1, 107, 99, 15, 94], [75, 48, 68, 112, 10]]]
Final Objective Value: 811
Iterations: 157
Duration: 151.0433 seconds

Restart 2/5:
Initial State: [[[39, 109, 66, 84, 7], [14, 121, 77, 32, 16], [23, 97, 90, 94, 36], [86, 78, 55, 6,
57], [88, 87, 74, 118, 48]], [[21, 114, 82, 115, 63], [35, 11, 58, 72, 13], [71, 102, 73, 45, 17], [64,
27, 107, 80, 62], [52, 47, 106, 69, 117]], [[31, 42, 96, 54, 51], [120, 105, 104, 8, 124], [59, 44, 15,
46, 61], [53, 70, 2, 12, 38], [81, 75, 10, 119, 56]], [[19, 34, 25, 1, 95], [111, 18, 85, 20, 76], [26,
49, 116, 100, 4], [79, 24, 112, 98, 67], [37, 28, 122, 41, 125]], [[65, 60, 93, 68, 113], [33, 92, 29,
91, 110], [103, 89, 30, 50, 123], [108, 101, 3, 9, 83], [5, 43, 40, 99, 22]]]
Final State: [[[39, 104, 93, 68, 11], [70, 114, 66, 38, 28], [22, 1, 92, 86, 113], [90, 69, 48, 4,
105], [94, 31, 18, 117, 56]], [[34, 103, 5, 108, 63], [77, 24, 35, 121, 64], [125, 99, 67, 2, 16], [15,
47, 111, 80, 62], [52, 42, 97, 14, 110]], [[79, 10, 96, 58, 72], [54, 26, 109, 8, 118], [55, 116, 71,
12, 61], [53, 88, 3, 120, 45], [78, 75, 36, 112, 19]], [[98, 37, 25, 33, 123], [101, 44, 20, 57, 89],
[51, 81, 76, 100, 7], [59, 27, 74, 82, 73], [6, 124, 122, 40, 23]], [[65, 60, 95, 49, 46], [13, 107,
87, 91, 17], [50, 21, 9, 115, 119], [102, 84, 83, 29, 30], [85, 43, 41, 32, 106]]]
Final Objective Value: 484
Iterations: 119
Duration: 114.7743 seconds

Restart 3/5:
Initial State: [[[55, 35, 34, 13, 50], [89, 57, 8, 123, 63], [20, 19, 125, 38, 114], [54, 76, 56, 40,
78], [116, 77, 117, 52, 9]], [[110, 118, 85, 12, 94], [113, 15, 5, 43, 88], [105, 22, 124, 39, 16],
[10, 24, 71, 70, 98], [93, 3, 33, 31, 27]], [[62, 87, 84, 120, 45], [107, 112, 23, 67, 108], [86, 49,
83, 32, 109], [92, 21, 18, 42, 74], [101, 28, 122, 91, 79]], [[80, 37, 53, 51, 7], [14, 100, 25, 75,
115], [119, 66, 29, 69, 121], [64, 103, 58, 60, 11], [96, 99, 102, 4, 6]], [[65, 46, 111, 26, 81], [59,
2, 47, 41, 68], [82, 44, 17, 48, 104], [30, 90, 61, 1, 72], [95, 97, 106, 36, 73]]]
Final State: [[[62, 39, 8, 110, 96], [48, 82, 123, 1, 61], [14, 25, 121, 103, 54], [78, 52, 57, 37,
92], [113, 115, 6, 70, 11]], [[7, 122, 81, 24, 94], [106, 50, 55, 63, 41], [99, 100, 66, 46, 4], [38,
27, 104, 72, 58], [65, 10, 9, 112, 119]], [[13, 79, 36, 120, 67], [101, 114, 45, 53, 2], [64, 56, 76,
35, 84], [108, 44, 69, 21, 74], [29, 22, 89, 86, 91]], [[118, 32, 95, 51, 18], [3, 73, 30, 85, 124],
[33, 34, 31, 83, 125], [68, 90, 60, 77, 20], [93, 88, 98, 19, 16]], [[111, 43, 109, 12, 40], [59, 5,
47, 117, 87], [105, 102, 17, 49, 42], [23, 97, 26, 107, 71], [15, 80, 116, 28, 75]]]
Final Objective Value: 588
Iterations: 156
Duration: 152.6317 seconds
```

```
Restart 4/5:
Initial State: [[[24, 41, 12, 3, 62], [120, 56, 71, 116, 122], [15, 90, 46, 34, 6], [32, 103, 86, 92,
67], [109, 48, 35, 97, 31]], [[70, 110, 4, 83, 119], [106, 37, 10, 89, 27], [101, 61, 2, 25, 85], [55,
36, 114, 66, 39], [82, 8, 105, 58, 9]], [[22, 63, 123, 125, 75], [49, 7, 26, 88, 59], [47, 107, 14, 84,
121], [42, 44, 108, 79, 13], [51, 23, 18, 95, 81]], [[93, 113, 80, 16, 33], [78, 111, 112, 11, 64],
[73, 100, 38, 124, 45], [28, 115, 57, 52, 94], [99, 68, 53, 118, 21]], [[65, 20, 104, 54, 17], [87, 5,
98, 69, 29], [96, 117, 30, 1, 102], [50, 74, 60, 72, 43], [40, 76, 19, 77, 91]]]
Final State: [[[33, 75, 120, 55, 31], [122, 42, 39, 7, 105], [11, 88, 113, 78, 25], [47, 56, 13, 92,
119], [102, 54, 30, 94, 35]], [[66, 86, 3, 48, 112], [111, 83, 10, 97, 14], [76, 51, 81, 20, 85], [60,
71, 107, 27, 45], [1, 21, 114, 121, 58]], [[22, 63, 98, 123, 9], [49, 80, 26, 68, 95], [52, 24, 62, 87,
90], [115, 44, 108, 36, 12], [74, 103, 19, 2, 116]], [[93, 38, 89, 16, 79], [4, 104, 109, 32, 61], [70,
34, 41, 124, 46], [57, 67, 23, 59, 110], [96, 72, 53, 82, 15]], [[100, 50, 8, 73, 84], [28, 5, 125,
117, 40], [106, 118, 18, 6, 69], [37, 77, 64, 101, 29], [43, 65, 99, 17, 91]]]
Final Objective Value: 588
Iterations: 124
Duration: 118.7912 seconds

Restart 5/5:
Initial State: [[[111, 27, 68, 56, 33], [57, 8, 69, 83, 72], [2, 7, 97, 74, 17], [50, 102, 84, 11, 5],
[46, 121, 32, 64, 90]], [[89, 25, 70, 52, 98], [54, 61, 47, 106, 116], [109, 37, 107, 71, 108], [78,
91, 26, 92, 77], [113, 85, 44, 22, 79]], [[13, 14, 82, 87, 51], [118, 4, 123, 105, 81], [114, 73, 62,
122, 124], [94, 86, 95, 34, 119], [99, 45, 112, 96, 103]], [[12, 24, 58, 40, 16], [15, 125, 43, 93,
88], [63, 38, 1, 3, 30], [20, 60, 21, 110, 31], [104, 48, 9, 41, 28]], [[19, 6, 115, 117, 23], [101,
65, 75, 67, 36], [35, 18, 55, 53, 100], [120, 10, 29, 66, 39], [42, 76, 49, 80, 59]]]
Final State: [[[120, 18, 9, 87, 83], [89, 6, 69, 79, 72], [7, 123, 76, 70, 39], [51, 102, 125, 14, 23],
[46, 66, 36, 65, 99]], [[49, 60, 74, 50, 98], [19, 58, 84, 29, 116], [97, 24, 71, 112, 11], [40, 82,
27, 104, 62], [113, 91, 59, 22, 28]], [[13, 122, 56, 81, 43], [101, 52, 44, 110, 8], [114, 73, 96, 1,
30], [85, 67, 12, 34, 117], [3, 2, 106, 90, 118]], [[107, 100, 61, 10, 37], [15, 78, 41, 93, 88], [63,
32, 17, 80, 124], [20, 57, 105, 95, 38], [111, 48, 94, 33, 25]], [[26, 16, 115, 86, 54], [92, 121, 77,
4, 31], [35, 64, 55, 53, 108], [119, 5, 47, 68, 75], [42, 109, 21, 103, 45]]]
Final Objective Value: 846
Iterations: 116
Duration: 111.1931 seconds

Best Overall Solution:
Final Objective Value: 484
Best Final Cube State: [[[39, 104, 93, 68, 11], [70, 114, 66, 38, 28], [22, 1, 92, 86, 113], [90, 69,
48, 4, 105], [94, 31, 18, 117, 56]], [[34, 103, 5, 108, 63], [77, 24, 35, 121, 64], [125, 99, 67, 2,
16], [15, 47, 111, 80, 62], [52, 42, 97, 14, 110]], [[79, 10, 96, 58, 72], [54, 26, 109, 8, 118], [55,
116, 71, 12, 61], [53, 88, 3, 120, 45], [78, 75, 36, 112, 19]], [[98, 37, 25, 33, 123], [101, 44, 20,
57, 89], [51, 81, 76, 100, 7], [59, 27, 74, 82, 73], [6, 124, 122, 40, 23]], [[65, 60, 95, 49, 46],
[13, 107, 87, 91, 17], [50, 21, 9, 115, 119], [102, 84, 83, 29, 30], [85, 43, 41, 32, 106]]]
Total Iterations Across All Restarts: 672
Total Duration (all restarts): 648.4336 seconds
```

Gambar 4.3.5. Hasil dari algoritma Random Restart Hill-climbing

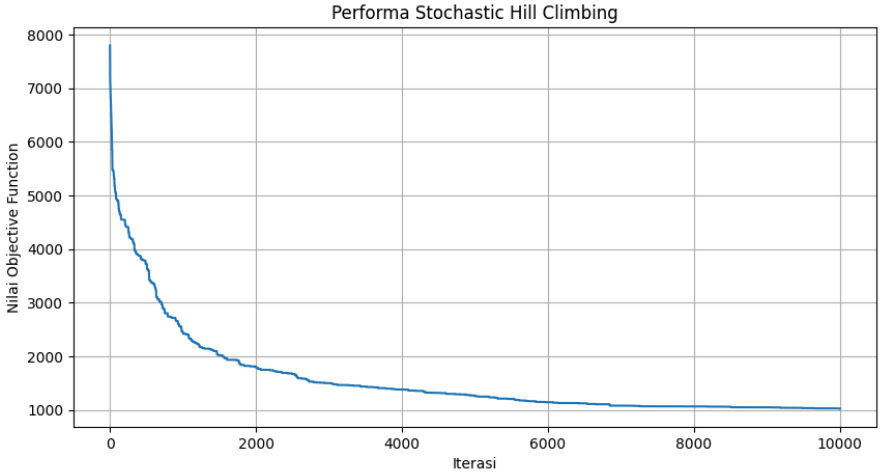


Gambar 4.3.6. Plot dari algoritma Random Restart Hill-climbing

iv. Stochastic Hill-climbing

