

**Tugas Besar 1 IF3170 Intelelegensi Artifisial**

**“Pencarian Solusi Diagonal Magic Cube dengan Local Search”**



Disusun oleh:

Kayla Namira Mariadi	K01	13522050
Andhita Naura Hariyanto	K01	13522060
Salsabiila	K02	13522062
Shulha	K02	13522087

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**  
**INSTITUT TEKNOLOGI BANDUNG**

**2024**

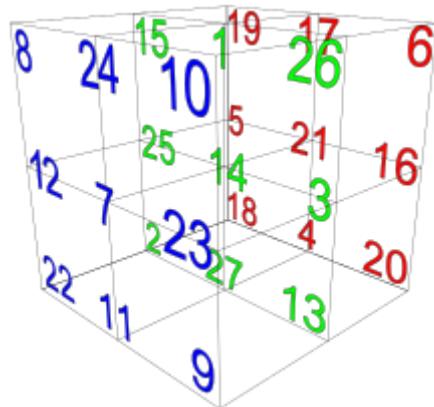
## DAFTAR ISI

<b>DAFTAR ISI</b>	<b>1</b>
<b>DESKRIPSI PERSOALAN</b>	<b>3</b>
<b>PEMBAHASAN</b>	<b>4</b>
1. Pemilihan dan Penjelasan Objective Function	4
2. Penjelasan Algoritma Local Search	9
A. Kelas Cube dan Fungsi-Fungsi Utilitas	9
B. Steepest Ascent Hill-Climbing	11
C. Hill-climbing with Sideways Move	14
D. Random Restart Hill-climbing	16
E. Stochastic Hill-climbing	18
F. Simulated Annealing	20
G. Genetic Algorithm	27
3. Hasil Eksperimen	34
A. Steepest Ascent Hill-Climbing	34
1. Eksperimen I	35
2. Eksperimen II	38
3. Eksperimen III	42
B. Hill-climbing with Sideways Move	44
1. Eksperimen I	45
2. Eksperimen II	48
3. Eksperimen III	52
C. Random Restart Hill-climbing	54
1. Eksperimen I	55
2. Eksperimen II	59
3. Eksperimen III	63
D. Stochastic Hill-climbing	65
1. Eksperimen I	66
2. Eksperimen II	69
3. Eksperimen III	72
E. Simulated Annealing	74
1. Eksperimen I	75
2. Eksperimen II	79
3. Eksperimen III	83
F. Genetic Algorithm	85
1. Eksperimen I	86
2. Eksperimen II	90
3. Eksperimen III	94
4. Eksperimen IV	98
5. Eksperimen V	101
6. Eksperimen VI	105
4. Analisis	108

A. Seberapa dekat tiap-tiap algoritma bisa mendekati global optima? Mengapa hasilnya demikian?	108
B. Bagaimana perbandingan hasil pencarian tiap-tiap algoritma dengan algoritma local search yang lain?	110
C. Bagaimana perbandingan durasi proses pencarian tiap algoritma relatif terhadap algoritma lainnya?	111
D. Seberapa konsisten hasil akhir yang didapatkan dari tiap-tiap eksperimen yang dilakukan	112
E. Bagaimana pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada Genetic Algorithm?	112
<b>PENUTUP</b>	<b>115</b>
1. Kesimpulan	115
2. Saran	115
3. Pembagian Tugas	115
<b>REFERENSI</b>	<b>116</b>

## DESKRIPSI PERSOALAN

Diagonal magic cube adalah salah satu dari enam kelas magic cube yang merupakan sebuah kubus yang terdiri dari angka-angka mulai dari 1 hingga  $n^3$  tanpa pengulangan, di mana  $n$  merupakan panjang sisi kubus tersebut. Dalam susunan ini, angka-angka ditempatkan sedemikian rupa sehingga beberapa properti tertentu harus dipenuhi. Salah satu properti utama diagonal magic cube adalah adanya sebuah angka yang disebut magic number. Magic number ini tidak harus termasuk dalam rentang angka 1 hingga  $n^3$  dan juga tidak harus menjadi bagian dari angka yang diisi ke dalam kubus. Setiap baris, kolom, dan tiang dalam kubus harus memiliki jumlah angka yang sama dengan magic number. Selain itu, jumlah angka pada semua diagonal ruang dalam kubus juga harus sama dengan magic number. Hal yang sama berlaku untuk setiap diagonal pada potongan bidang dari kubus tersebut, yang juga harus memiliki jumlah yang sama dengan magic number.



Tugas ini melibatkan sebuah diagonal magic cube berukuran  $5 \times 5 \times 5$  dengan kondisi awal kubus diisi dengan angka 1 hingga  $5^3$  yang disusun secara acak. Pada setiap iterasi dalam algoritma pencarian lokal, langkah yang diizinkan adalah menukar posisi dua angka dalam kubus tersebut, dan angka yang ditukar tidak harus bersebelahan.

## PEMBAHASAN

### 1. Pemilihan dan Penjelasan *Objective Function*

Fungsi objektif atau *objective function* secara definisi adalah fungsi matematika yang digunakan dalam masalah optimasi untuk mengevaluasi solusi dan menentukan seberapa "baik" solusi tersebut dalam mencapai tujuan yang diinginkan. Dalam permasalahan *diagonal magic cube* ini, fungsi objektif adalah fungsi yang mengevaluasi seberapa 'dekat' *state* kubus saat ini dengan *diagonal magic cube*.

Fungsi objektif mengukur nilai atau kualitas dari suatu solusi dengan menghitung hasil dari variabel-variabel yang terlibat. Pada permasalahan *diagonal magic cube* ini, dapat dilakukan berbagai cara untuk merumuskan fungsi objektif. Pada masalah ini, penulis akan memaparkan fungsi objektif yang akan digunakan untuk mencari solusi *diagonal magic cube*. Fungsi objektif untuk permasalahan ini dapat berupa nilai jumlah dari perbedaan mutlak antara nilai *magic number* dengan nilai jumlah baris/kolom/diagonal pada kubus. Fungsi objektif tersebut dapat diilustrasikan sebagai berikut:

1. Pada suatu kubus, dihitung jumlah dari masing-masing baris, kolom, tiang, serta diagonal sisi, ruang, dan bidang. Untuk kemudahan penjelasan pada dokumen ini, jumlah dari suatu dan masing-masing baris/kolom/diagonal tersebut disebut sebagai 'sum line'.
2. Pada setiap jumlah tersebut, dihitung perbedaan absolutnya terhadap nilai *magic number*
3. Nilai fungsi objektif adalah jumlah seluruh perbedaan absolut terhadap nilai *magic number*

Untuk itu, fungsi objektif ini dapat dinyatakan sebagai

$$f(C) = \Sigma |sum\ line - magic\ number|$$

Suatu *state* kubus akan semakin baik saat nilai fungsi objektif mendekati nol dan meraih nol. Untuk itu, program untuk mencari solusi *diagonal magic cube* berharap meminimumkan nilai fungsi objektif pada setiap iterasi pencarinya.

Namun, penulis berupaya untuk mengeskalasi ide fungsi objektif ini dengan metode fungsi objektif lain yang akan disebut 'Balanced Constraint Objective Function'. Selain hanya menambahkan semua perbedaan absolut antara 'sum line' dan *magic number*,

fungsi objektif ini juga berusaha untuk menyeimbangkan/*balancing constraints* (perbedaan absolut antara ‘sum line’ dan *magic number*) dengan mempertimbangkan variansi dan *mean difference* dari seluruh ‘sum line’.

‘Balanced Constraint Objective Function’ ini bertujuan untuk meminimalkan bukan hanya deviasi keseluruhan dari *magic number*, namun juga variansi dari seluruh ‘sum line’. Fungsi objektif ini meminimalkan deviasi rata-rata maupun varians di seluruh ‘sum line’ sehingga mendorong nilai objektif yang lebih merata, menghindari kasus di mana beberapa ‘sum line’ sangat dekat dengan *magic number* sementara bagian kubus lainnya jauh dari jumlah tersebut. Fungsi objektif dengan *balanced constraint* ini diharapkan dapat merepresentasikan kondisi aktual kedekatan antara *state* kubus saat ini dengan *diagonal magic cube*. ‘Balanced Constraint Objective Function’ dijelaskan sebagai berikut:

$$f(C) = \alpha \times \text{Mean Difference} + \beta \times \text{Variance}$$

Fungsi tersebut menunjukkan kombinasi pembobotan antara perbedaan rata-rata ‘mean difference’ dan variansi. ‘Mean difference’ pada fungsi objektif ini mengukur seberapa jauh secara rata-rata seluruh ‘sum line’ dari keadaan kubus pada saat ini. ‘Mean difference’ ini dapat dihitung dengan

$$\text{mean difference} = \frac{1}{N} \sum |\text{sum line} - \text{magic number}|$$

Sementara itu, varians menunjukkan seberapa besar ‘sum line’ bervariasi dari rata-rata seluruh ‘sum line’. Jadi, misal nilai ‘sum line’ tersebar dengan rentang yang cukup jauh, varians akan menyoroti betapa tidak meratanya jumlah-jumlah ini didistribusikan di sekitar jumlah rata-rata. Varians dapat dihitung dengan

$$\text{variance} = \frac{1}{N} \sum (\text{sum line} - \text{avg. sum line})^2$$

Variabel  $\alpha$  dan  $\beta$  pada fungsi objektif merupakan faktor pembobot pentingnya dan prioritas ‘main difference’ dan varians. Memberi nilai  $\alpha$  yang lebih besar berarti memprioritaskan mengurangi nilai ‘main difference’. Sementara itu, memberi nilai  $\beta$  lebih besar memfokuskan pada mengurangi variansi nilai ‘sum line’.

Alasan kuat penulis yang sangat mempertimbangkan fungsi objektif ini dari sekadar menghitung ‘sum line’ adalah bahwa dengan meminimalkan perbedaan rata-rata dan varians, solusi tidak hanya mendekati *magic number* tetapi juga konsisten di seluruh

kubus. Dalam hal ini, varians atau ukuran variasi dari ‘sum line’ menjadi diperlukan. Jika hanya meminimalkan ‘main difference’, solusi mungkin berkumpul pada keadaan di mana beberapa baris/kolom sangat dekat dengan *magic number* sementara yang lain jauh.

Sebenarnya, ada salah satu alternatif fungsi objektif yang sempat penulis pertimbangkan yaitu dengan membandingkan posisi setiap elemen dalam kubus kita saat ini dengan posisi yang bersesuaian pada *magic cube* asli, yang sudah berhasil diselesaikan oleh Daniel Trump dan Christian Boyer.

Fungsi objektif ini akan memiliki keunggulan dalam hal efisiensi komputasi. Hanya diperlukan 125 kali perbandingan, atau dengan kata lain, kompleksitas algoritma menjadi  $O(1)$  karena kita hanya membandingkan setiap elemen satu per satu dengan referensi kubus yang sudah sempurna. Dari segi performa, ini tentu akan sangat menguntungkan.

Namun, setelah mempertimbangkan lebih lanjut, penulis merasa bahwa pendekatan ini tidaklah sesuai dengan tujuan umum dari masalah yang ingin kita selesaikan. Salah satu alasannya adalah bahwa dengan hanya berpatokan pada satu kubus sempurna sebagai referensi, kita secara tidak langsung mengabaikan kemungkinan solusi lain yang juga sama-sama valid. Sebagai contoh, Daniel Trump dan Christian Boyer telah menemukan tiga kubus ordo-5 yang sempurna, dan memilih hanya salah satu dari mereka sebagai acuan rasanya kurang tepat. Kita akan seolah-olah menutup mata terhadap solusi-solusi lain yang sama-sama optimal.

Jadi, meskipun fungsi objektif berbasis perbandingan dengan kubus sempurna memberikan keuntungan dalam hal efisiensi, ia gagal menangkap esensi permasalahan secara utuh dan membuka peluang untuk menemukan solusi yang bervariasi dan lebih kaya.

Dengan memilih **Balanced Constraint Objective Function**, penulis berusaha mencari solusi yang lebih fleksibel. Pendekatan ini memungkinkan kita untuk mengeksplorasi berbagai kemungkinan solusi tanpa harus membatasi diri hanya pada satu solusi spesifik yang sudah diketahui. Selain itu, fungsi objektif ini juga lebih mencerminkan kompleksitas dari masalah diagonal *magic cube*, yang melibatkan berbagai constraint yang harus dipenuhi, bukan sekadar kesesuaian elemen dengan satu kubus sempurna.