```
===== Laporan Hasil Stochastic Hill Climbing =====
Durasi Pencarian       : 1.3110 detik
Total Iterasi          : 10000
Nilai Objective Awal   : 7800
Initial State:
[[[27, 59, 73, 118, 67], [121, 72, 89, 25, 1], [68, 9, 33, 63, 55], [51, 13, 93, 119, 84], [49, 100,
82, 78, 10]], [[23, 47, 43, 5, 34], [103, 54, 86, 22, 2], [90, 123, 116, 105, 74], [77, 37, 120, 46,
30], [70, 17, 85, 16, 113]], [[98, 39, 26, 52, 114], [58, 112, 81, 79, 53], [94, 108, 110, 64, 109],
[107, 4, 29, 111, 104], [35, 44, 31, 41, 19]], [[40, 7, 101, 45, 36], [80, 62, 122, 96, 88], [83, 42,
12, 106, 66], [14, 15, 91, 75, 97], [102, 65, 76, 3, 20]], [[125, 115, 69, 92, 117], [32, 56, 38, 28,
6], [24, 99, 18, 124, 95], [11, 87, 48, 60, 61], [8, 21, 71, 50, 57]]]
Nilai Objective Akhir  : 1024
Final State:
[[[46, 61, 25, 105, 80], [98, 24, 4, 78, 116], [15, 107, 117, 6, 60], [123, 2, 91, 87, 42], [34, 122,
92, 41, 18]], [[10, 115, 106, 3, 86], [64, 104, 70, 71, 5], [114, 28, 30, 110, 32], [44, 45, 108, 51,
65], [84, 37, 1, 74, 119]], [[111, 14, 31, 101, 59], [82, 56, 73, 47, 57], [66, 48, 85, 26, 89], [40,
94, 38, 43, 102], [17, 103, 88, 99, 8]], [[36, 95, 124, 16, 27], [52, 49, 20, 97, 96], [113, 53, 75,
58, 21], [39, 83, 23, 79, 93], [72, 35, 63, 69, 77]], [[120, 29, 12, 90, 62], [11, 81, 125, 22, 67],
[7, 68, 9, 109, 121], [76, 118, 55, 54, 13], [100, 19, 112, 33, 50]]]
```

Gambar 4.3.7. Hasil dari algoritma Stochastic Hill-climbing

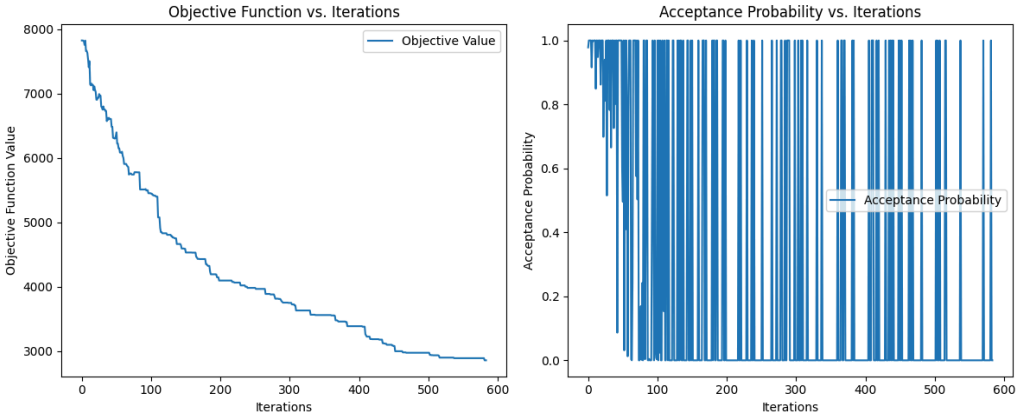


Gambar 4.2.8. Plot dari algoritma Stochastic Hill-climbing

v. Simulated Annealing

```
=== Simulated Annealing Report ===
Initial Objective Value: 7822
Final Objective Value: 2855
Total Iterations: 584
Execution Time: 0.1698 seconds
Frequency of getting stuck in local optima: 428

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

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

Gambar 4.3.9. Hasil dari algoritma Simulated Annealing



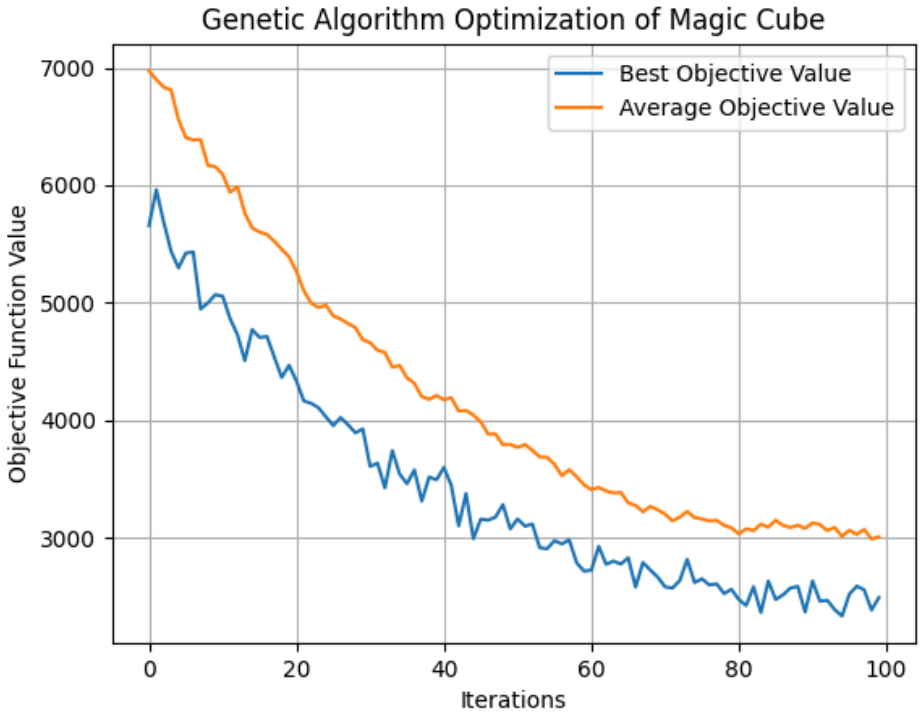
Gambar 4.3.10. Hasil dari algoritma Simulated Annealing

vi. Genetic Algorithm

Pada percobaan pertama dipakai dengan population size 100, iterations 100 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[55, 74, 58, 39, 62], [66, 34, 63, 88, 60], [57, 75, 71, 44, 52], [82, 42, 57, 83, 65], [69, 82, 50,
47, 64]], [[59, 45, 37, 78, 95], [46, 53, 77, 71, 54], [36, 72, 84, 99, 67], [74, 30, 60, 49, 67], [71,
84, 53, 52, 77]], [[53, 69, 88, 67, 42], [87, 82, 47, 45, 39], [50, 47, 56, 70, 84], [46, 87, 79, 43,
75], [68, 80, 41, 43, 52]], [[89, 43, 90, 66, 50], [58, 42, 62, 57, 100], [84, 62, 44, 81, 39], [73,
46, 63, 76, 78], [55, 71, 77, 72, 44]], [[69, 82, 45, 68, 73], [65, 43, 99, 77, 44], [91, 59, 56, 78,
65], [62, 80, 55, 81, 50], [67, 54, 62, 51, 63]]]