Pada fungsi Balanced Constraint Objective Function, elemen pada setiap baris, kolom, dan diagonal (disebut line) masing-masing akan dihitung jumlah dan besar perbedaan

jumlahnya dengan magic number. Pada persoalan diagonal magic cube 5x5x5, magic number didapatkan dengan memasukkan nilai n = 5 menggunakan rumus berikut.

$$M_3(n) = \frac{n(n^3 + 1)}{2} = 315$$

Setelah mendapat jumlah selisih pada seluruh line dan membaginya dengan banyak line, maka akan didapatkan rata-rata perbedaan atau mean difference dari state kubus tersebut. Setelah itu, dilakukan iterasi sebanyak jumlah line untuk mendapatkan variansi kubus. Nilai dari mean difference dan variansi masing-masing akan dikalikan dengan konstanta alpha dan beta.

Berikut adalah kode algoritma Balanced Constraint Objective Function yang digunakan

Fungsi menggunakan alpha = beta dan magic number acuan yaitu 315 serta menyimpan line_sums seluruh line. Best line dan worst line disimpan untuk kebutuhan genetic algorithm.	<pre>import numpy as np  def objective_function(cube):     """Calculate the objective function based on mean difference and variance of line sums."""     alpha=0.5     beta=0.5     magic_number=315     line_sums = []     best_lines = []     worst_lines = []</pre>
Bagian ini akan memperbarui nilai best lines dan worst lines dari kubus, yakni line yang memiliki deviasi terkecil dan terbesar dari magic number. Best dan worst lines adalah array yang berisi 3 elemen indeks dari line-line terbaik dan terburuk tersebut.	<pre># Helper to update best and worst lines def update_best_lines(deviation, indices):     if len(best_lines) &lt; 3:         best_lines.append((deviation, indices))         best_lines.sort() # Keep best_lines sorted by deviation (smallest first)     else:         if deviation &lt; best_lines[-1][0]:             best_lines[-1] = (deviation, indices)             best_lines.sort() # Keep best_lines sorted by deviation  def update_worst_lines(deviation, indices):     if len(worst_lines) &lt; 3:         worst_lines.append((deviation, indices))         worst_lines.sort(reverse=True) # Keep worst_lines sorted by deviation (largest first)     else:         if deviation &gt; worst_lines[-1][0]:             worst_lines[-1] = (deviation, indices)             worst_lines.sort(reverse=True) # Keep worst_lines sorted by deviation</pre>
Fungsi menghitung line_sum dari masing-masing baris	<pre>for i in range(5):     for j in range(5):         row_sum = np.sum(cube[i, j, :])</pre>

(25), kolom (25), dan tiang (25) pada kubus.	<pre> line_sums.append(row_sum)  column_sum = np.sum(cube[i, :, j]) line_sums.append(column_sum)  pillar_sum = np.sum(cube[:, i, j]) line_sums.append(pillar_sum) </pre>
Fungsi menghitung line_sum dari masing-masing diagonal sisi pada kubus (30).	<pre> for i in range(5):     face_diag1_sum = np.sum([cube[i, j, j] for j in range(5)])     face_diag2_sum = np.sum([cube[i, j, 5 - 1 - j]                            for j in range(5)])     line_sums.extend([face_diag1_sum,                       face_diag2_sum])      face_diag3_sum = np.sum([cube[j, i, j] for j in range(5)])     face_diag4_sum = np.sum([cube[j, i, 5 - 1 - j]                            for j in range(5)])     line_sums.extend([face_diag3_sum,                       face_diag4_sum])      face_diag5_sum = np.sum([cube[j, j, i] for j in range(5)])     face_diag6_sum = np.sum([cube[j, 5 - 1 - j, i]                            for j in range(5)])     line_sums.extend([face_diag5_sum,                       face_diag6_sum]) </pre>
Fungsi menghitung line_sum dari masing-masing diagonal ruang pada kubus (4).	<pre> space_diag1_sum = np.sum([cube[i, i, i] for i in range(5)]) space_diag2_sum = np.sum([cube[i, i, 5 - 1 - i] for i in range(5)]) space_diag3_sum = np.sum([cube[i, 5 - 1 - i, i] for i in range(5)]) space_diag4_sum = np.sum([cube[i, 5 - 1 - i, 5 - 1 - i] for i in range(5)]) line_sums.extend([space_diag1_sum, space_diag2_sum,                   space_diag3_sum, space_diag4_sum]) </pre>
Fungsi rata rata deviasi dan variansi deviasi dan mengembalikan nilai fungsi objektif berupa jumlah keduanya secara terbobot.	<pre> mean_difference = np.mean([abs(line_sum - 315) for sum_line in line_sums]) avg_sum_line = np.mean(line_sums) variance = np.mean([(sum_line - avg_sum_line) ** 2                     for sum_line in line_sums])  objective_value = alpha * mean_difference + beta * variance return objective_value </pre>

## 2. Penjelasan Algoritma *Local Search*

### A. Kelas Cube dan Fungsi-Fungsi Utilitas

Kelas kubus yang diberi nama Cube memiliki atribut-atribut state, fitness-value, switched\_coordinate. Kelas ini juga memiliki metode dan fungsi-fungsi utilitas seperti calculate\_fitness, display, swap\_two\_elements, generate\_successors, hingga find\_random\_successor.

- a. Atribut state adalah konfigurasi kubus saat ini yang merupakan array 3 dimensi.
- b. Atribut fitness\_value merupakan nilai hasil fungsi objektif yang didapat dari penghitungan melalui fungsi objektif. Semakin mendekati nol suatu fitness\_value suatu kubus, maka konfigurasi kubus tersebut semakin mendekati konfigurasi diagonal magic cube yang sesuai.
- c. Atribut switched\_coordinate adalah array yang menyimpan data koordinat penukaran elemen pada iterasi sebelumnya. Atribut ini mencegah *endless loop* yang terus menukar pada koordinat tertentu pada pencarian dengan *hill climbing*.
- d. Atribut best\_lines dan worst\_lines akan menyimpan informasi indeks-indeks line yang memiliki deviasi terkecil dan terbesar dari magic number.
- e. Fungsi calculate\_fitness memanggil fungsi objektif untuk *assign* nilai fitness\_value dari cube.
- f. Fungsi display menampilkan state kubus di layar CLI dengan sleep selama 0.5 detik.

```
import numpy as np
import random
import time
from objective_function import objective_function

class Cube:
    def __init__(self, state=None):
        self.state = state if state is not None else
np.random.permutation(np.arange(1, 126)).reshape(5, 5, 5)
        self.switched_coordinate = None
        self.fitness_value, self.best_lines, self.worst_lines
= self.calculate_fitness()

    def calculate_fitness(self):
        fitness_value, best_lines, worst_lines =
objective_function_latest(self.state)
        return fitness_value, best_lines, worst_lines
```

```

def display(self):
    for i in range(5):
        time.sleep(0.5)
        print(f"Layer {i + 1}:\n{self.state[i]}\n")

    print()

```

- g. Fungsi swap\_two\_elements menukar dua elemen pada kubus dan menghitung kembali nilai fitness\_value dari cube.

```

def swap_two_elements(self, i, j, k, x, y, z):
    self.state[i, j, k], self.state[x, y, z] =
    self.state[x, y, z], self.state[i, j, k]
    self.fitness_value = self.calculate_fitness()

```

- h. Fungsi generate\_successors membuat semua suksesor dari kubus saat ini dalam array dan menghitung nilai fitness value dari seluruh kubus suksesor. Iterasi dilakukan dengan loop while pada seluruh indeks koordinat pada kubus dan increment satu persatu.

```

def generate_successors(self):
    successors = []
    i, j, k = 0, 0, 0
    x, y, z = 0, 0, 1
    while i < 4 and j < 4 and k < 4 :
        if ((x, y, z, i, j, k) != self.switched_coordinate) :
            successor = Cube(np.copy(self.state))
            successor.swap_two_elements(i, j, k, x, y, z)
            successor.switched_coordinate = (i, j, k, x, y, z)
            successors.append(successor)
        if (z < 4) :
            z += 1
        elif (y < 4 and z == 4) :
            y += 1
            z = 0
        elif (y == 4 and z == 4) :
            if (x < 4) :
                x += 1
                y = 0
                z = 0
            else : # x == 4
                if (k < 4) :
                    k += 1
                elif (j < 4 and k == 4) :
                    j += 1
                    k = 0
                elif (j == 4 and k == 4) :
                    if (i < 4) :
                        j = 0
                        k = 0
                    i += 1
                    x, y, z = i, j, k + 1
    return successors

```

- i. Fungsi `find_random_successor` memilih dan mengembalikan neighbor secara random, yakni dengan melakukan `swap_two_elements` pada koordinat-koordinat yang *random* pada kubus.

```
def find_random_successor(self) :
    i1, j1, k1 = random.randint(0, 4), random.randint(0,
4), random.randint(0, 4)
    i2, j2, k2 = random.randint(0, 4), random.randint(0,
4), random.randint(0, 4)

    while ((i1, j1, k1) == (i2, j2, k2)) :
        i2, j2, k2 = random.randint(0, 4),
random.randint(0, 4), random.randint(0, 4)

    successor = Cube(np.copy(self.state))
    successor.swap_two_elements(i1, j1, k1, i2, j2, k2)

    return successor
```

## B. Steepest Ascent Hill-Climbing

Algoritma *steepest ascent hill-climbing* merupakan algoritma *local search* paling dasar yang diawali dengan *initial state* yang acak (*randomly generated*). Pada setiap iterasinya, algoritma ini akan mencari *successor* terbaik dari *current state* berdasarkan *objective function* yang berlaku. *Successor* terbaik didefinisikan sebagai *successor* dengan nilai *state value* yang lebih tinggi (*objective*) atau lebih rendah (*cost*). Algoritma ini akan mengembalikan cube optimum lokal dengan harapan optimum lokal tersebut merupakan solusi global optimal.

Berikut langkah-langkah yang dapat digunakan untuk menemukan konfigurasi *diagonal magic cube* yang tepat, yang diimplementasikan pada kelas *SteepestHillClimb* yaitu:

1. Inisialisasi dimulai dengan suatu state kubus random ataupun memuat hasil kubus pada iterasi sebelumnya ataupun yang disimpan. Kelas ini memiliki atribut `current_cube` (menyimpan state kubus berjalan), `history` (array dari struktur json terkait data setiap iterasi yang akan disimpan), `initial_state` (0 ataupun memuat dari data), serta `final_iteration` atau jumlah iterasi hingga algoritma ini berhenti (menemukan solusi atau gagal mencapai optimum global),

```

import json
import sys
import time

class SteepestHillClimb:
    def __init__(self, cube, history=None,
initial_iteration=0):
        self.current_cube = cube

        if not history:
            self.history = [{ "iteration": 0,
                "state": [[row.tolist() for row in layer]
for layer in self.current_cube.state],
                "fitness_value":
int(self.current_cube.fitness_value)
            }]
            self.initial_iteration = 0

        else:
            with open("result/" + history, "r") as f:
                self.history = json.load(f)

            self.history.append({
                "iteration": initial_iteration + 1,
                "state": [[row.tolist() for row in
layer] for layer in self.current_cube.state],
                "fitness_value":
int(self.current_cube.fitness_value)
            })
            self.initial_iteration = initial_iteration + 1

```

2. Untuk setiap pasangan angka di kubus, tukar angka tersebut dan hitung *state value*-nya dengan menggunakan fungsi objektif. Ini diimplementasikan pada fungsi generate\_successors pada kelas kubus yang telah didefinisikan sebelumnya.
3. Pilih *successor* dengan *state value* terbaik (semakin mendekati nol, semakin minimum) sebagai *neighbor*.

```

def find_best_successor(self):
    best_successor = None
    best_fitness = float('inf')

    for successor in
self.current_cube.generate_successors():
        if successor.fitness_value < best_fitness:
            best_successor = successor
            best_fitness = successor.fitness_value

    return best_successor

```

4. Jika *neighbor* memiliki nilai *state value* yang sama atau lebih besar dari *current state*, maka *current state* akan dipilih sebagai solusi optimal.

5. Jika *neighbor* memiliki nilai *state value* yang lebih rendah dari *current state*, maka *neighbor* akan menjadi *current state* dan langkah-langkah akan diulangi kembali. Langkah umum algoritma yaitu pada kode berikut (kode juga dapat menyimpan state iterasi hill climbing pada file json).

```

def climb(self, output_file, max_iterations=1000,
initial_iteration=0):
    if initial_iteration > 0:
        x = self.initial_iteration + 1
    else:
        x = 1

    fitness_value_per_iteration = {}
    sys.stdout.write("Loading...\n")
    sys.stdout.flush()

    start_time = time.time()

    iteration = 0
    fitness_value_per_iteration[iteration + x] =
self.current_cube.fitness_value
    while iteration < max_iterations:
        best_successor = self.find_best_successor()

        if best_successor.fitness_value >=
self.current_cube.fitness_value:
            break

        self.current_cube = best_successor
        fitness_value_per_iteration[iteration + x] =
self.current_cube.fitness_value

        if output_file is not None:
            self.history.append({
                "iteration": iteration + x,
                "state": [[row.tolist() for row in
layer] for layer in self.current_cube.state],
                "fitness_value":
int(self.current_cube.fitness_value)
            })

        with open("result/" + output_file, "w") as f:
            json.dump(self.history, f, indent=4)

        if self.current_cube.fitness_value == 0:
            break

        if iteration % 10 == 0:
            sys.stdout.write("\rCurrent iteration:
{}".format(iteration))
            sys.stdout.flush()
            time.sleep(0.1)

        iteration += 1

    finish_time = time.time()
    sys.stdout.write("\r" + " " * 50 + "\r")

```

```

        sys.stdout.write("\033[F" + " " * 50 + "\r")

        self.final_iteration = iteration +
self.initial_iteration
            return self.current_cube, iteration,
fitness_value_per_iteration, (finish_time - start_time)

    def get_final_iteration(self):
        return self.final_iteration

```

### C. Hill-climbing with Sideways Move

Algoritma *hill-climbing with sideways move* merupakan modifikasi dari algoritma *steepest ascent hill-climbing* dalam upaya tidak terjadi ‘stuck’ pada optimum lokal yang rata. Pada algoritma ini, masih mungkin menjadikan *neighbor* sebagai *current* jika *neighbor* tersebut memiliki *state value* yang lebih tinggi atau sama dengan *current state*.

Berikut langkah-langkah yang digunakan untuk menemukan konfigurasi *diagonal magic cube* yang tepat dan sangat serupa dengan algoritma Steepest Ascent Hill Climbing, yang diimplementasikan pada kelas SidewaysHillClimb yaitu:

1. Inisialisasi dimulai dengan suatu state kubus random ataupun memuat hasil kubus pada iterasi sebelumnya ataupun yang disimpan (sama seperti pada kelas SteepestHillClimb)
2. Untuk setiap pasangan angka di kubus, tukar angka tersebut dan hitung *state value*-nya dengan menggunakan fungsi objektif.
3. Pilih *successor* dengan *state value* terbaik (semakin mendekati nol) sebagai *neighbor*.
4. Jika *neighbor* memiliki nilai *state value* yang lebih besar dari *current state*, maka *current state* akan dipilih sebagai solusi optimal.
5. Jika *neighbor* memiliki nilai *state value* yang lebih rendah **atau sama dari *current state***, maka *neighbor* akan menjadi *current state* dan langkah-langkah akan diulangi kembali. Pada algoritma ini juga dibatasi maksimal iterasi agar jika pencarian stuck pada shoulder juga tetap akan dihentikan setelah beberapa kali iterasi.

Berikut kode umum algoritma Hill Climbing with Sideways Move. Kode bercetak tebal menunjukkan perbedaan utama dengan algoritma Steepest Ascent Hill Climbing.

```

import json
import sys
import time

class SidewaysHillClimb:
    def __init__(self, cube):
        self.current_cube = cube
        self.history = [{{
            "iteration": 0,
            "state": [[row.tolist() for row in layer] for layer in
self.current_cube.state],
            "fitness_value": int(self.current_cube.fitness_value)
        }]

    def find_best_successor(self):
        best_successor = None
        best_fitness = float('inf')

        for successor in self.current_cube.generate_successors():
            if successor.fitness_value < best_fitness:
                best_successor = successor
                best_fitness = successor.fitness_value

        return best_successor

    def climb(self, output_file, max_iterations=1000):
        iteration = 0
        fitness_value_per_iteration = {}

        sys.stdout.write("Loading...\n")
        sys.stdout.flush()

        start_time = time.time()

        fitness_value_per_iteration[iteration] =
self.current_cube.fitness_value
        while iteration < max_iterations:
            best_successor = self.find_best_successor()

            if best_successor.fitness_value >
self.current_cube.fitness_value:
                break

            self.current_cube = best_successor
            fitness_value_per_iteration[iteration] =
self.current_cube.fitness_value

            if output_file:
                self.history.append({
                    "iteration": iteration + 1,
                    "state": [[row.tolist() for row in layer] for
layer in self.current_cube.state],
                    "fitness_value":
int(self.current_cube.fitness_value)
                })

```

```

        })

        with open("result/" + output_file, "w") as f:
            json.dump(self.history, f, indent=4)

        if self.current_cube.fitness_value == 0:
            break

        if iteration % 10 == 0:
            sys.stdout.write("\rCurrent iteration:\n{}"
                           .format(iteration))
            sys.stdout.flush()
            time.sleep(0.1)

        iteration += 1

        finish_time = time.time()
        sys.stdout.write("\r" + " " * 50 + "\r")
        sys.stdout.write("\033[F" + " " * 50 + "\r")

    return self.current_cube, iteration,
fitness_value_per_iteration, (finish_time - start_time)

```

#### D. Random Restart Hill-climbing

Algoritma *random restart hill-climbing* merupakan algoritma *hill-climbing* yang menghindari jebakan optimum lokal dengan melakukan *restart* dari solusi acak baru setelah mencapai solusi lokal. Algoritma ini mengulangi proses *hill-climbing* dari titik awal acak hingga beberapa kali dan menyimpan solusi terbaik yang ditemukan. Pencarian solusi dengan algoritma ini bersifat *complete* jika diberikan percobaan yang tak terbatas atau jumlah percobaan yang cukup banyak. Pada program kami, dibuat nilai maksimal restart default 20 dan maksimal iterasi 1000 atau berdasarkan variabel yang diinput pengguna.

Berikut langkah-langkah yang digunakan untuk menemukan konfigurasi *diagonal magic cube* yang mengulang-ulang algoritma Steepest Ascent Hill Climbing, yang diimplementasikan pada kelas RandomRestartHillClimb yaitu:

1. Inisialisasi dimulai dengan suatu state kubus dari parameter (di-*pass* dari main.py). Kelas ini juga menyimpan state kubus terbaik dari seluruh restart pencarian beserta nilai fitness terbaik dari state kubus terbaik tersebut.

```

import time
import sys
from cube import Cube

```

```

from steepest_hill_climb import SteepestHillClimb

class RandomRestartHillClimb:
    def __init__(self, cube):
        self.initial_cube = cube
        self.best_cube = None
        self.best_fitness = float('inf')

```

2. Pencarian dilakukan dengan dengan algoritma *steepest-ascent hill-climbing* hingga sudah ditemukannya solusi atau telah dilakukan restart sebanyak input parameter dengan maksimum iterasi yang juga dari input parameter. Jika solusi tidak ditemukan, maka state yang dijadikan solusi adalah state dengan value terbaik dari seluruh restart dan iterasi.

```

def climb(self, output_file=None, max_restarts=20,
max_iterations=1000):
    iterations_per_restart = []
    fitness_value_per_iteration = {}
    current_iteration = 0

    start_time = time.time()

    for restart in range(max_restarts):
        sys.stdout.write("\rRestart:
{} \n".format(restart))
        sys.stdout.flush()
        if restart > 0:
            current_cube = Cube()
            climber = SteepestHillClimb(current_cube,
output_file, current_iteration)
        else:
            current_cube = self.initial_cube
            climber = SteepestHillClimb(current_cube)

            result, _, per_iteration, _ =
climber.climb(output_file, max_iterations,
current_iteration)
            fitness_value_per_iteration[restart+1] =
per_iteration

            current_iteration =
climber.get_final_iteration()
            iterations_per_restart.append([restart+1,
current_iteration])

            if result.fitness_value < self.best_fitness:
                self.best_cube = result
                self.best_fitness = result.fitness_value

            if self.best_fitness == 0:
                sys.stdout.write("\033[F" + " " * 50 +
"\r")
                break

    sys.stdout.write("\033[F" + " " * 50 + "\r")

```

```

        finish_time = time.time()

        return self.best_cube, current_iteration,
iterations_per_restart, fitness_value_per_iteration,
(finish_time - start_time)

```

## E. Stochastic Hill-climbing

Algoritma *stochastic hill-climbing* merupakan algoritma *hill-climbing* yang memilih *neighbor* secara acak untuk melanjutkan proses pencarian, alih-alih selalu memilih successor terbaik. Jika *neighbor* tersebut memiliki nilai yang lebih baik, maka algoritma akan bergerak ke sana. Hal ini membuat algoritma lebih fleksibel dalam menghindari jebakan optimum lokal dibandingkan algoritma *hill-climb* biasa karena tidak hanya bergantung pada peningkatan *state value* terbesar. Berbeda dari algoritma *hill-climb* yang telah dijelaskan sebelumnya, pada *looping* algoritma ini, *successor* tidak seluruhnya dibangkitkan dan pemilihan *neighbor* dilakukan secara acak.

Berikut langkah-langkah yang digunakan untuk menemukan konfigurasi *diagonal magic cube* yang tepat yang diimplementasikan pada kelas *StochasticHillClimb* yaitu:

1. Inisialisasi dimulai dengan suatu state kubus random ataupun memuat hasil kubus pada iterasi sebelumnya ataupun yang disimpan (sama seperti pada kelas *SteepestHillClimb*)
2. Untuk setiap pasangan angka di kubus, tukar angka tersebut dan hitung *state value*-nya dengan menggunakan fungsi objektif.
3. Pilih *successor* secara acak dari sebagian *successor* yang dibangkitkan sebagai *neighbor*.
4. Jika *neighbor* memiliki nilai *state value* yang lebih besar dari *current state*, maka *current state* tidak diganti dan tetap digunakan untuk iterasi berikutnya.
5. Jika *neighbor* memiliki nilai *state value* yang lebih rendah atau sama dari *current state*, maka *neighbor* akan menjadi *current state* dan langkah-langkah akan diulangi kembali dengan pengulangan maksimal sebanyak maksimum iterasi sesuai input parameter.

Berikut kode umum algoritma Stochastic Hill Climbing. Kode bercetak tebal menunjukkan inti atau pembeda unik dari algoritma ini.

```
import json
import sys
import time

class StochasticHillClimb:
    def __init__(self, cube):
        self.current_cube = cube
        self.history = [{{
            "iteration": 0,
            "state": [[row.tolist() for row in layer] for
layer in self.current_cube.state],
            "fitness_value":
int(self.current_cube.fitness_value)
        }]

    def climb(self, output_file, max_iterations=1000):
        iteration = 0

        sys.stdout.write("Loading...\n")
        sys.stdout.flush()

        start_time = time.time()

        while iteration < max_iterations:
            neighbor =
self.current_cube.find_random_successor()

            if neighbor.fitness_value <
self.current_cube.fitness_value:
                self.current_cube = neighbor

            if output_file is not None:
                self.history.append({{
                    "iteration": iteration + 1,
                    "state": [[row.tolist() for row in
layer] for layer in self.current_cube.state],
                    "fitness_value":
int(self.current_cube.fitness_value)
                }})

                with open("result/" + output_file, "w") as f:
                    json.dump(self.history, f, indent=4)

            if self.current_cube.fitness_value == 0:
                break

            if iteration % 10 == 0:
                sys.stdout.write("\rCurrent iteration:
{}.".format(iteration))
                sys.stdout.flush()
                time.sleep(0.1)

            iteration += 1

        finish_time = time.time()
        sys.stdout.write("\r" + " " * 50 + "\r")
```

```

        sys.stdout.write("\033[F" + " " * 50 + "\r")

    return self.current_cube, iteration, (finish_time -
start_time)

```

## F. Simulated Annealing

Algoritma *simulated annealing* merupakan algoritma *local search* yang umum digunakan untuk menyelesaikan permasalahan optimasi. Algoritma *simulated annealing* terinspirasi oleh teknik *simulated annealing* dalam bidang metalurgi yang menyatakan bahwa pada proses pembentukan kristal, perlu dilakukan pemanasan materi di awal yang dilanjutkan proses pendinginan perlahan-lahan untuk memberikan fleksibilitas lebih bagi atom-atom sehingga dapat menemukan tempat optimum, di mana energi internal yang dibutuhkan atom untuk mempertahankan posisinya bernilai minimum. Dalam algoritma *simulated annealing* untuk permasalahan optimasi, proses pendinginan ini diwakilkan oleh dimungkinkan diterimanya kondisi (*state*) yang memiliki nilai lebih buruk dibandingkan nilai kondisi permasalahan yang sedang ditinjau sebagai kondisi berikutnya melalui probabilitas yang nilainya ditentukan dengan

$$P = \begin{cases} 1 & \text{if } \Delta c \leq 0 \\ e^{-\Delta c / t} & \text{if } \Delta c > 0 \end{cases}$$

Nilai dari  $\Delta c$  melambangkan selisih antara nilai state hasil generasi random yang akan ditinjau kelayakannya untuk menjadi *neighbor* dengan nilai state yang sedang ditinjau. Nilai  $t$  melambangkan temperatur dari state yang akan terus mengalami pengurangan nilai seiring berjalannya iterasi. Penurunan nilai temperatur ditentukan oleh fungsi temperatur. Selain itu, penurunan nilai temperatur yang terjadi seiring iterasi membuat nilai probabilitas semakin lama akan semakin kecil.