Final Objective Value: 2339
Population Size: 100
Iterations: 100
Duration: 7.43 seconds
```

Gambar 4.3.11. Hasil dari algoritma Genetic Algorithm population size 100 dan iterations 100

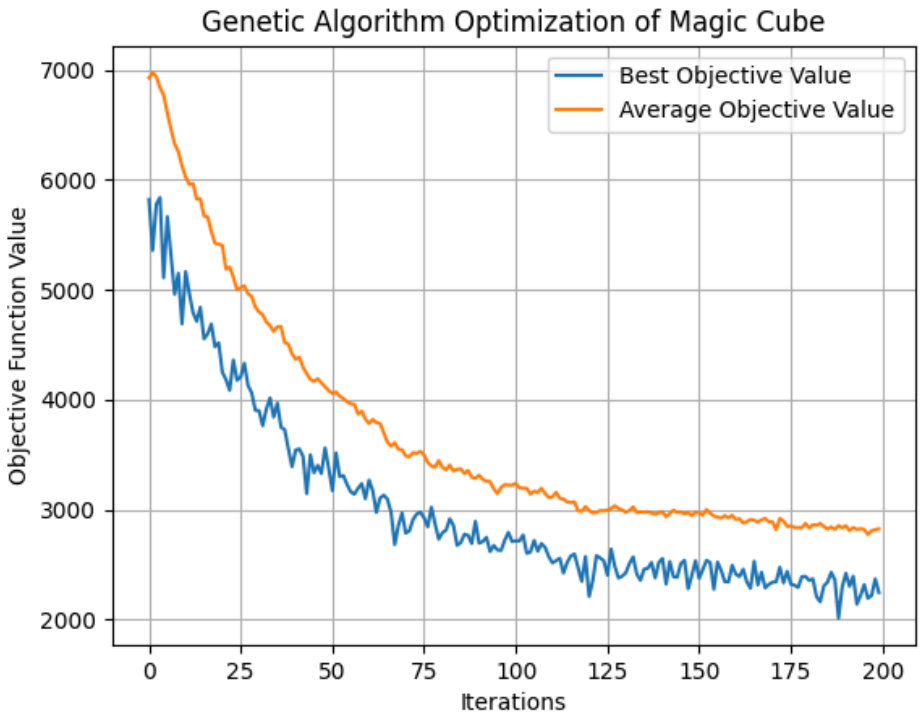


Gambar 4.3.12. Plot dari algoritma Genetic Algorithm population size 100 dan iterations 100

Pada percobaan selanjutnya dipakai dengan population size 100, iterations 200 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[70, 75, 56, 63, 68], [42, 51, 91, 69, 73], [51, 62, 66, 49, 77], [90, 43, 41, 70, 77], [76, 49, 68,
51, 34]], [[46, 48, 91, 51, 54], [89, 71, 41, 44, 68], [53, 49, 46, 90, 70], [35, 75, 88, 61, 52], [56,
51, 76, 68, 62]], [[55, 74, 44, 60, 80], [47, 58, 70, 73, 60], [68, 66, 73, 75, 76], [77, 34, 78, 80,
49], [75, 86, 37, 72, 76]], [[57, 68, 38, 54, 70], [80, 52, 66, 62, 79], [81, 77, 78, 47, 35], [58, 59,
66, 45, 71], [36, 76, 82, 64, 73]], [[80, 61, 65, 81, 66], [65, 61, 79, 60, 64], [43, 69, 55, 62, 83],
[92, 37, 40, 77, 57], [69, 47, 87, 61, 62]]]

Final Objective Value: 2013
Population Size: 100
Iterations: 200
Duration: 14.76 seconds
```

Gambar 4.3.13. Hasil dari algoritma Genetic Algorithm population size 100 dan iterations 200

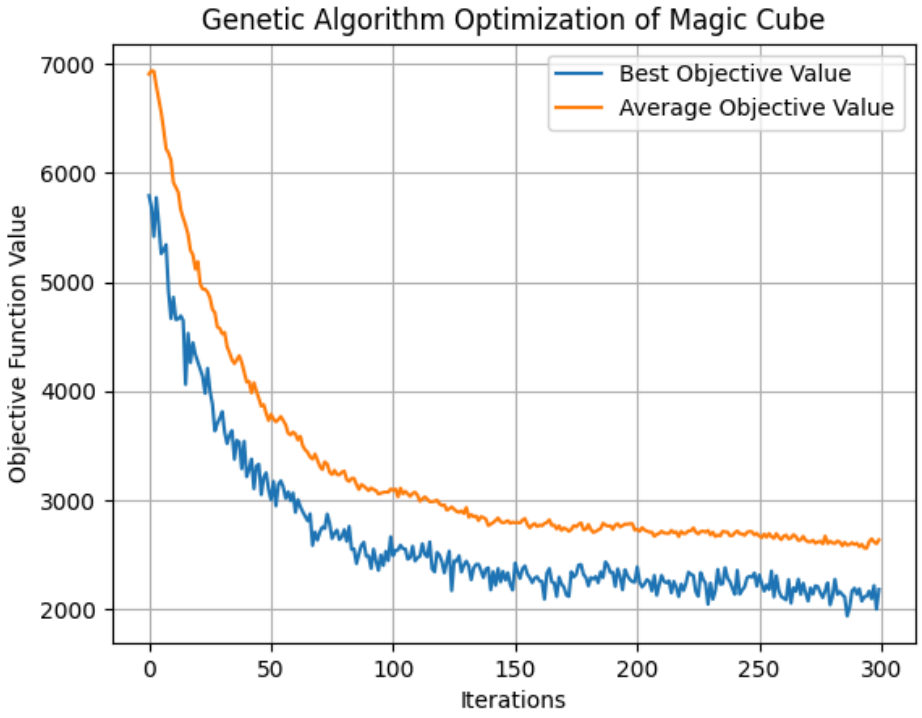


Gambar 4.3.14. Plot dari algoritma Genetic Algorithm population size 100 dan iterations 200

Pada percobaan selanjutnya dipakai dengan population size 100, iterations 300 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[60, 71, 44, 79, 51], [60, 49, 64, 87, 61], [66, 76, 53, 67, 87], [83, 60, 65, 58, 48], [71, 49, 76,
48, 65]], [[68, 54, 81, 49, 86], [75, 45, 67, 62, 77], [55, 79, 85, 70, 78], [66, 73, 45, 59, 49], [66,
67, 46, 75, 63]], [[68, 62, 73, 83, 56], [59, 85, 61, 95, 41], [62, 44, 69, 59, 42], [43, 59, 65, 61,
79], [71, 61, 70, 38, 57]], [[65, 67, 69, 59, 72], [50, 93, 49, 68, 51], [80, 32, 39, 60, 96], [45, 75,
52, 64, 43], [72, 49, 76, 55, 47]], [[40, 58, 77, 60, 55], [37, 71, 67, 82, 85], [68, 56, 61, 82, 52],
[73, 47, 69, 61, 53], [68, 65, 48, 77, 66]]]

Final Objective Value: 1942
Population Size: 100
Iterations: 300
Duration: 22.24 seconds
```

Gambar 4.3.15. Hasil dari algoritma Genetic Algorithm population size 100 dan iterations 300

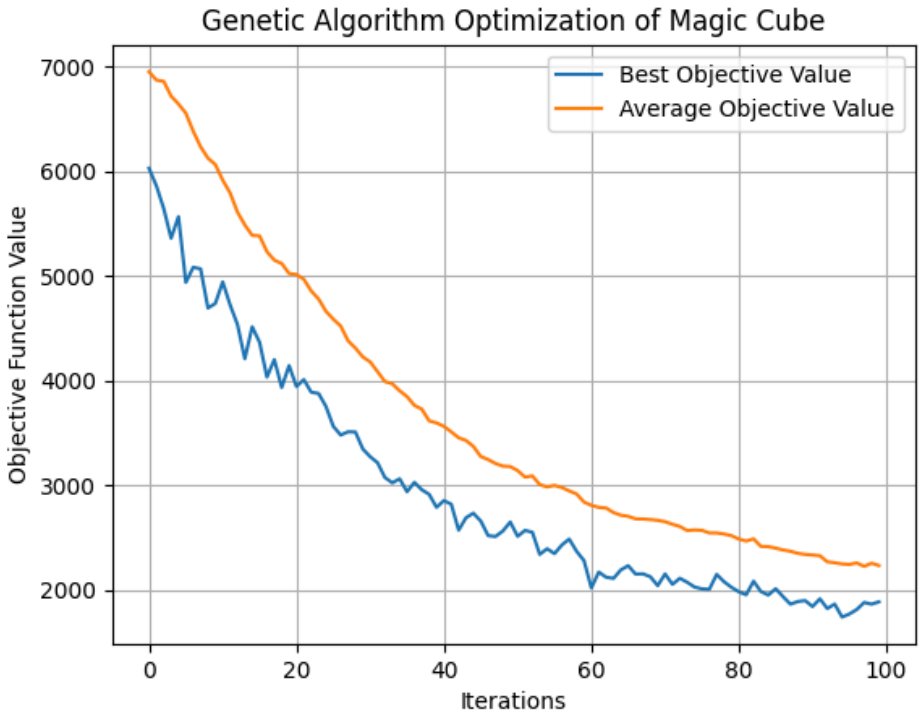


Gambar 4.3.16. Plot dari algoritma Genetic Algorithm population size 100 dan iterations 300

Pada percobaan selanjutnya dipakai dengan population size 200, iterations 100 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[57, 63, 73, 55, 60], [53, 61, 43, 49, 50], [54, 45, 69, 51, 79], [62, 58, 63, 50, 90], [75, 52, 77,
61, 58]], [[61, 78, 38, 79, 71], [53, 74, 57, 49, 65], [62, 53, 70, 67, 63], [57, 76, 50, 42, 53], [58,
62, 63, 66, 68]], [[70, 61, 59, 82, 39], [80, 42, 70, 58, 44], [77, 52, 64, 70, 71], [60, 57, 65, 67,
80], [68, 89, 44, 63, 72]], [[51, 55, 57, 64, 72], [67, 72, 66, 48, 60], [42, 76, 71, 84, 43], [77, 62,
58, 60, 83], [51, 71, 67, 78, 53]], [[79, 65, 53, 59, 55], [51, 77, 89, 58, 67], [62, 80, 45, 88, 78],
[69, 63, 73, 56, 52], [63, 52, 50, 61, 65]]]

Final Objective Value: 1745
Population Size: 200
Iterations: 100
Duration: 15.01 seconds
```

Gambar 4.3.17. Hasil dari algoritma Genetic Algorithm population size 200 dan iterations 100

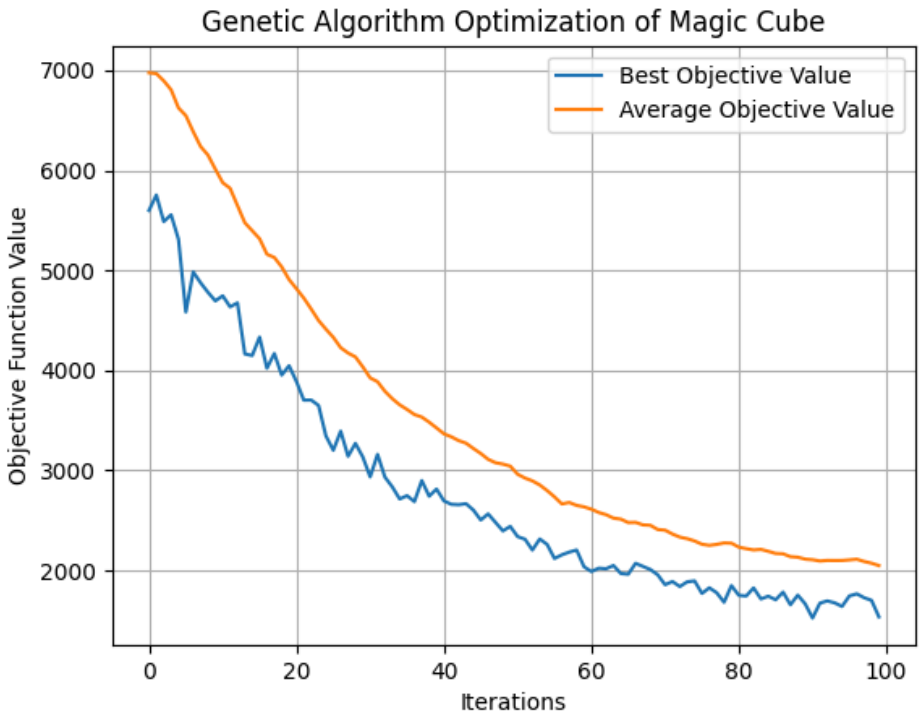


Gambar 4.3.18. Plot dari algoritma Genetic Algorithm population size 200 dan iterations 100

Pada percobaan selanjutnya dipakai dengan population size 300, iterations 100 dan mutation rate 0.01. Hasilnya sebagai berikut.

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

Final State:
[[[70, 52, 66, 57, 84], [68, 57, 55, 76, 51], [58, 49, 61, 74, 71], [50, 48, 58, 70, 82], [51, 73, 65,
52, 66]], [[62, 74, 54, 51, 70], [59, 73, 57, 77, 51], [52, 61, 70, 71, 62], [55, 62, 60, 66, 49], [76,
78, 54, 74, 70]], [[61, 64, 56, 72, 46], [63, 65, 44, 81, 78], [78, 58, 68, 44, 77], [40, 83, 69, 68,
49], [63, 51, 68, 59, 48]], [[52, 58, 68, 59, 66], [59, 58, 72, 67, 50], [70, 83, 49, 59, 66], [68, 58,
63, 62, 54], [66, 54, 59, 82, 68]], [[67, 52, 64, 74, 49], [82, 59, 50, 62, 56], [60, 79, 63, 66, 62],
[59, 60, 73, 55, 81], [64, 60, 77, 47, 76]]]

Final Objective Value: 1525
Population Size: 300
Iterations: 100
Duration: 22.51 seconds
```

Gambar 4.3.19. Hasil dari algoritma Genetic Algorithm population size 300 dan iterations 100

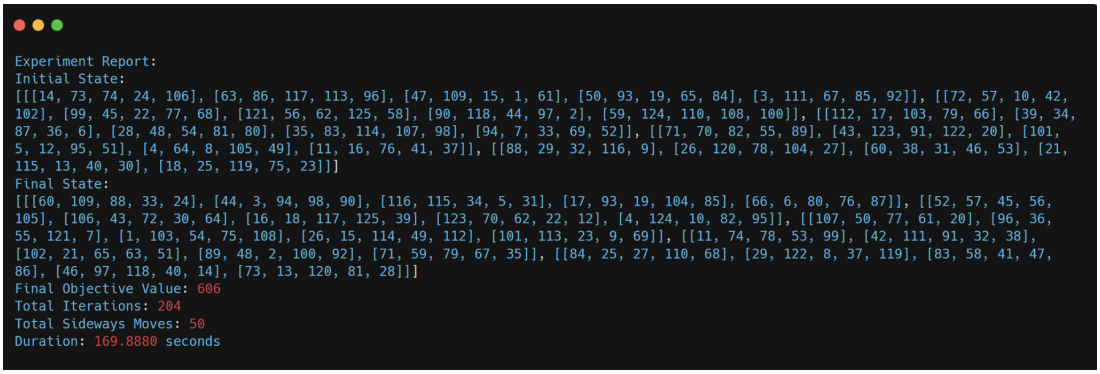


Gambar 4.3.20. Plot dari algoritma Genetic Algorithm population size 300 dan iterations 100

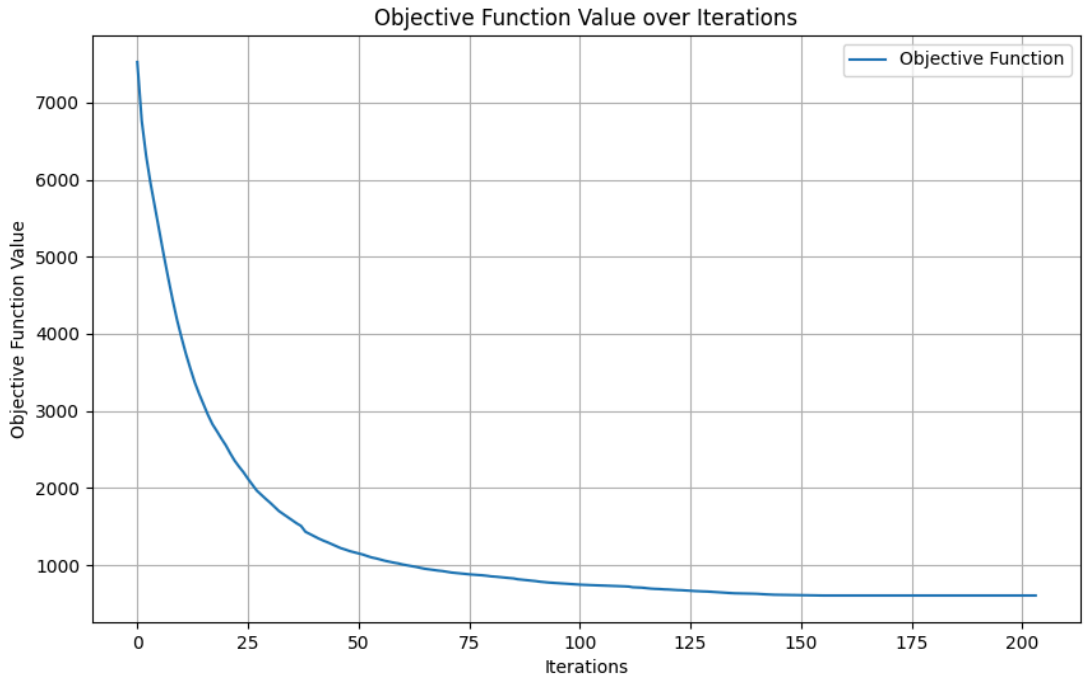
d. Eksperimen 4

Kemudian kami melakukan 1 eksperimen tambahan, lanjutan dari eksperimen 1, eksperimen ini bertujuan untuk mengetahui pengaruh parameter pada tiap algoritma.

i. Hill-climbing with Sideways Move



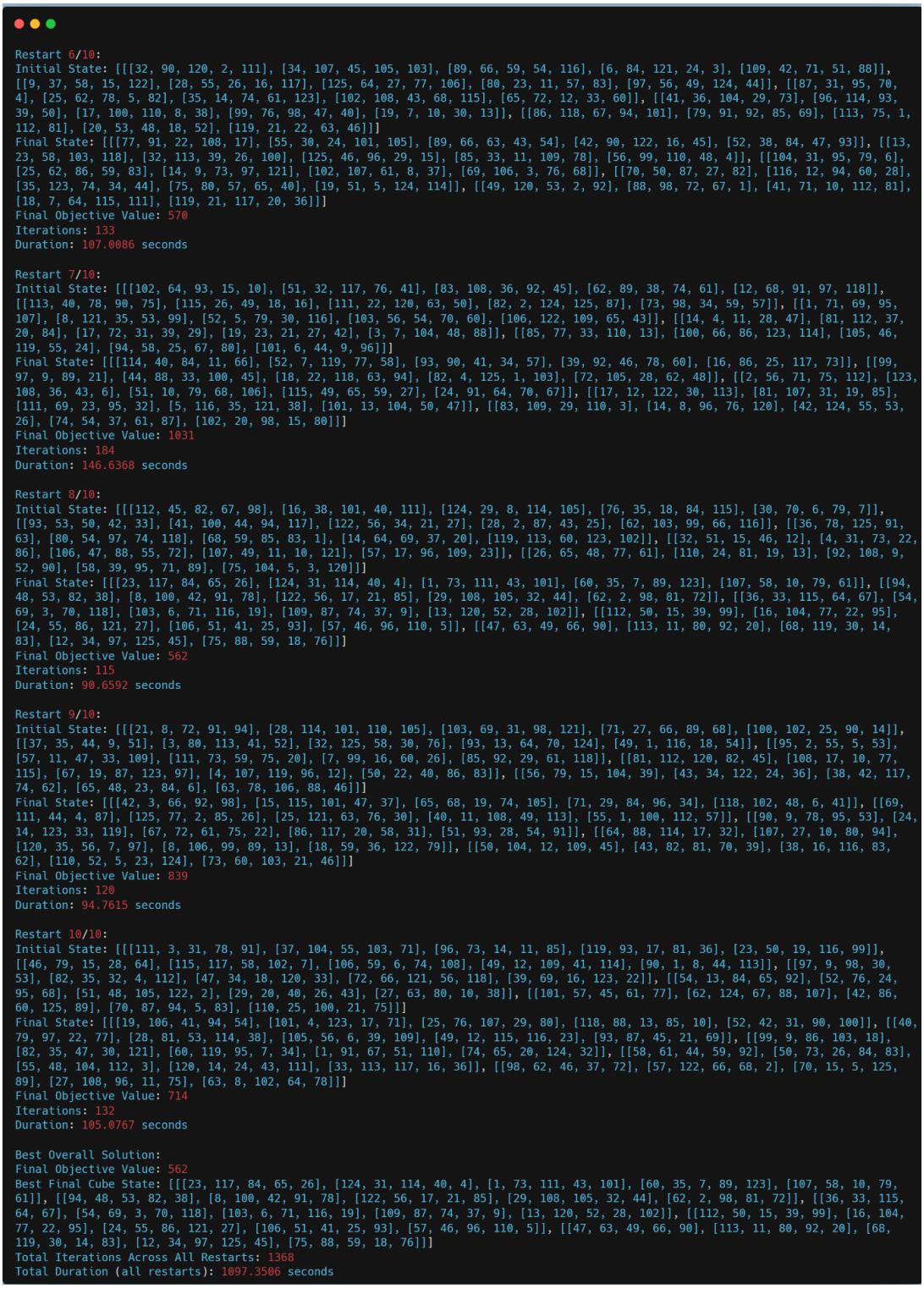
Gambar 5.1. Hasil dari algoritma Hill-climbing with Sideways Move



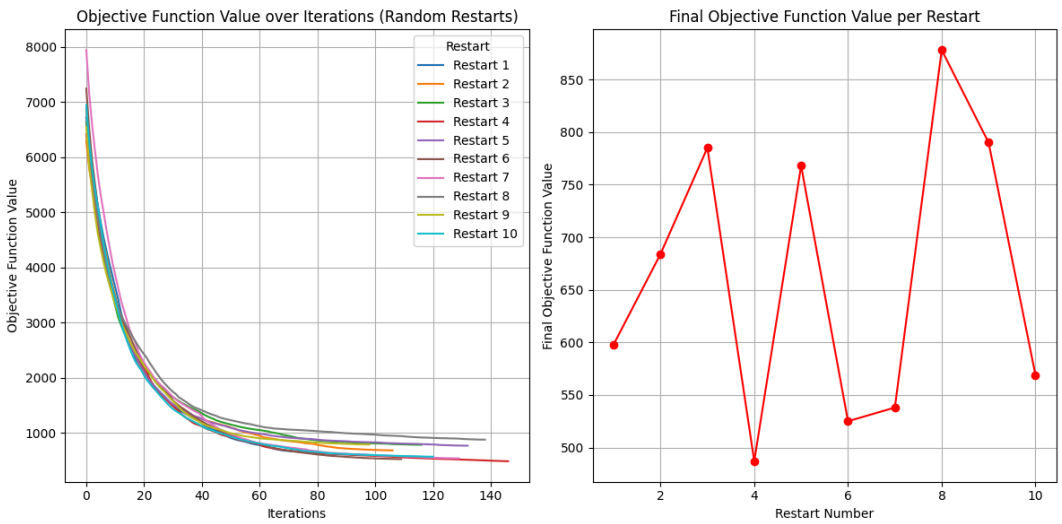
Gambar 5.2. Plot dari algoritma Hill-climbing with Sideways Move

ii. Random Restart Hill-climbing