Penentuan fungsi temperatur yang kerap disebut sebagai *cooling schedule* merupakan salah satu komponen penting dalam proses algoritma *simulated annealing*. Fungsi temperatur yang tepat tentu memberikan hasil akhir yang

maksimal untuk setiap fungsi objektif yang digunakan pada algoritma *simulated annealing*. Secara umum, terdapat empat fungsi temperatur yang lazim digunakan dalam menentukan fungsi temperatur untuk algoritma *simulated annealing* :

- Linear Multiplicative Monotonic Cooling Schedule

$$T(k) = T_{max} - \alpha k$$

- Quadratic Multiplicative Monotonic Cooling Schedule

$$T(k) = \frac{T_{max}}{1 + \alpha k^2}$$

- Logarithmic Multiplicative Monotonic Cooling Schedule

$$T(k) = \frac{T_{max}}{1 + \alpha \log(k + 1)}$$

- Exponential Multiplicative Monotonic Cooling Schedule

$$T(k) = T_{max} \alpha^k$$

Hasil observasi literatur yang didapat dari situs berjudul “Effective Simulated Annealing with Python” menyebutkan bahwa penggunaan fungsi temperatur yang tepat bergantung pada bagaimana fungsi objektif dapat memetakan kondisi dari suatu permasalahan yang ada. Apabila suatu fungsi objektif dapat memetakan permasalahan sehingga hasil representasi state-nya bersifat unimodal (kecil kemungkinan memiliki kondisi lokal optimum sehingga lebih optimum mengantarkan solusi permasalahan ke global optimum), fungsi temperatur yang paling sesuai adalah Logarithmic Multiplicative Monotonic Cooling Schedule dan Exponential Multiplicative Monotonic Cooling Schedule. Apabila suatu fungsi objektif memetakan permasalahan sehingga hasil representasi state-nya bersifat multimodal (besar kemungkinan memiliki beberapa kondisi lokal optimum

sebelum mencapai global optimum), fungsi temperatur yang paling sesuai digunakan adalah Linear Multiplicative Monotonic Cooling Schedule dan Quadratic Multiplicative Monotonic Cooling Schedule.

Nilai  $\alpha$  pada fungsi temperatur merupakan *acceptance constant* yang merepresentasikan tingkatan penurunan dari temperatur pada setiap iterasi bergantung dari fungsi temperatur yang ditetapkan.

Pada kasus magic cube yang diselesaikan dalam laporan ini, penulis mengamati bahwa fungsi objektif yang digunakan untuk merepresentasikan nilai dari *state* bersifat multimodal setelah melakukan beberapa kali percobaan dengan menggunakan algoritma Steepest Ascent. Oleh karena itu, penulis berupaya membandingkan fungsi temperatur mana yang paling baik digunakan sehingga dapat mengupayakan dikembalikannya nilai *state* optimum untuk setiap penyelesaian permasalahannya. Setelah melakukan observasi dengan menggunakan sepuluh initial state yang berbeda, didapatkan Quadratic Multiplicative Monotonic Cooling Schedule mendukung pengembalian hasil terbaik dari algoritma *simulated annealing*.

Kemudian, dalam peninjauan nilai probabilitas, terdapat nilai batasan (*threshold*) yang digunakan sebagai acuan apakah *successor* dengan nilai lebih buruk itu akan diterima sebagai *neighbor* dari *state* saat ini atau tidak. Semakin mengecilnya nilai probabilitas yang dihasilkan akan membuat tingkat penerimaan *successor* dengan nilai yang lebih buruk semakin rendah. Kemungkinan penerimaan *successor* dengan nilai yang lebih buruk ini membuat algoritma *simulated annealing* lebih fleksibel dalam menghindari jebakan optimum lokal dibandingkan algoritma *hill-climb* karena algoritma *simulated annealing* memiliki mekanisme untuk keluar dari kondisi lokal optimum dengan memasuki kondisi *successor* yang bernilai lebih buruk terlebih dahulu sehingga pergerakannya memperbesar kemungkinan didapatkannya solusi yang bernilai global optimum.

Berikut langkah-langkah yang dapat digunakan untuk menemukan konfigurasi *diagonal magic cube* yang tepat, yang diimplementasikan pada kelas SimulatedAnnealing yaitu :

1. Inisialisasi dimulai dengan suatu state kubus random ataupun memuat hasil kubus pada iterasi sebelumnya ataupun yang disimpan bersamaan dengan didapatkannya juga informasi nilai batasan penerimaan hasil penghitungan probabilitas *successor state* yang lebih buruk dari *current state*. Kelas SimulatedAnnealing juga menyimpan beberapa informasi relevan, seperti jumlah iterasi maksimum untuk membatasi kasus fungsi temperatur logaritmik, eksponensial, dan kuadratik yang tidak akan menyentuh angka 0, serta temperatur awal, nilai probabilitas di setiap iterasi, hingga jumlah penerimaan *successor state* yang lebih buruk dari hasil perhitungan nilai probabilitas.

```
import math
import json
import sys
import time

class SimulatedAnnealing :
    def __init__(self, cube):
        self.initial_cube = cube
        self.max_iterations = 2000
        self.initial_temperature = 100
        self.history = [{{
            "iteration": 0,
            "state": [[row.tolist() for row in layer] for
layer in self.initial_cube.state],
            "fitness_value":
int(self.initial_cube.fitness_value),
        }]}
        self.probability_values = []
        self.probability_iterations = []
        self.worse_moves_accepted = 0
```

2. Untuk setiap pasangan angka di kubus, tukar angka tersebut dan hitung *state value*-nya dengan menggunakan fungsi objektif.
3. Pilih *successor* secara acak dari *successor* yang dibangkitkan sebagai *neighbor*.
4. Jika *neighbor* memiliki *state value* yang lebih besar dari *current state*, lakukan peninjauan nilai probabilitas *neighbor state*. Pada program yang dibuat, nilai probabilitas akan dihitung dengan fungsi *probability\_function* yang menerima masukan selisih nilai state dan temperatur dari *state* saat ini.

```

def probability_function(delta_e, temperature) :
    return math.exp((-1) * delta_e) / temperature

```

- Jika nilai probabilitas *neighbor state* lebih besar daripada atau sama dengan *current state*, *neighbor state* akan dipilih sebagai *current state* untuk iterasi berikutnya.

```

if state_value_difference < 0 :
    current_state = neighbor
else :
    probability =
SimulatedAnnealing.probability_function(state_value_difference, temperature)
    if (probability >= threshold) :
        current_state = neighbor

self.probability_values.append(probability)
self.probability_iterations.append(i)
self.worse_moves_accepted += 1

```

- Jika *neighbor* memiliki nilai *state value* yang lebih rendah dari *current state*, maka *neighbor* akan menjadi *current state* pada iterasi berikutnya.
- Pada setiap tahapan iterasi, besaran temperatur akan mengecil sesuai dengan fungsi temperatur (*cooling schedule*) yang telah ditetapkan.

```

def temperature_function(self, choice, iteration) :
    result = 0
    if (choice == 1) :
        result = self.initial_temperature / (1 + 0.05 * iteration * iteration)
    elif (choice == 2) :
        result = self.initial_temperature - (0.05 * iteration)
    elif (choice == 3) :
        result = self.initial_temperature / (1 + 0.05 * math.log(iteration + 1))
    elif (choice == 4) :
        result = self.initial_temperature *
math.pow(0.05, iteration)
    else :
        result = self.initial_temperature / (1 + 0.05 * iteration * iteration)
    return result

```

- Langkah-langkah akan diulangi kembali hingga temperatur bernilai 0 atau pengulangan sudah terjadi sebanyak maksimum iterasi.

Berikut kode umum algoritma Simulated Annealing :

```
import math
import json
import sys
import time

class SimulatedAnnealing :
    def __init__(self, cube):
        self.initial_cube = cube
        self.max_iterations = 2000
        self.initial_temperature = 100
        self.history = [{ "iteration": 0,
                          "state": [[row.tolist() for row in layer] for
                                    layer in self.initial_cube.state],
                          "fitness_value": int(self.initial_cube.fitness_value),
                        }]
        self.probability_values = []
        self.probability_iterations = []
        self.worse_moves_accepted = 0

    def temperature_function(self, choice, iteration) :
        result = 0
        if (choice == 1) :
            result = self.initial_temperature / (1 + 0.05 *
iteration * iteration)
        elif (choice == 2) :
            result = self.initial_temperature - (0.05 *
iteration)
        elif (choice == 3) :
            result = self.initial_temperature / (1 + 0.05 *
math.log(iteration + 1))
        elif (choice == 4) :
            result = self.initial_temperature *
math.pow(0.05, iteration)
        else :
            result = self.initial_temperature / (1 + 0.05 *
iteration * iteration)
        return result

    def probability_function(delta_e, temperature) :
        return math.exp((-1) * delta_e) / temperature

    def simulated_annealing_algorithm(self, threshold,
output_file) :
        current_state = self.initial_cube
        i = 0

        sys.stdout.write("Loading...\n")
        sys.stdout.flush()

        start_time = time.time()

        print("Choose cooling schedule you want to use from
options below!\n1. Default (Quadratic Multiplicative
```

```

Monotonic)\n2. Linear Multiplicative Monotonic\n3.
Logarithmic Multiplicative\n4. Exponential Multiplicative
Monotonic")
        temp_function_choice = int(input("Input your choice
(1-4) : "))
        print("")

        while (True and i < self.max_iterations) :
            temperature =
SimulatedAnnealing.temperature_function(self,
temp_function_choice, i)
            if (temperature == 0) :
                return current_state, i, (finish_time -
start_time)

            neighbor =
current_state.find_random_successor()

            state_value_difference =
neighbor.calculate_fitness()[0] -
current_state.calculate_fitness()[0]

            if state_value_difference < 0 :
                current_state = neighbor
            else :
                probability =
SimulatedAnnealing.probability_function(state_value_difference,
temperature)
                if (probability >= threshold) :
                    current_state = neighbor

self.probability_values.append(probability)
self.probability_iterations.append(i)
self.worse_moves_accepted += 1

            if output_file:
                self.history.append({
                    "iteration": i + 1,
                    "state": [[row.tolist() for row in
layer] for layer in current_state.state],
                    "fitness_value":
int(current_state.fitness_value)
                })

            with open("result/" + output_file, "w") as f:
                json.dump(self.history, f, indent=4)

            if i % 10 == 0:
                sys.stdout.write("\rCurrent iteration:
{}".format(i))
                sys.stdout.flush()
                time.sleep(0.1)

            i += 1

            finish_time = time.time()
            sys.stdout.write("\r" + " " * 50 + "\r")
            sys.stdout.write("\033[F" + " " * 50 + "\r")

        return current_state, i, (finish_time - start_time)

```

## G. Genetic Algorithm

Genetic Algorithm (GA) adalah algoritma yang terinspirasi oleh proses seleksi alam dalam biologi. Algoritma ini bekerja dengan menghasilkan sekumpulan solusi awal yang disebut populasi, di mana setiap solusi dianggap sebagai individu yang diwakili oleh serangkaian parameter. GA menggunakan mekanisme evolusi seperti seleksi, *crossover* (rekombinasi), dan mutasi untuk menghasilkan solusi yang lebih baik dari generasi (iterasi) ke generasi.

Genetic Algorithm mirip dengan *random restart hill-climbing* dalam hal eksplorasi ruang solusi secara acak untuk menghindari jebakan optimum lokal, tetapi GA melakukannya secara paralel daripada secara sekuensial. GA melacak beberapa solusi sekaligus dengan memilih beberapa penerus terbaik ( $k$  successors) di setiap langkah. Ini memungkinkan eksplorasi ruang solusi yang lebih efisien. Kemudian dari  $k$  successors tersebut dipilih penerus berdasarkan probabilitas, bukan secara deterministik memilih yang terbaik sehingga lebih baik dalam menghindari optimum lokal. Algoritma ini berjalan secara paralel serta berbagi informasi berguna antar generasi sehingga diklaim lebih efektif dalam menjelajahi ruang solusi.