```
Experiment Report:

Restart 1/10:
Initial State: [[[125, 26, 58, 116, 80], [6, 40, 92, 35, 93], [99, 61, 32, 72, 57], [51, 91, 66, 24, 115], [70, 86, 43, 103, 81]], [[60,
68, 120, 90, 41], [76, 10, 98, 104, 50], [123, 37, 45, 55, 42], [36, 46, 89, 16, 84], [49, 109, 94, 95, 48]], [[88, 23, 8, 12, 5], [62,
74, 7, 108, 34], [79, 83, 107, 100, 73], [77, 113, 38, 54, 97], [3, 63, 31, 14, 118]], [[122, 4, 117, 64, 27], [11, 33, 2, 59, 18],
[119, 65, 15, 106, 1], [44, 22, 121, 20, 75], [101, 17, 82, 13, 110]], [[29, 87, 28, 78, 71], [114, 56, 69, 105, 124], [85, 19, 25, 30,
47], [102, 112, 52, 111, 9], [67, 96, 39, 53, 21]]]
Final State: [[[119, 26, 83, 30, 57], [12, 54, 121, 35, 93], [5, 61, 65, 112, 72], [113, 89, 24, 39, 51], [71, 86, 22, 99, 37]], [[59,
42, 125, 21, 68], [45, 97, 9, 95, 69], [123, 34, 66, 56, 36], [40, 38, 90, 49, 98], [48, 104, 25, 94, 44]], [[88, 31, 77, 115, 4], [81,
75, 19, 108, 32], [67, 73, 63, 27, 85], [62, 122, 46, 7, 78], [17, 14, 110, 58, 117]], [[13, 120, 2, 64, 116], [91, 33, 114, 60, 18],
[107, 118, 16, 74, 1], [3, 23, 100, 109, 80], [101, 15, 82, 11, 106]], [[41, 92, 28, 84, 70], [87, 55, 50, 20, 103], [6, 29, 105, 47,
124], [102, 43, 52, 111, 8], [79, 96, 76, 53, 10]]]
Final Objective Value: 598
Iterations: 100
Duration: 81.1643 seconds

Restart 2/10:
Initial State: [[[91, 30, 60, 110, 59], [70, 43, 79, 29, 105], [40, 104, 12, 53, 61], [94, 6, 35, 38, 39], [18, 109, 124, 57, 111]],
[[72, 76, 82, 47, 92], [120, 15, 100, 122, 102], [21, 34, 63, 42, 37], [114, 52, 25, 101, 16], [123, 88, 9, 1, 90]], [[49, 86, 58, 75,
84], [119, 62, 87, 117, 20], [116, 14, 108, 44, 74], [67, 121, 28, 3, 33], [41, 95, 32, 99, 11]], [[54, 78, 13, 80, 93], [8, 24, 19,
113, 77], [36, 65, 45, 118, 46], [83, 89, 71, 50, 5], [68, 85, 22, 4, 115]], [[2, 66, 112, 81, 103], [10, 97, 55, 51, 7], [98, 27, 64,
31, 17], [106, 26, 48, 23, 125], [73, 107, 96, 56, 69]]]
Final State: [[[93, 20, 23, 123, 56], [49, 43, 81, 52, 90], [40, 119, 45, 53, 57], [94, 111, 59, 35, 16], [39, 22, 107, 51, 97]], [[72,
30, 117, 50, 44], [5, 67, 100, 33, 108], [28, 76, 63, 27, 121], [95, 54, 26, 109, 31], [115, 88, 9, 96, 6]], [[92, 86, 58, 1, 83], [71,
38, 87, 106, 15], [112, 48, 64, 14, 74], [18, 36, 21, 110, 125], [24, 102, 82, 99, 11]], [[61, 105, 13, 78, 77], [120, 41, 19, 66, 68],
[37, 69, 42, 118, 46], [32, 89, 124, 47, 8], [65, 12, 113, 7, 116]], [[2, 75, 104, 79, 55], [70, 122, 29, 60, 34], [98, 3, 101, 103,
17], [80, 25, 84, 10, 114], [73, 91, 4, 62, 85]]]
Final Objective Value: 684
Iterations: 106
Duration: 86.2241 seconds

Restart 3/10:
Initial State: [[[84, 35, 25, 89, 48], [32, 38, 98, 99, 124], [1, 109, 83, 121, 37], [60, 117, 49, 94, 73], [104, 88, 122, 18, 115]],
[[3, 15, 21, 62, 97], [28, 46, 107, 101, 34], [53, 75, 90, 39, 40], [65, 27, 2, 51, 63], [33, 79, 87, 110, 30]], [[82, 55, 6, 54, 36],
[23, 5, 112, 114, 91], [24, 71, 103, 74, 31], [77, 86, 68, 12, 16], [81, 58, 7, 59, 106]], [[120, 22, 4, 119, 20], [93, 78, 80, 61, 11],
[123, 116, 9, 108, 100], [64, 125, 85, 50, 8], [105, 52, 29, 67, 66]], [[113, 95, 14, 17, 26], [19, 96, 69, 70, 41], [44, 72, 10, 111,
118], [47, 13, 57, 42, 102], [76, 92, 56, 45, 43]]]
Final State: [[[47, 42, 48, 92, 77], [96, 87, 32, 6, 97], [8, 88, 74, 109, 36], [60, 56, 43, 94, 62], [104, 38, 118, 14, 41]], [[13,
108, 21, 49, 124], [101, 46, 100, 28, 37], [119, 65, 110, 11, 10], [67, 23, 2, 116, 114], [18, 73, 90, 107, 30]], [[112, 55, 63, 52,
34], [81, 1, 75, 69, 89], [24, 76, 86, 79, 50], [93, 125, 59, 12, 22], [7, 58, 27, 103, 120]], [[26, 15, 115, 106, 53], [4, 83, 70, 78,
82], [117, 85, 9, 5, 99], [64, 71, 91, 72, 16], [105, 61, 29, 54, 66]], [[113, 95, 68, 17, 25], [33, 98, 39, 123, 19], [45, 3, 35, 111,
121], [31, 40, 122, 20, 102], [80, 84, 51, 44, 57]]]
Final Objective Value: 785
Iterations: 116
Duration: 92.7706 seconds

Restart 4/10:
Initial State: [[[36, 48, 16, 82, 17], [54, 11, 95, 74, 26], [51, 60, 69, 40, 57], [18, 52, 78, 66, 19], [119, 81, 33, 34, 65]], [[73,
45, 37, 99, 67], [86, 109, 35, 56, 21], [77, 24, 75, 38, 39], [70, 6, 46, 22, 92], [31, 44, 64, 83, 111]], [[118, 62, 85, 23, 110],
[123, 108, 41, 27, 117], [90, 80, 113, 58, 102], [47, 91, 43, 96, 42], [9, 71, 107, 89, 103]], [[61, 68, 25, 55, 125], [79, 49, 7, 72,
3], [115, 8, 124, 98, 76], [13, 101, 2, 114, 94], [63, 10, 120, 32, 50]], [[53, 28, 93, 100, 4], [14, 12, 59, 20, 87], [29, 105, 15, 5,
84], [97, 88, 122, 30, 121], [104, 106, 112, 116, 1]]]
Final State: [[[67, 48, 91, 82, 28], [21, 15, 99, 60, 119], [51, 117, 49, 32, 69], [59, 53, 75, 106, 22], [118, 84, 1, 35, 76]], [[8,
109, 29, 54, 115], [123, 95, 50, 46, 2], [80, 24, 81, 120, 10], [73, 42, 78, 30, 92], [31, 44, 77, 62, 101]], [[110, 52, 86, 11, 55],
[122, 14, 40, 37, 102], [20, 64, 57, 58, 116], [56, 114, 45, 96, 4], [7, 71, 87, 113, 38]], [[61, 79, 17, 47, 111], [36, 98, 108, 70,
3], [125, 25, 23, 100, 41], [16, 97, 43, 65, 94], [72, 13, 124, 33, 66]], [[68, 27, 93, 121, 6], [12, 90, 18, 107, 89], [39, 85, 105, 5,
83], [112, 9, 74, 19, 103], [88, 104, 26, 63, 34]]]
Final Objective Value: 487
Iterations: 146
Duration: 115.7028 seconds

Restart 5/10:
Initial State: [[[86, 79, 57, 11, 70], [31, 36, 22, 30, 107], [116, 49, 25, 113, 89], [101, 13, 115, 45, 106], [88, 7, 84, 43, 47]],
[[58, 105, 96, 97, 109], [27, 92, 69, 40, 119], [117, 29, 82, 85, 20], [125, 102, 5, 48, 18], [103, 1, 81, 94, 67]], [[74, 50, 42, 118,
8], [6, 35, 90, 99, 76], [120, 122, 24, 33, 3], [93, 121, 98, 72, 91], [39, 73, 14, 26, 56]], [[53, 59, 83, 19, 114], [54, 28, 80, 37,
52], [111, 12, 124, 110, 38], [17, 95, 63, 87, 61], [16, 32, 15, 112, 100]], [[60, 65, 21, 4, 123], [44, 34, 2, 23, 51], [78, 71, 104,
64, 10], [108, 62, 41, 77, 66], [55, 68, 46, 75, 9]]]
Final State: [[[81, 74, 73, 20, 67], [41, 125, 15, 122, 12], [116, 56, 26, 27, 90], [14, 43, 119, 36, 99], [59, 17, 83, 110, 47]], [[57,
63, 95, 97, 3], [35, 88, 28, 40, 124], [5, 29, 71, 107, 105], [120, 102, 30, 49, 13], [98, 33, 91, 23, 70]], [[84, 53, 46, 121, 9], [2,
32, 96, 106, 77], [58, 100, 62, 16, 79], [93, 54, 10, 72, 86], [85, 76, 89, 1, 64]], [[50, 60, 80, 8, 118], [114, 31, 82, 38, 51], [111,
19, 44, 104, 37], [22, 92, 103, 87, 11], [18, 113, 6, 78, 101]], [[45, 65, 21, 68, 117], [123, 39, 94, 7, 52], [25, 115, 112, 61, 4],
[66, 24, 42, 75, 108], [55, 69, 48, 109, 34]]]
Final Objective Value: 768
Iterations: 132
Duration: 104.8382 seconds
```



Gambar 5.3. Hasil dari algoritma Random Restart Hill-climbing

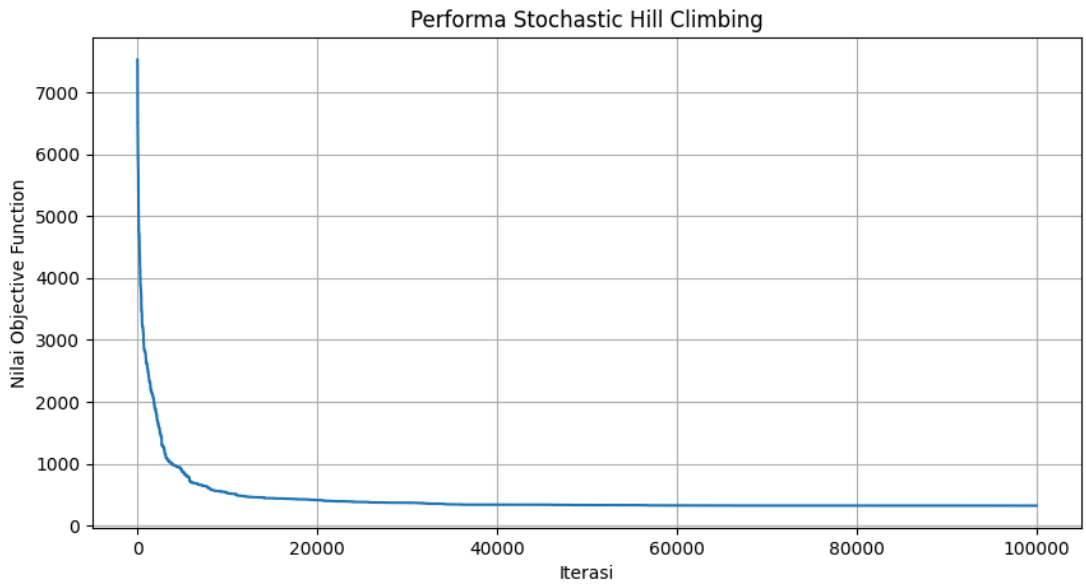


Gambar 5.4. Plot dari algoritma Random Restart Hill-climbing

iii. Stochastic Hill-climbing

```
===== Laporan Hasil Stochastic Hill Climbing =====
Durasi Pencarian       : 13.2489 detik
Total Iterasi          : 100000
Nilai Objective Awal   : 7525
Initial State:
[[[14, 73, 74, 24, 106], [63, 86, 117, 113, 96], [47, 109, 15, 1, 61], [50, 93, 19, 65, 84], [3, 111, 67, 85, 92]], [[72, 57, 10, 42,
102], [99, 45, 22, 77, 68], [121, 56, 62, 125, 58], [90, 118, 44, 97, 2], [59, 124, 110, 108, 100]], [[112, 17, 103, 79, 66], [39, 34,
87, 36, 6], [28, 48, 54, 81, 80], [35, 83, 114, 107, 98], [94, 7, 33, 69, 52]], [[71, 70, 82, 55, 89], [43, 123, 91, 122, 20], [101,
5, 12, 95, 51], [4, 64, 8, 105, 49], [11, 16, 76, 41, 37]], [[88, 29, 32, 116, 9], [26, 120, 78, 104, 27], [60, 38, 31, 46, 53], [21,
115, 13, 40, 30], [18, 25, 119, 75, 23]]]
Nilai Objective Akhir  : 324
Final State:
[[[43, 119, 103, 16, 29], [68, 5, 108, 33, 99], [3, 49, 88, 113, 61], [92, 53, 10, 112, 48], [114, 89, 6, 20, 81]], [[56, 122, 4, 51,
85], [121, 38, 17, 87, 52], [60, 75, 105, 47, 28], [63, 24, 120, 32, 76], [15, 54, 69, 104, 73]], [[50, 7, 96, 93, 77], [115, 123, 19,
45, 13], [25, 64, 74, 57, 95], [14, 21, 101, 55, 118], [106, 97, 22, 78, 12]], [[66, 36, 83, 30, 100], [9, 107, 80, 79, 40], [117, 18,
11, 98, 70], [84, 86, 27, 72, 46], [39, 67, 116, 34, 59]], [[102, 31, 35, 124, 23], [2, 42, 91, 71, 111], [110, 109, 37, 1, 65], [62,
125, 58, 44, 26], [41, 8, 94, 82, 90]]]
```

Gambar 5.5. Hasil dari algoritma Stochastic Hill-climbing

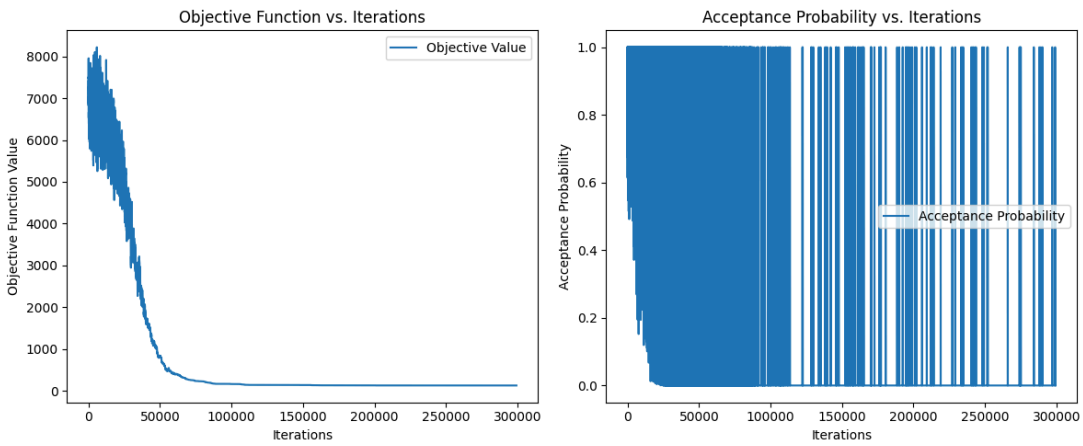


Gambar 5.6. Plot dari algoritma Stochastic Hill-climbing

iv. Simulated Annealing