Berikut merupakan langkah-langkah dan kode untuk menemukan solusi permasalahan *diagonal magic cube* dengan menggunakan algoritma ini:

1. [Representasi State] Pada permasalahan *diagonal magic cube* ini, status setiap individu (kromosom) dapat direpresentasikan dalam kelas kubus Cube yang sudah kita gunakan pada seluruh algoritma *local search* lainnya.
2. [Populasi Awal] Pada awal dimulainya GA ini, akan dipilih beberapa individu kubus secara acak sebagai populasi, di mana setiap individu merupakan permutasi acak dari angka 1 hingga 125 yang diisi dalam kubus. Jumlah populasi ini akan menjadi variabel kontrol untuk analisis algoritma ini.

```
def initialize_population(population_size):
    """Initialize a population of Cube objects with random
```

```

states."""
    return [Cube() for _ in range(population_size)]

```

3. [Penghitungan Nilai Fungsi Objektif] Untuk setiap individu pada populasi awal, dihitung nilai fungsi objektifnya (*fitness value*) berdasarkan fungsi objektif yang telah ditetapkan. Nilai *fitness value* menjadi atribut setiap Cube dengan fungsi objektif yang sudah dijelaskan.
4. [Seleksi] Seleksi akan dilakukan dengan *random roulette wheel*. Pada *roulette wheel*, setiap individu berpartisipasi seluas tingkat *fitness value*-nya. Persentase ukuran luas pada *roulette wheel* ditentukan berdasarkan nilai *fitness value* individu tersebut dibagi dengan jumlah nilai *fitness value* individu pada populasi (dibantu diperoleh dengan np.cumsum pada kode di bawah). Dengan *roulette wheel* seperti ini, individu dengan nilai fungsi objektif yang lebih baik akan lebih mungkin terpilih sebagai calon orang tua.

```

def roulette_wheel_selection(population):
    """Select a parent from the population using roulette
    wheel selection based on fitness."""
    total_fitness = sum(cube.fitness_value for cube in
population)
    relative_fitness = [round(cube.fitness_value * 100 /
total_fitness, 2) for cube in population]
    cumulative_probabilities = np.cumsum(relative_fitness)

    rand = random.random() * 100
    for i, cumulative_probability in
enumerate(cumulative_probabilities):
        if rand <= cumulative_probability:
            return population[i]
    return population[-1]

```

5. Setelah itu pada setiap seleksi di algoritma ini akan dipilih populasi/pasangan orang tua menggunakan *roulette wheel*, ini akan memungkinkan eksplorasi ruang solusi dan juga memungkinkan mengeliminasi kubus yang memiliki *fitness value* yang rendah dari populasi.
6. [*Crossover*] Kemudian, dari pasangan orang tua yang sudah dipilih, dilakukan *crossover* untuk membuat keturunan dengan menggabungkan bagian-bagian dari kromosom mereka. Untuk *crossover* ini, digunakan heuristik agar crossover dapat menghasilkan hasil yang optimal, yakni dengan melakukan *crossover* dari beberapa line yang sudah baik pada

kubus satu dengan line yang buruk pada kubus lainnya. Oleh karenanya, teknik crossover ini mungkin terjadi di beberapa koordinat, arah, dan posisi yang berbeda pada kedua kubus. Teknik *crossover* ini dapat membuka ruang solusi baru dengan harapan memiliki nilai fungsi objektif yang lebih baik.

Pertama-tama, disiapkan anak yang merupakan copy dari orang tuanya, array yang menyimpan indeks crossover dari masing-masing individu yang bertukar, serta mapping dari angka lama ke angka baru	<pre>def crossover_optimized(parent1, parent2):     """partially mapped crossover """     child1 = np.copy(parent1.state)     child2 = np.copy(parent2.state)      # PMX mappings for the area     mapping1 = {}     mapping2 = {}     crossover_index_parent1 = []     crossover_index_parent2 = []</pre>
Karena pada masing-masing cube terdapat 3 best dan worst lines, maka akan terdapat 3 crossover antara best lines dari parent2 dan worst lines dari parent1.  Misal percabangan pertama di mana worst line parent1 baris, maka nilai awal child1 akan disimpan dalam ori_val, kemudian child1 akan diubah nilai nya dengan nilai yang lebih baik dari best_line parent2.  Selaun itu, juga disimpan nilai crossover index untuk parent 1, dan nilai mapping1 dan mapping2 yang nantinya akan memetakan angka awal apa, berubah menjadi apa (karena bertukar).	<pre>for crossover_idx in range(3):     # Replace in child1 using parent2's best line     worst_line_indices_parent1 = parent1.worst_lines[crossover_idx][1]     best_line_indices_parent2 = parent2.best_lines[crossover_idx][1]      if worst_line_indices_parent1[2] == -1: # worst line is a row         for k in range(5):             ori_val = child1[worst_line_indices_parent1[0], worst_line_indices_parent1[1], k]             child1[worst_line_indices_parent1[0], worst_line_indices_parent1[1], k] = parent2.state[best_line_indices_parent2[0], best_line_indices_parent2[1], k]      crossover_index_parent1.append((worst_line_indices_parent1[0] ], worst_line_indices_parent1[1], k))         if (ori_val != parent2.state[best_line_indices_parent2[0], best_line_indices_parent2[1], k]):             mapping1[ori_val] = parent2.state[best_line_indices_parent2[0], best_line_indices_parent2[1], k]  mapping2[parent2.state[best_line_indices_parent2[0], best_line_indices_parent2[1], k]] = ori_val      elif worst_line_indices_parent1[1] == -1: # worst line is a column         for j in range(5):             ori_val = child1[worst_line_indices_parent1[0], j, worst_line_indices_parent1[2]]             child1[worst_line_indices_parent1[0], j, worst_line_indices_parent1[2]] = parent2.state[best_line_indices_parent2[0], j, best_line_indices_parent2[2]]      crossover_index_parent1.append((worst_line_indices_parent1[0]</pre>

	<pre> ], j, worst_line_indices_parent1[2]))         if (ori_val != parent2.state[best_line_indices_parent2[0], j, best_line_indices_parent2[2]]):             mapping1[ori_val] = parent2.state[best_line_indices_parent2[0], j, best_line_indices_parent2[2]]  mapping2[parent2.state[best_line_indices_parent2[0], j, best_line_indices_parent2[2]]] = ori_val          elif worst_line_indices_parent1[0] == -1: # Worst line is a pillar             for i in range(5):                 ori_val = child1[i, worst_line_indices_parent1[1], worst_line_indices_parent1[2]]                 child1[i, worst_line_indices_parent1[1], worst_line_indices_parent1[2]] = parent2.state[i, best_line_indices_parent2[1], best_line_indices_parent2[2]]                 crossover_index_parent1.append((i, worst_line_indices_parent1[1], worst_line_indices_parent1[2]))                 if (ori_val != parent2.state[i, best_line_indices_parent2[1], best_line_indices_parent2[2]]):                     mapping1[ori_val] = parent2.state[i, best_line_indices_parent2[1], best_line_indices_parent2[2]]                     mapping2[parent2.state[i, best_line_indices_parent2[1], best_line_indices_parent2[2]]] = ori_val </pre>
<p>Sama halnya dengan crossover dari best parent 2 ke worst child1, dilakukan sebaliknya best child 2 akan menerima nilai-nilai dari worst parent1</p> <p>Kemudian, juga disimpan nilai crossover index untuk parent 2, tempat penukaran tersebut</p>	<pre> if best_line_indices_parent2[2] == -1: # Best line is a row     for l in range(5):         ori_val = child2[best_line_indices_parent2[0], best_line_indices_parent2[1], 1]         child2[best_line_indices_parent2[0], best_line_indices_parent2[1], 1] = parent1.state[worst_line_indices_parent1[0], worst_line_indices_parent1[1], 1]  crossover_index_parent2.append((best_line_indices_parent2[0] , best_line_indices_parent2[1], 1))          elif best_line_indices_parent2[1] == -1: # Best line is a column             for m in range(5):                 ori_val = child2[best_line_indices_parent2[0], m, best_line_indices_parent2[2]]                 child2[best_line_indices_parent2[0], m, best_line_indices_parent2[2]] = parent1.state[worst_line_indices_parent1[0], m, worst_line_indices_parent1[2]]  crossover_index_parent2.append((best_line_indices_parent2[0] , m, best_line_indices_parent2[2]))          elif best_line_indices_parent2[0] == -1: # Best line is a pillar             for n in range(5):                 ori_val = child2[n, best_line_indices_parent2[1], best_line_indices_parent2[2]]                 child2[n, best_line_indices_parent2[1], </pre>

	<pre>best_line_indices_parent2[2]] = parent1.state[n, worst_line_indices_parent1[1], worst_line_indices_parent1[2]] crossover_index_parent2.append((n, best_line_indices_parent2[1], best_line_indices_parent2[2]))</pre>
Terakhir, nilai duplikat dari cube harus di-resolve. Baru direturn.	<pre>resolve_mapping_conflicts(child1, mapping1, set(crossover_index_parent1)) resolve_mapping_conflicts(child2, mapping2, set(crossover_index_parent2)) return Cube(child1), Cube(child2)</pre>

7. [Penanganan Crossover yang Konflik] Mengingat, batasan bahwa kubus haruslah bernilai 1 s.d. 125 dan unik, crossover dalam kondisi apapun hampir tak memungkinkan adanya duplikasi. Oleh karena itu, dengan nilai mapping1, mapping2, dan indeks crossover pada fungsi crossover, dilakukan pemanggilan fungsi penanganan konflik berikut. Untuk kubus child yang sudah dilakukan crossover sebelumnya, dilakukan iterasi seluruh elemen kubus kecuali titik-titik crossover (karena tentu saja titik crossover tidak ingin diubah kembali). Jika terdapat nilai yang berada pada mapping (duplikat), maka nilai di indeks tersebut akan diganti dengan nilai yang menghilangkannya pada crossover. Misalkan angka 5 pada cube[1,2,3] harus tergantikan oleh angka 9 karena crossover, maka akan dicari angka 9 pada sisi lain kubus untuk digantikan dengan angka 5.

```
def resolve_mapping_conflicts(child, mapping,
crossover_indices):
    visited = set()
    for index in np.ndindex(child.shape):
        if index in crossover_indices:
            continue

        value = child[index]
        while value in mapping and value not in visited:
            visited.add(value)
            value = mapping[value]

        if value in visited:
            break

        child[index] = value
```

8. [Mutasi] Untuk memastikan eksplorasi ruang solusi, dilakukan mutasi terhadap anak hasil crossover dengan menukar dua angka secara acak

dalam kubus. Pada pasangan kromosom, akan dilakukan mutasi dengan nilai yang diminta dari user.

```
def mutate(state):
    """Mutate a cube state by randomly swapping two
    elements."""
    x1, y1, z1 = random.randint(0, 4), random.randint(0,
4), random.randint(0, 4)
    x2, y2, z2 = random.randint(0, 4), random.randint(0,
4), random.randint(0, 4)

    state[x1, y1, z1], state[x2, y2, z2] = state[x2, y2,
z2], state[x1, y1, z1]

    return state
```

9. [Generasi Selanjutnya] Dari seluruh pasang anak hasil reproduksi orang tua, berarti menghasilkan individu baru sejumlah ukuran populasi. Untuk populasi baru yang akan melakukan ulang proses reproduksi dan eksplorasi, akan dipilih individu terbaik sejumlah ukuran populasi dari total populasi awal (parent) dan individu baru (child) yang memiliki nilai *fitness value* terbaik. Misalnya ada populasi berukuran 20, maka akan dipilih 20 individu terbaik dari 20 parent dan 20 child, untuk melakukan reproduksi selanjutnya.
10. [Penghentian Pencarian] Algoritma Genetic Algorithm ini akan dihentikan ketika memperoleh *diagonal magic cube* yang diharapkan atau mencapai maksimal iterasi yang merupakan parameter input. Kode

Berikut kode umum Genetic Algorithm.

```
def genetic_algorithm(population_size, max_iterations,
mutation_rate, output_file=None):
    # Initialize the population
    if output_file:
        history = []

    sys.stdout.write("Loading...\n")
    sys.stdout.flush()

    start_time = time.time()

    population = initialize_population(population_size)

    for iteration in range(max_iterations):
        # Generate a new population
        new_population = []
        while len(new_population) < population_size:
            # Select parents using roulette wheel selection
```

```

        parent1 = roulette_wheel_selection(population)
        parent2 = roulette_wheel_selection(population)
        print(f"parent1 {parent1.fitness_value} -
parent2 {parent2.fitness_value}")

        # Apply crossover to produce two children
        child1, child2 = crossover_optimized(parent1,
parent2)
        print(f"child {child1.fitness_value} -
{child2.fitness_value}")

        # Apply mutation to children
        if random.random() < mutation_rate:
            child1.state = mutate(child1.state)
            child1.fitness_value, _, _ =
child1.calculate_fitness()
            if random.random() < mutation_rate:
                child2.state = mutate(child2.state)
                child2.fitness_value, _, _ =
child2.calculate_fitness()

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

        # Select the best individuals from the parent and
        child population
        combined_population = population + new_population
        combined_population.sort(key=lambda cube:
cube.fitness_value)
        population = combined_population[:population_size]

        best_individual = population[0]

        if output_file:
            history.append({
                "iteration": iteration + 1,
                "state": [[row.tolist() for row in layer]
for layer in best_individual.state],
                "fitness_value":
int(best_individual.fitness_value)
            })

            with open("result/" + output_file, "w") as f:
                json.dump(history, f, indent=4)

        # if isGoalState(best_individual):
        if best_individual.fitness_value == 0:
            break

        if iteration % 10 == 0:
            sys.stdout.write("\rCurrent iteration:
{}".format(iteration))
            sys.stdout.flush()
            time.sleep(0.1)

        finish_time = time.time()
        sys.stdout.write("\r" + " " * 50 + "\r")
        sys.stdout.write("\033[F" + " " * 50 + "\r")

```

```
    return best_individual, iteration, (finish_time -  
start_time)
```

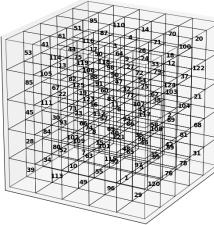
### 3. Hasil Eksperimen

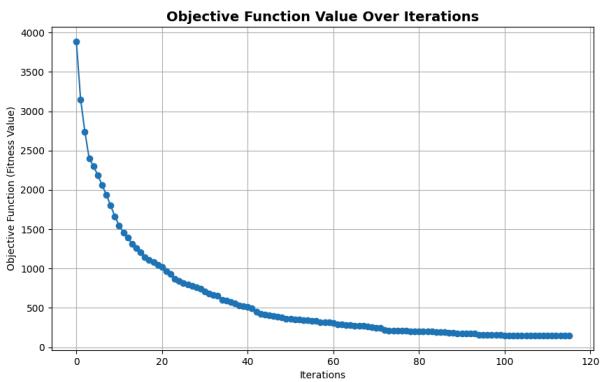
#### A. Steepest Ascent Hill-Climbing

##### 1. Eksperimen I

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[ 8 109 52 34 39]  
[121 101 63 10 113]  
[ 46 65 92 55 49]  
[ 59 99 5 1 96]  
[ 31 78 76 120 29]]  
  
Layer 2:  
[[ 16 23 93 84 28]  
[ 6 57 74 102 80]  
[117 86 40 15 7]  
[ 89 108 56 48 115]  
[ 68 81 42 35 97]]  
  
Layer 3:  
[[ 88 125 22 111 45]  
[ 47 60 123 73 30]  
[ 25 77 11 112 66]  
[103 38 43 94 69]  
[ 21 104 2 58 36]]  
  
Layer 4:  
[[ 17 75 13 105 85]  
[ 64 118 106 82 67]  
[ 24 27 90 91 19]  
[ 12 79 54 32 83]  
[122 37 124 62 107]]  
  
Layer 5:  
[[ 95 51 61 41 53]  
[110 87 116 44 114]  
[ 14 4 3 50 119]  
[ 70 71 26 9 98]  
[ 20 100 18 33 72]]  
  
Initial Fitness: 3886.38
```

	<p style="text-align: center;">Iteration 0 - Fitness: 3886.38</p> 
State Akhir Kubus	<p>Final cube state after climbing:</p> <p>Layer 1:</p> <pre>[[ 86 31 118 53 7]  [121 82 11 10 195]  [ 46 65 92 51 64]  [ 12 119 5 84 96]  [ 44 26 89 117 36]]</pre> <p>Layer 2:</p> <pre>[[ 35 23 112 122 28]  [ 6 45 74 80 94]  [123 76 52 15 42]  [ 78 108 56 48 37]  [ 68 81 16 41 114]]</pre> <p>Layer 3:</p> <pre>[[ 83 110 22 2 115]  [ 57 32 97 73 39]  [ 25 77 59 93 66]  [103 1 43 102 69]  [ 60 104 87 50 27]]</pre> <p>Layer 4:</p> <pre>[[ 17 75 13 120 88]  [106 49 34 40 67]  [ 91 111 90 21 19]  [ 71 79 54 55 47]  [ 24 14 116 62 107]]</pre> <p>Layer 5:</p> <pre>[[ 95 58 61 29 63]  [ 3 101 99 98 8]  [ 38 4 20 125 113]  [ 70 30 124 9 85]  [109 100 18 33 72]]</pre> <p>Final Fitness: 147.71</p>
Durasi Pencarian	44.19 detik

Banyak Iterasi	115																								
Plot Nilai Objektif	 <p>The plot shows the relationship between the number of iterations and the objective function value. The x-axis is labeled 'Iterations' and ranges from 0 to 120. The y-axis is labeled 'Objective Function (Fitness Value)' and ranges from 0 to 4000. The data points show a rapid initial decrease in the objective function value, followed by a more gradual convergence towards a minimum.</p> <table border="1"> <thead> <tr> <th>Iterations</th> <th>Objective Function (Fitness Value)</th> </tr> </thead> <tbody> <tr><td>0</td><td>3800</td></tr> <tr><td>1</td><td>3100</td></tr> <tr><td>2</td><td>2700</td></tr> <tr><td>5</td><td>2200</td></tr> <tr><td>10</td><td>1800</td></tr> <tr><td>20</td><td>1000</td></tr> <tr><td>40</td><td>500</td></tr> <tr><td>60</td><td>250</td></tr> <tr><td>80</td><td>150</td></tr> <tr><td>100</td><td>100</td></tr> <tr><td>115</td><td>100</td></tr> </tbody> </table>	Iterations	Objective Function (Fitness Value)	0	3800	1	3100	2	2700	5	2200	10	1800	20	1000	40	500	60	250	80	150	100	100	115	100
Iterations	Objective Function (Fitness Value)																								
0	3800																								
1	3100																								
2	2700																								
5	2200																								
10	1800																								
20	1000																								
40	500																								
60	250																								
80	150																								
100	100																								
115	100																								

## Bukti Eksperimen

```
Local Search Algorithms:  
1. Steepest Hill Climbing  
2. Hill Climbing with Sideways Move  
3. Random Restart Hill Climbing  
4. Stochastic Hill Climbing  
5. Simulated Annealing  
6. Genetic Algorithm  
7. Back to Main Menu  
Choose the algorithm you want to use: 1
```

```
Do you want to keep the replay of the cube solving process? (y/n): y  
Enter the output file name (without extension): steepest_1  
The file will be saved as 'steepest_1.json' in the 'result/' directory.
```

```
Input the maximum iteration (default 1000): 1000
```

```
Final cube state is reached after 115 iterations in 44.19 seconds.
```

```
Final cube state after climbing:
```

```
Layer 1:  
[[ 86 31 118 53 7]  
[121 82 11 10 105]  
[ 46 65 92 51 64]  
[ 12 119 5 84 96]  
[ 44 26 89 117 36]]
```

```
Layer 2:
```

```
[[ 35 23 112 122 28]  
[ 6 45 74 80 94]  
[123 76 52 15 42]  
[ 78 108 56 48 37]  
[ 68 81 16 41 114]]
```

```
Layer 3:
```

```
[[ 83 110 22 2 115]  
[ 57 32 97 73 39]  
[ 25 77 59 93 66]  
[103 1 43 102 69]  
[ 60 104 87 50 27]]
```

```
Layer 4:
```

```
[[ 17 75 13 120 88]  
[106 49 34 48 67]  
[ 91 111 90 21 19]  
[ 71 79 54 55 47]  
[ 24 14 116 62 107]]
```

```
Layer 5:
```

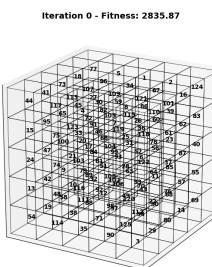
```
[[ 95 58 61 29 63]  
[ 3 101 99 98 8]  
[ 38 4 20 125 113]  
[ 70 30 124 9 85]  
[109 100 18 33 72]]
```

```
Final Fitness: 147.71
```

## 2. Eksperimen II

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[ 32 116 58 19 54]  
[106 12 88 38 114]  
[ 43 120 97 71 35]  
[ 68 50 94 125 90]  
[ 69 14 80 29 3]]  
  
Layer 2:  
[[ 84 103 9 42 13]  
[ 93 51 89 10 48]  
[122 63 102 92 111]  
[ 85 11 70 49 98]  
[ 55 57 76 22 112]]  
  
Layer 3:  
[[ 91 33 100 47 24]  
[119 66 17 21 74]  
[118 31 99 81 79]  
[ 23 56 6 28 108]  
[ 40 87 37 64 53]]  
  
Layer 4:  
[[ 27 45 65 95 15]  
[ 59 82 72 123 75]  
[ 86 115 7 46 20]  
[ 39 60 36 25 104]  
[ 83 62 61 78 52]]  
  
Layer 5:  
[[ 77 18 73 41 44]  
[ 5 96 107 113 117]  
[ 1 34 109 30 4]  
[ 2 67 121 8 105]  
[124 16 101 110 26]]  
  
Initial Fitness: 2835.87
```



### State Akhir Kubus

Final cube state after climbing:

Layer 1:

```
[[ 26 121 13 30 102]
 [106 1 88 24 114]
 [ 43 117 97 71 8]
 [ 68 10 94 125 31]
 [ 69 101 29 47 55]]
```

Layer 2:

```
[[ 84 103 9 42 78]
 [ 93 51 116 41 3]
 [ 19 50 59 92 111]
 [ 85 58 70 49 14]
 [ 32 57 76 56 112]]
```

Layer 3:

```
[[ 91 27 100 62 35]
 [119 66 17 36 74]
 [ 65 6 99 81 79]
 [ 12 123 15 53 108]
 [ 40 87 67 64 21]]
```

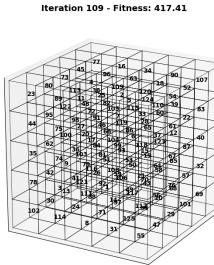
Layer 4:

```
[[ 38 11 122 95 44]
 [ 2 82 72 98 75]
 [124 115 7 46 20]
 [ 39 60 28 86 104]
 [ 83 22 61 37 118]]
```

Layer 5:

```
[[ 77 45 73 80 23]
 [ 16 96 4 113 89]
 [ 34 63 109 25 48]
 [ 90 18 120 5 105]
 [107 52 54 110 33]]
```

Final Fitness: 417.41



Iteration 109 - Fitness: 417.41

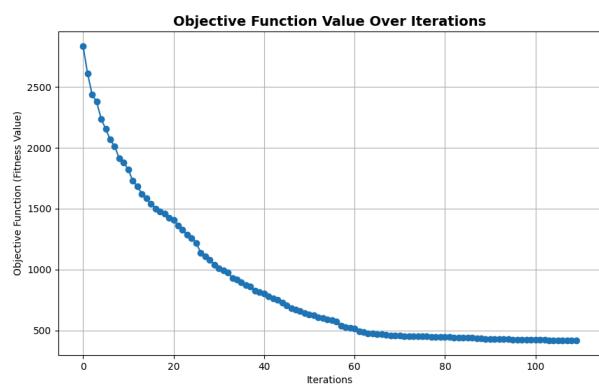
Durasi Pencarian

52.68 detik

Banyak Iterasi

109

Plot Nilai Objektif



## Bukti Eksperimen

```
Local Search Algorithms:  
1. Steepest Hill Climbing  
2. Hill Climbing with Sideways Move  
3. Random Restart Hill Climbing  
4. Stochastic Hill Climbing  
5. Simulated Annealing  
6. Genetic Algorithm  
7. Back to Main Menu  
Choose the algorithm you want to use: 1  
  
Do you want to keep the replay of the cube solving process? (y/n): y  
Enter the output file name (without extension): steepest_2  
The file will be saved as 'steepest_2.json' in the 'result/' directory.  
  
Input the maximum iteration (default 1000): 1000  
  
  
Final cube state is reached after 109 iterations in 52.68 seconds.  
  
Final cube state after climbing:  
Layer 1:  
[[ 26 121 13 30 102]  
[106 1 88 24 114]  
[ 43 117 97 71 8]  
[ 68 10 94 125 31]  
[ 69 101 29 47 55]]  
  
Layer 2:  
[[ 84 103 9 42 78]  
[ 93 51 116 41 3]  
[ 19 50 59 92 111]  
[ 85 58 70 49 14]  
[ 32 57 76 56 112]]  
  
Layer 3:  
[[ 91 27 100 62 35]  
[119 66 17 36 74]  
[ 65 6 99 81 79]  
[ 12 123 15 53 108]  
[ 40 87 67 64 21]]  
  
Layer 4:  
[[ 38 11 122 95 44]  
[ 2 82 72 98 75]  
[124 115 7 46 20]  
[ 39 60 28 86 104]  
[ 83 22 61 37 118]]  
  
Layer 5:  
[[ 77 45 73 80 23]  
[ 16 96 4 113 89]  
[ 34 63 109 25 48]  
[ 90 18 120 5 105]  
[107 52 54 110 33]]  
  
Final Fitness: 417.41
```