```
=== Simulated Annealing Report ===
Initial Objective Value: 7407
Final Objective Value: 127
Total Iterations: 299322
Execution Time: 108.7811 seconds
Frequency of getting stuck in local optima: 271641

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

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

Gambar 5.7. Hasil dari algoritma Simulated Annealing

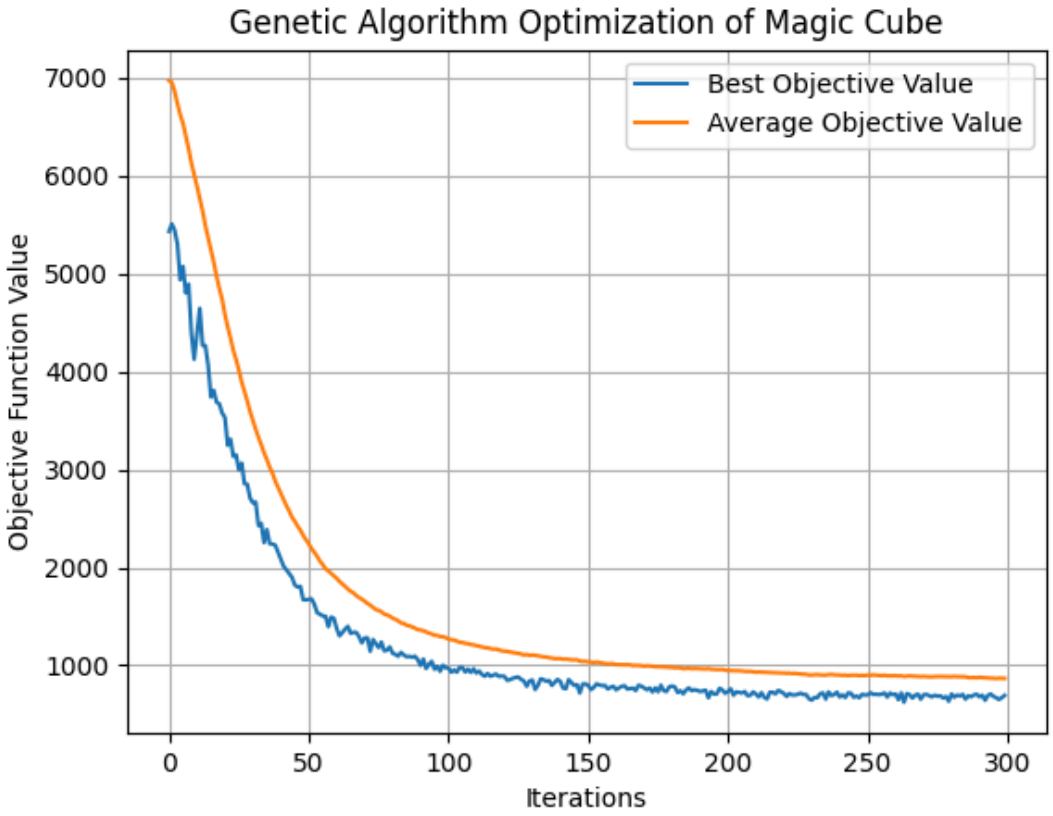


Gambar 5.8. Hasil dari algoritma Simulated Annealing

v. Genetic Algorithm

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

Final State:
[[[67, 56, 63, 62, 64], [62, 61, 56, 67, 69], [62, 70, 56, 71, 64], [68, 56, 67, 60, 62], [67, 63, 68, 66, 57]], [[65, 66, 68, 62,
64], [56, 64, 69, 57, 67], [64, 61, 57, 56, 63], [65, 63, 64, 68, 61], [69, 70, 54, 67, 66]], [[64, 59, 68, 60, 62], [63, 67, 64, 62,
61], [64, 63, 61, 62, 65], [57, 64, 66, 59, 67], [66, 56, 59, 68, 62]], [[58, 66, 62, 57, 70], [70, 65, 64, 61, 59], [62, 60, 57, 68,
61], [59, 65, 54, 69, 63], [61, 65, 60, 63, 66]], [[63, 70, 64, 59, 57], [60, 61, 62, 66, 63], [71, 63, 55, 57, 66], [66, 54, 59, 68,
62], [57, 66, 63, 69, 56]]]