### 3. Eksperimen III

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[103 54 109 78 90]  
 [ 35 23 87 49 28]  
 [ 57 118 120 121 50]  
 [122 31 105 33 19]  
 [ 66 72 76 69 102]]  
  
Layer 2:  
[[ 21 1 68 29 7]  
 [ 67 83 13 81 111]  
 [ 42 38 95 115 47]  
 [ 80 12 114 59 10]  
 [ 20 46 6 52 112]]  
  
Layer 3:  
[[ 75 44 14 73 64]  
 [ 8 61 34 17 4]  
 [117 123 58 119 24]  
 [ 89 22 71 91 11]  
 [ 3 43 51 70 26]]  
  
Layer 4:  
[[ 36 56 9 124 5]  
 [ 18 15 79 53 88]  
 [ 60 96 32 86 48]  
 [ 2 55 82 92 116]  
 [ 27 113 94 45 85]]  
  
Layer 5:  
[[101 104 41 108 110]  
 [ 65 25 30 39 93]  
 [ 99 125 40 84 77]  
 [ 97 37 100 63 62]  
 [ 98 16 74 107 106]]  
  
Initial Fitness: 3910.25
```

	<p style="text-align: center;"><b>Iteration 0 - Fitness: 3910.25</b></p>
State Akhir Kubus	<p>Final cube state after climbing:</p> <p>Layer 1:</p> <pre>[[ 32 61 117 14 90]  [ 25 111 87 69 13]  [ 86 3 51 121 50]  [122 71 12 31 62]  [ 49 72 46 63 102]]</pre> <p>Layer 2:</p> <pre>[[ 21 103 120 35 7]  [104 47 1 81 105]  [ 42 5 95 116 66]  [ 80 38 114 54 23]  [ 83 84 6 11 112]]</pre> <p>Layer 3:</p> <pre>[[ 75 44 15 73 106]  [ 20 60 124 28 78]  [113 123 58 19 4]  [ 18 64 56 91 76]  [ 99 43 65 98 26]]</pre> <p>Layer 4:</p> <pre>[[118 77 9 101 17]  [ 52 10 79 74 88]  [ 67 96 82 27 48]  [ 2 70 53 92 115]  [ 41 93 94 45 36]]</pre> <p>Layer 5:</p> <pre>[[ 68 33 29 108 110]  [ 89 85 30 39 57]  [ 8 125 40 24 119]  [ 97 37 100 59 22]  [ 55 16 109 107 34]]</pre> <p style="text-align: center;">Final Fitness: 207.32</p>

	<p><b>Iteration 122 - Fitness: 207.32</b></p>
Durasi Pencarian	76.49 detik
Banyak Iterasi	122
Plot Nilai Objektif	<p><b>Objective Function Value Over Iterations</b></p>
Bukti Eksperimen	<pre> Initial Fitness: 3910.25 Do you want to see the cube's 3d representation? (y/n): y Local Search Algorithms: 1. Steepest Hill Climbing 2. Hill Climbing with Sideways Move 3. Random Restart Hill Climbing 4. Stochastic Hill Climbing 5. Simulated Annealing 6. Genetic Algorithm 7. Back to Main Menu Choose the algorithm you want to use: 1  Do you want to keep the replay of the cube solving process? (y/n): y Enter the output file name (without extension): steepest_var3_new The file will be saved as 'steepest_var3_new.json' in the 'result/' directory.  Input the maximum iteration (default 1000): 1000  Final cube state is reached after 122 iterations in 76.49 seconds.  Final cube state after climbing: Layer 1: [[ 32  61 117 14  90]  [ 25 111  87  69  13]  [ 86   3  51 121  50]  [122  71  12  31  62]  [ 49  72  46  63 102]]</pre>

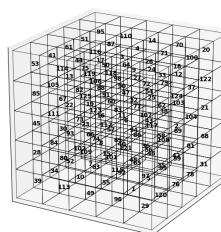
## B. Hill-climbing with Sideways Move

## 1. Eksperimen I

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[ 8 109 52 34 39]  
[121 101 63 10 113]  
[ 46 65 92 55 49]  
[ 59 99 5 1 96]  
[ 31 78 76 120 29]]  
  
Layer 2:  
[[ 16 23 93 84 28]  
[ 6 57 74 102 80]  
[117 86 40 15 7]  
[ 89 108 56 48 115]  
[ 68 81 42 35 97]]  
  
Layer 3:  
[[ 88 125 22 111 45]  
[ 47 60 123 73 30]  
[ 25 77 11 112 66]  
[103 38 43 94 69]  
[ 21 104 2 58 36]]  
  
Layer 4:  
[[ 17 75 13 105 85]  
[ 64 118 106 82 67]  
[ 24 27 90 91 19]  
[ 12 79 54 32 83]  
[122 37 124 62 107]]  
  
Layer 5:  
[[ 95 51 61 41 53]  
[110 87 116 44 114]  
[ 14 4 3 50 119]  
[ 70 71 26 9 98]  
[ 20 100 18 33 72]]  
  
Initial Fitness: 3886.38
```

Iteration 0 - Fitness: 3886.38



### State Akhir Kubus

Final cube state after climbing:  
 Layer 1:  
`[[ 86 31 118 53 7]  
 [121 82 11 10 105]  
 [ 46 65 92 51 64]  
 [ 12 119 5 84 96]  
 [ 44 26 89 117 36]]`

Layer 2:  
`[[ 35 23 112 122 28]  
 [ 6 45 74 80 94]  
 [123 76 52 15 42]  
 [ 78 108 56 48 37]  
 [ 68 81 16 41 114]]`

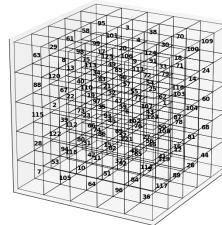
Layer 3:  
`[[ 83 110 22 2 115]  
 [ 57 32 97 73 39]  
 [ 25 77 59 93 66]  
 [103 1 43 102 69]  
 [ 60 104 87 50 27]]`

Layer 4:  
`[[ 17 75 13 120 88]  
 [106 49 34 40 67]  
 [ 91 111 90 21 19]  
 [ 71 79 54 55 47]  
 [ 24 14 116 62 107]]`

Layer 5:  
`[[ 95 58 61 29 63]  
 [ 3 101 99 98 8]  
 [ 38 4 20 125 113]  
 [ 70 30 124 9 85]  
 [109 100 18 33 72]]`

Final Fitness: 147.71

Iteration 115 - Fitness: 147.71



Durasi Pencarian

38.00 detik

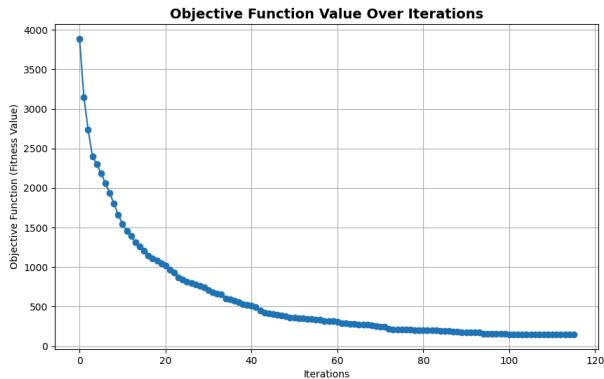
Banyak Iterasi

115

Maksimal  
Sideways Move

50

Plot Nilai Objektif



Bukti Eksperimen

1. Steepest Hill Climbing
  2. Hill Climbing with Sideways Move
  3. Random Restart Hill Climbing
  4. Stochastic Hill Climbing
  5. Simulated Annealing
  6. Genetic Algorithm
  7. Back to Main Menu
- Choose the algorithm you want to use: 2

```
Do you want to keep the replay of the cube solving process? (y/n): y
Enter the output file name (without extension): sideways_1
The file will be saved as 'sideways_1.json' in the 'result/' directory.
```

```
Input the maximum iteration (default 1000): 1000
Input the maximum sideways move (default 50): 50
```

```
Final cube state is reached after 115 iterations in 38.00 seconds.
```

```
Final cube state after climbing:
```

Layer 1:

```
[[ 86  31 118  53   7]
 [121  82  11  18 105]
 [ 46  65  92  51  64]
 [ 12 119    5  84  96]
 [ 44  26  89 117  36]]
```

Layer 2:

```
[[ 35  23 112 122  28]
 [  6  45  74  80  94]
 [123  76  52  15  42]
 [ 78 108  56  48  37]
 [ 68  81  16  41 114]]
```

Layer 3:

```
[[ 83 110  22    2 115]
 [ 57  32  97  73  39]
 [ 25  77  59  93  66]
 [103   1  43 102  69]
 [ 60 104  87  58  27]]
```

Layer 4:

```
[[ 17  75  13 120  88]
 [106  49  34  40  67]
 [ 91 111  90  21  19]
 [ 71  79  54  55  47]
 [ 24  14 116  62 107]]
```

Layer 5:

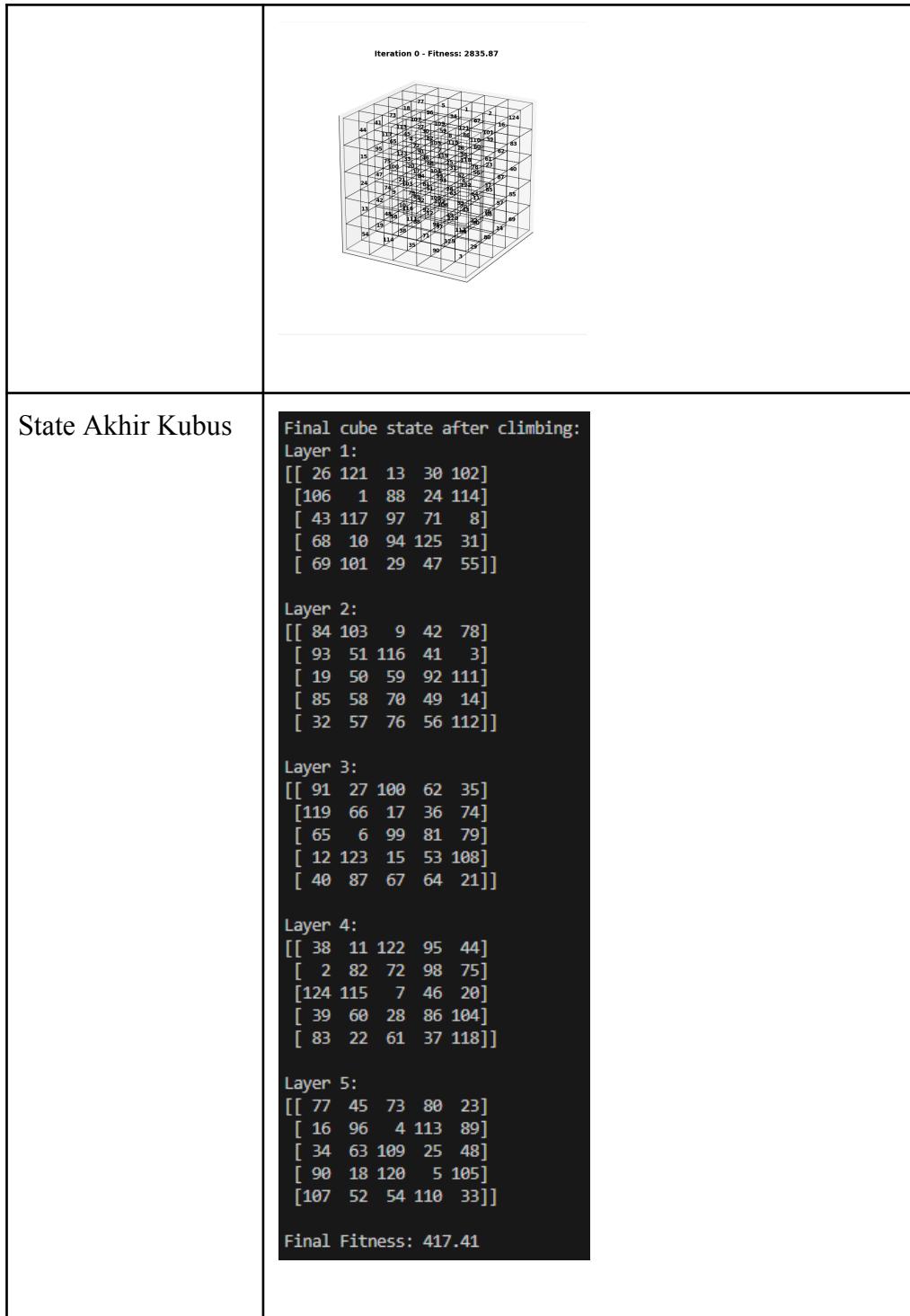
```
[[ 95  58  61  29  63]
 [  3 101  99  98   8]
 [ 38   4  20 125 113]
 [ 70  30 124    9  85]
 [109 100  18  33  72]]
```