Final Objective Value: 624
Population Size: 1000
Iterations: 300
Duration: 194.70 seconds
```

Gambar 5.9. Hasil dari algoritma Genetic Algorithm population size 1000 dan iterations 300



Gambar 5.10. Plot dari algoritma Genetic Algorithm population size 1000 dan iterations 300

5. Analisis dan Pembahasan

Dari hasil eksperimen diperoleh rangkuman hasil sebagai berikut.

Tabel 1 Analisis dan Pembahasan

No Eksperimen	Algoritma	Parameter	Objective Value	Duration(s)
1	Steepest HC	-	611	120
	Sideways HC	Sideways = 10	606	146
	Restart HC	Restart = 5	480	516
	Stochastic HC	Iter = 10000	749	1
	Simulated Annealing	Temp0 = 1000 Cooling rate = 0.95	3033	0.2
	Genetic Algorithm	Population size 100, Iterations 100 dan Mutation rate 0.01	2448	7
		Population size 100, Iterations 200 dan Mutation rate 0.01	2397	14
		Population size 100, Iterations 300 dan Mutation rate 0.01	2382	25
		Population size 200, Iterations 100 dan Mutation rate 0.01	1911	15
		Population size 300, Iterations 100 dan Mutation rate 0.01	1390	21

2	Steepest HC	-	849	354
	Sideways HC	Sideways = 10	849	200
	Restart HC	Restart = 5	571	742
	Stochastic HC	Iter = 10000	978	3
	Simulated Annealing	Temp0 = 1000 Cooling rate = 0.95	2775	0.6
	Genetic Algorithm	Population size 100, Iterations 100 dan Mutation rate 0.01	2378	12
		Population size 100, Iterations 200 dan Mutation rate 0.01	2221	20
		Population size 100, Iterations 300 dan Mutation rate 0.01	1914	49
		Population size 200, Iterations 100 dan Mutation rate 0.01	1886	23
		Population size 300, Iterations 100 dan Mutation rate 0.01	1612	33
	Steepest HC	-	347	126
	Sideways HC	Sideways = 10	347	137
	Restart HC	Restart = 5	484	648

3	Stochastic HC	Iter = 10000	1024	1
	Simulated Annealing	Temp0 = 1000 Cooling rate = 0.95	2855	0.1
	Genetic Algorithm	Population size 100, Iterations 100 dan Mutation rate 0.01	2339	7
		Population size 100, Iterations 200 dan Mutation rate 0.01	2013	14
		Population size 100, Iterations 300 dan Mutation rate 0.01	1942	22
		Population size 200, Iterations 100 dan Mutation rate 0.01	1745	15
		Population size 300, Iterations 100 dan Mutation rate 0.01	1525	22
4	Sideways HC	Sideways = 50	606	169
	Restart HC	Restart = 10	562	1097
	Stochastic HC	Iter = 100000	324	13
	Simulated Annealing	Temp0 = 1000 Cooling rate = 0.9999	127	108
	Genetic Algorithm	Population size 1000, Iterations 300 dan Mutation rate 0.01	624	194

Pada eksperimen 1 hingga 3, beberapa algoritma optimasi diuji untuk melihat performanya dalam menemukan nilai objektif terbaik pada kondisi awal yang sama. Hal ini dilakukan untuk membandingkan algoritma dalam hal efektivitas dan efisiensi dalam menangani masalah pencarian solusi.

Steepest Ascent Hill-Climbing (HC) menunjukkan hasil yang cepat namun seringkali terjebak dalam *local optima*. Pada eksperimen-eksperimen ini, algoritma ini menghasilkan nilai objektif yang cukup rendah, yaitu 611 pada Eksperimen 1, 849 pada Eksperimen 2, dan 347 pada Eksperimen 3. Meskipun demikian, hasilnya cenderung suboptimal karena Steepest HC tidak memiliki mekanisme untuk keluar dari local optima. Durasi eksekusinya relatif singkat, dengan waktu 120 detik pada Eksperimen 1 dan 126 detik pada Eksperimen 3. Ini menandakan bahwa meskipun Steepest HC efisien, algoritma ini kurang cocok untuk landscape yang memiliki banyak titik lokal.

Hill-Climbing dengan Sideways Move memperbaiki keterbatasan *Steepest HC* dalam mengatasi *local optima* dengan menambahkan mekanisme *sideways move* sebanyak 10 kali. Pada eksperimen-eksperimen ini, *Hill-Climbing* dengan *Sideways Move* menghasilkan nilai objektif yang sedikit lebih baik atau mirip dengan *Steepest HC*, yaitu 606 pada Eksperimen 1 dan tetap 849 pada Eksperimen 2. Waktu eksekusinya sedikit lebih lama, dengan durasi 146 detik pada Eksperimen 1, namun hasil yang dicapai tidak terlalu berbeda secara signifikan dari *Steepest HC*. Penambahan *sideways move* memberikan fleksibilitas tambahan, tetapi peningkatan hasilnya masih terbatas.

Random Restart Hill-Climbing menunjukkan peningkatan yang cukup signifikan dalam hal menghindari jebakan *local optima* dengan melakukan *restart* sebanyak 5 kali. Algoritma ini menghasilkan nilai objektif yang lebih rendah dibandingkan *Hill-Climbing* biasa dan varian dengan *sideways move,* yaitu 480 pada Eksperimen 1 dan 571 pada Eksperimen 2. Durasi eksekusinya meningkat cukup tinggi, menjadi 516 detik pada Eksperimen 1 dan 742 detik pada Eksperimen 2. Hal ini menunjukkan bahwa restart dapat membantu algoritma keluar dari titik lokal yang buruk, namun memerlukan waktu pemrosesan yang lebih lama.

Stochastic Hill-Climbing menonjol dalam kecepatan eksekusinya karena menggunakan pendekatan acak untuk memilih *neighbor*. Pada Eksperimen 1, algoritma ini menghasilkan nilai objektif 749 hanya dalam 1 detik. Namun, nilai objektif yang diperoleh umumnya lebih tinggi dibandingkan algoritma lain, seperti pada Eksperimen 2 yang menghasilkan nilai 978 dalam waktu 3 detik. Algoritma ini cepat, tetapi kurang efektif dalam menemukan solusi optimal pada *landscape* yang lebih kompleks karena hasilnya sangat bergantung pada keberuntungan dalam memilih solusi yang baik secara acak.

Simulated Annealing (SA) memanfaatkan penurunan temperatur untuk mengendalikan penerimaan solusi yang lebih buruk pada tahap awal, memungkinkan algoritma untuk mengeksplorasi lebih luas. Pada Eksperimen 1, *SA* menghasilkan nilai objektif 3033 dalam waktu yang sangat singkat, yaitu 0,2 detik. Hasil ini menunjukkan bahwa, meskipun cepat, parameter temperatur dan *cooling rate* yang digunakan tidak cukup optimal untuk *landscape* yang lebih sulit. Pada eksperimen ini, *SA* belum mampu menemukan nilai objektif serendah algoritma lainnya.

Genetic Algorithm (GA) memiliki pendekatan yang berbeda dengan menggunakan populasi solusi dan mekanisme *crossover* serta mutasi untuk mengeksplorasi ruang solusi. Pada Eksperimen 1 hingga 3, GA menunjukkan bahwa dengan meningkatkan jumlah populasi dan iterasi, hasilnya terus membaik meskipun membutuhkan durasi yang lebih lama. Sebagai contoh, pada Eksperimen 1

dengan populasi 300 dan iterasi 100, *GA* mencapai nilai objektif 1390 dalam 21 detik. Pada Eksperimen 2 dan 3, hasil terbaik *GA* masing-masing adalah 1612 dan 1525. *GA* mampu mengeksplorasi solusi yang lebih luas melalui variasi dalam populasi, namun durasi eksekusinya lebih panjang dibandingkan algoritma lain.

Pada Eksperimen 4, parameter algoritma diatur lebih spesifik untuk meningkatkan hasil dari eksperimen sebelumnya dan mengoptimalkan performa setiap algoritma. Eksperimen ini merupakan lanjutan dari Eksperimen 1, dengan tujuan mengeksplorasi apakah penyesuaian parameter dapat meningkatkan kemampuan algoritma dalam menghindari *local optima* atau mencapai solusi yang lebih baik.

Hill-Climbing dengan *Sideways Move* pada eksperimen ini memiliki jumlah *sideways* yang lebih tinggi, yaitu sebanyak 50 kali. Meskipun penambahan *sideways move* ini meningkatkan fleksibilitas algoritma untuk melanjutkan pencarian di area yang sama, hasilnya tidak menunjukkan peningkatan signifikan dibandingkan eksperimen sebelumnya. *Hill-Climbing* dengan 50 *sideways* menghasilkan nilai objektif 606 dengan durasi yang sedikit lebih panjang, yaitu 169 detik. Hal ini menunjukkan bahwa peningkatan *sideways* tidak selalu efektif untuk *landscape* yang lebih kompleks.

Random Restart Hill-Climbing pada Eksperimen 4 menggunakan jumlah *restart* yang lebih banyak, yaitu sebanyak 10 kali. Dengan penambahan *restart* ini, algoritma diharapkan dapat lebih baik dalam menghindari *local optima* yang buruk. Hasilnya, algoritma ini mencapai nilai objektif 562 dengan durasi eksekusi yang lebih panjang, yaitu 1097 detik. Meskipun memerlukan waktu lebih lama, *Random Restart HC* mampu menghasilkan solusi yang lebih baik dibandingkan pada Eksperimen 1-3. Ini menandakan bahwa restart tambahan efektif dalam mengatasi keterbatasan *landscape* yang kompleks.

Stochastic Hill-Climbing pada eksperimen ini menggunakan iterasi sebanyak 100.000 untuk memungkinkan pencarian yang lebih luas di *landscape*. Dengan peningkatan jumlah iterasi, algoritma ini mampu mencapai nilai objektif yang lebih rendah, yaitu 324, meskipun durasinya relatif lebih panjang, yaitu 13 detik. Peningkatan jumlah iterasi membantu *Stochastic HC* untuk keluar dari solusi yang tidak optimal, meskipun hasilnya tidak seefektif beberapa algoritma lainnya.

Simulated Annealing (*SA*) pada eksperimen ini menggunakan nilai temperatur awal sebesar 1000 dan *cooling rate* yang lebih lambat, yaitu 0,9999. Dengan parameter ini, SA dapat mengeksplorasi solusi lebih luas pada tahap awal dan mempertahankan eksplorasi tersebut lebih lama seiring penurunan suhu yang lambat. Hasilnya adalah nilai objektif 127 dengan durasi yang cukup panjang, yaitu 108 detik. Penurunan suhu yang lebih lambat memungkinkan algoritma ini menemukan solusi yang lebih optimal dibandingkan eksperimen sebelumnya, menjadikan *SA* salah satu algoritma dengan hasil terbaik pada Eksperimen 4.

Genetic Algorithm (*GA*) pada eksperimen ini memiliki parameter yang ditingkatkan secara signifikan, dengan ukuran populasi sebanyak 1000, iterasi 300, dan *mutation rate* sebesar 0,01. Penambahan populasi ini memungkinkan *GA* mengeksplorasi solusi yang lebih luas, sementara iterasi yang banyak memberi waktu bagi algoritma untuk mengkonsolidasi solusi yang lebih baik melalui seleksi alamiah. Dengan pengaturan ini, *GA* berhasil mencapai nilai objektif 624 dalam waktu eksekusi 194 detik. Hasil ini memperlihatkan bahwa *GA* efektif untuk *landscape* yang memerlukan eksplorasi luas, tetapi memerlukan durasi yang cukup panjang.

6. Kesimpulan

Kesimpulannya, dari eksperimen 1 hingga 3, algoritma dengan kemampuan eksplorasi lebih luas, seperti *Random Restart Hill-Climbing*, *Simulated Annealing*, dan *Genetic Algorithm*, cenderung memberikan solusi lebih baik pada *landscape* pencarian yang kompleks, meskipun waktu komputasi yang dibutuhkan lebih lama. Sebaliknya, algoritma *Steepest HC* dan *Stochastic HC* unggul dalam kecepatan durasi tetapi sering terjebak pada solusi suboptimal karena kurangnya mekanisme untuk keluar dari optima lokal.

Pada Eksperimen 4, penyesuaian parameter menunjukkan peran krusial dalam meningkatkan performa. *Simulated Annealing* dengan *cooling rate* yang lambat dan *Genetic Algorithm* dengan populasi besar berhasil mencapai hasil terbaik, memperlihatkan bahwa eksplorasi yang luas dan kemampuan algoritma untuk mempertahankan pencarian pada area yang kompleks memberikan keuntungan dalam menemukan solusi yang lebih optimal. Hal ini menegaskan bahwa, terutama untuk *landscape* pencarian yang rumit, konfigurasi parameter yang tepat dapat meningkatkan efisiensi algoritma dalam mencapai solusi yang diinginkan.

Daftar Pustaka

1. AIMACode. (n.d.). Simulated Annealing. Retrieved from https://github.com/aimacode/aima-pseudocode/blob/master/md/Simulated-Annealing.md
2. Artificial Intelligence Kosta. (n.d.). Local search algorithms – Magic Square problem. Retrieved from https://github.com/Artificial-Intelligence-kosta/Local-search-algorithms-Magic-Square-problem/blob/master/Algorithms.py
3. Baeldung. (n.d.). Simulated annealing in computer science. Retrieved from https://www.baeldung.com/cs/simulated-annealing
4. GeeksforGeeks. (n.d.). Genetic algorithms. Retrieved from https://www.geeksforgeeks.org/genetic-algorithms/
5. GeeksforGeeks. (n.d.). Introduction to hill climbing in artificial intelligence. Retrieved from https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/
6. GeeksforGeeks. (n.d.). Simulated annealing. Retrieved from https://www.geeksforgeeks.org/simulated-annealing/
7. Hacettepe University. (n.d.). Local search. Retrieved from https://web.cs.hacettepe.edu.tr/~ilyas/Courses/VBM688/lec05_localsearch.pdf
8. JavaTpoint. (n.d.). Hill climbing algorithm in AI. Retrieved from https://www.javatpoint.com/hill-climbing-algorithm-in-ai
9. Magisch Vierkant. (n.d.). Three-dimensional magic squares – 5x5x5. Retrieved from https://www.magischvierkant.com/three-dimensional-eng/5x5x5/
10. Russell, S., & Norvig, P. (n.d.). Artificial Intelligence: A Modern Approach. Retrieved from https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf
11. ScienceDirect. (n.d.). Local search algorithm. Retrieved from https://www.sciencedirect.com/topics/computer-science/local-search-algorithm
12. Trump, R. (n.d.). Magic cubes. Retrieved from https://www.trump.de/magic-squares/magic-cubes/cubes-1.html
13. Wikipedia. (n.d.). Magic cube. Retrieved from https://en.wikipedia.org/wiki/Magic_cube
14. Wolfram Research. (n.d.). Magic cube (m021). Retrieved from https://archive.lib.msu.edu/crcmath/math/math/m/m021.htm
15. Wolfram Research. (n.d.). Magic cube (m022). Retrieved from https://archive.lib.msu.edu/crcmath/math/math/m/m022.htm