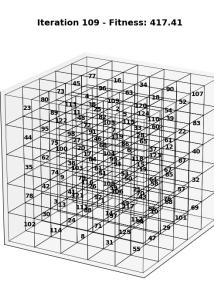
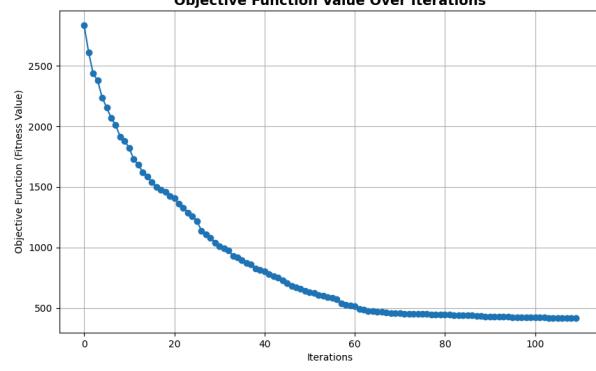
```
Final Fitness: 147.71
```

## 2. Eksperimen II

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[ 32 116 58 19 54]  
[106 12 88 38 114]  
[ 43 120 97 71 35]  
[ 68 50 94 125 90]  
[ 69 14 80 29  3]]  
  
Layer 2:  
[[ 84 103  9 42 13]  
[ 93 51 89 10 48]  
[122 63 102 92 111]  
[ 85 11 70 49 98]  
[ 55 57 76 22 112]]  
  
Layer 3:  
[[ 91 33 100 47 24]  
[119 66 17 21 74]  
[118 31 99 81 79]  
[ 23 56  6 28 108]  
[ 40 87 37 64 53]]  
  
Layer 4:  
[[ 27 45 65 95 15]  
[ 59 82 72 123 75]  
[ 86 115  7 46 20]  
[ 39 60 36 25 104]  
[ 83 62 61 78 52]]  
  
Layer 5:  
[[ 77 18 73 41 44]  
[  5 96 107 113 117]  
[  1 34 109 30  4]  
[  2 67 121  8 105]  
[124 16 101 110  26]]  
  
Initial Fitness: 2835.87
```



	
Durasi Pencarian	35.56 detik
Banyak Iterasi	109
Maksimal Sideways Move	50
Plot Nilai Objektif	<p style="text-align: center;"><b>Objective Function Value Over Iterations</b></p> 

## Bukti Eksperimen

```
1. Steepest Hill Climbing
2. Hill Climbing with Sideways Move
3. Random Restart Hill Climbing
4. Stochastic Hill Climbing
5. Simulated Annealing
6. Genetic Algorithm
7. Back to Main Menu
Choose the algorithm you want to use: 2

Do you want to keep the replay of the cube solving process? (y/n): y
Enter the output file name (without extension): sideways_2
The file will be saved as 'sideways_2.json' in the 'result/' directory.

Input the maximum iteration (default 1000): 1000
Input the maximum sideways move (default 50): 50

Final cube state is reached after 109 iterations in 35.56 seconds.

Final cube state after climbing:
Layer 1:
[[ 26 121 13 30 102]
 [106 1 88 24 114]
 [ 43 117 97 71 8]
 [ 68 10 94 125 31]
 [ 69 101 29 47 55]]

Layer 2:
[[ 84 103 9 42 78]
 [ 93 51 116 41 3]
 [ 19 50 59 92 111]
 [ 85 58 70 49 14]
 [ 32 57 76 56 112]]

Layer 3:
[[ 91 27 100 62 35]
 [119 66 17 36 74]
 [ 65 6 99 81 79]
 [ 12 123 15 53 108]
 [ 40 87 67 64 21]]

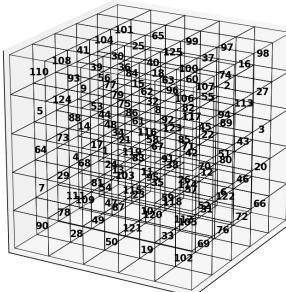
Layer 4:
[[ 38 11 122 95 44]
 [ 2 82 72 98 75]
 [124 115 7 46 20]
 [ 39 60 28 86 104]
 [ 83 22 61 37 118]]

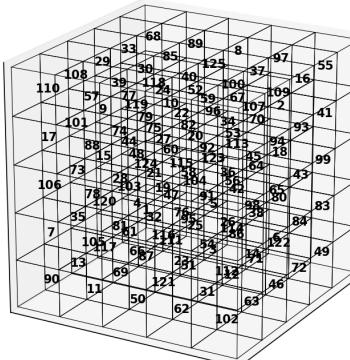
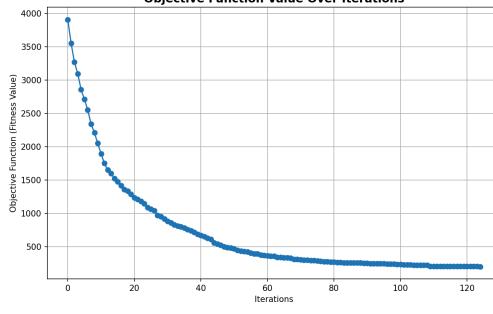
Layer 5:
[[ 77 45 73 80 23]
 [ 16 96 4 113 89]
 [ 34 63 109 25 48]
 [ 90 18 120 5 105]
 [107 52 54 110 33]]]

Final Fitness: 417.41
```

### 3. Eksperimen III

State Awal Kubus	<pre>Initial Cube State: Layer 1: [[103 54 109 78 90]  [ 35 23 87 49 28]  [ 57 118 120 121 50]  [122 31 105 33 19]  [ 66 72 76 69 102]]  Layer 2: [[ 21 1 68 29 7]  [ 67 83 13 81 111]  [ 42 38 95 115 47]  [ 80 12 114 59 10]  [ 20 46 6 52 112]]  Layer 3: [[ 75 44 14 73 64]  [ 8 61 34 17 4]  [117 123 58 119 24]  [ 89 22 71 91 11]  [ 3 43 51 70 26]]  Layer 4: [[ 36 56 9 124 5]  [ 18 15 79 53 88]  [ 60 96 32 86 48]  [ 2 55 82 92 116]  [ 27 113 94 45 85]]  Layer 5: [[101 104 41 108 110]  [ 65 25 30 39 93]  [ 99 125 40 84 77]  [ 97 37 100 63 62]  [ 98 16 74 107 106]]  Initial Fitness: 3910.25</pre>
------------------	--

	<p style="text-align: center;"><b>Iteration 0 - Fitness: 3910.25</b></p> 
State Akhir Kubus	<pre> Final cube state after climbing: Layer 1: [[ 32  61 117  13  90]  [ 25 111  87  69  11]  [ 86   3  51 121  50]  [122  71  12  31  62]  [ 49  72  46  63 102]]   Layer 2: [[ 21 103 120  35   7]  [104  47   1  81 105]  [ 42   5  95 116  66]  [ 80  38 114  54  23]  [ 83  84   6  14 112]]   Layer 3: [[ 75  44  15  73 106]  [ 20  60 124  28  78]  [113 123  58  19   4]  [ 18  64  56  91  76]  [ 99  43  65  98  26]]   Layer 4: [[118  77    9 101  17]  [ 52  10  79  74  88]  [ 67  96  82  27  48]  [  2  70  53  92 115]  [ 41  93  94  45  36]]   Layer 5: [[ 68  33  29 108 110]  [ 89  85  30  39  57]  [  8 125  40  24 119]  [ 97  37 100  59  22]  [ 55  16 109 107  34]]   Final Fitness: 206.77 </pre>

	<b>Iteration 124 - Fitness: 206.77</b> 
Durasi Pencarian	77.14 detik
Banyak Iterasi	124
Maksimal Sideways Move	50
Plot Nilai Objektif	
Bukti Eksperimen	<pre> Local Search Algorithms: 1. Steepest Hill Climbing 2. Hill Climbing with Sideways Move 3. Random Restart Hill Climbing 4. Stochastic Hill Climbing 5. Simulated Annealing 6. Genetic Algorithm 7. Back to Main Menu Choose the algorithm you want to use: 2  Do you want to keep the replay of the cube solving process? (y/n): y Enter the output file name (without extension): sideways_var3_new The file will be saved as 'sideways_var3_new.json' in the 'result/' directory.  Input the maximum iteration (default 1000): 1000 Input the maximum sideways move (default 50): 50  Final cube state is reached after 124 iterations in 77.14 seconds.  Final cube state after climbing: Layer 1: [[ 32  61 117 13  90]] </pre>

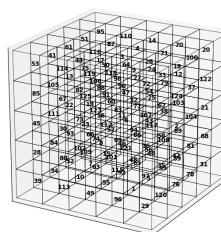
### C. Random Restart Hill-climbing

## 1. Eksperimen I

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[ 8 109 52 34 39]  
[121 101 63 10 113]  
[ 46 65 92 55 49]  
[ 59 99 5 1 96]  
[ 31 78 76 120 29]]  
  
Layer 2:  
[[ 16 23 93 84 28]  
[ 6 57 74 102 80]  
[117 86 40 15 7]  
[ 89 108 56 48 115]  
[ 68 81 42 35 97]]  
  
Layer 3:  
[[ 88 125 22 111 45]  
[ 47 60 123 73 30]  
[ 25 77 11 112 66]  
[103 38 43 94 69]  
[ 21 104 2 58 36]]  
  
Layer 4:  
[[ 17 75 13 105 85]  
[ 64 118 106 82 67]  
[ 24 27 90 91 19]  
[ 12 79 54 32 83]  
[122 37 124 62 107]]  
  
Layer 5:  
[[ 95 51 61 41 53]  
[110 87 116 44 114]  
[ 14 4 3 50 119]  
[ 70 71 26 9 98]  
[ 20 100 18 33 72]]  
  
Initial Fitness: 3886.38
```

Iteration 0 - Fitness: 3886.38



### State Akhir Kubus

Final cube state after climbing:

Layer 1:

```
[[ 86  31 118  53   7]
 [121  82  11  10 105]
 [ 46  65  92  51  64]
 [ 12 119   5  84  96]
 [ 44  26  89 117  36]]
```

Layer 2:

```
[[ 35  23 112 122  28]
 [  6  45  74  80  94]
 [123  76  52  15  42]
 [ 78 108  56  48  37]
 [ 68  81  16  41 114]]
```

Layer 3:

```
[[ 83 110  22   2 115]
 [ 57  32  97  73  39]
 [ 25  77  59  93  66]
 [103   1  43 102  69]
 [ 60 104  87  50  27]]
```

Layer 4:

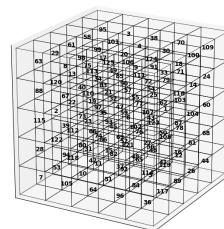
```
[[ 17  75  13 120  88]
 [106  49  34  40  67]
 [ 91 111  90  21  19]
 [ 71  79  54  55  47]
 [ 24  14 116  62 107]]
```

Layer 5:

```
[[ 95  58  61  29  63]
 [  3 101  99  98   8]
 [ 38   4  20 125 113]
 [ 70  30 124   9  85]
 [109 100  18  33  72]]
```

Final Fitness: 147.71

Iteration 2191 - Fitness: 147.71



Durasi Pencarian

1267.81 detik

Banyak Iterasi Per Restart	<pre> Restart: 0 - Jumlah Iterasi: 115 Restart: 1 - Jumlah Iterasi: 130 Restart: 2 - Jumlah Iterasi: 140 Restart: 3 - Jumlah Iterasi: 109 Restart: 4 - Jumlah Iterasi: 124 Restart: 5 - Jumlah Iterasi: 93 Restart: 6 - Jumlah Iterasi: 90 Restart: 7 - Jumlah Iterasi: 121 Restart: 8 - Jumlah Iterasi: 87 Restart: 9 - Jumlah Iterasi: 135 Restart: 10 - Jumlah Iterasi: 75 Restart: 11 - Jumlah Iterasi: 123 Restart: 12 - Jumlah Iterasi: 131 Restart: 13 - Jumlah Iterasi: 115 Restart: 14 - Jumlah Iterasi: 76 Restart: 15 - Jumlah Iterasi: 92 Restart: 16 - Jumlah Iterasi: 85 Restart: 17 - Jumlah Iterasi: 97 Restart: 18 - Jumlah Iterasi: 130 Restart: 19 - Jumlah Iterasi: 104 </pre>
Plot Nilai Objektif	<p>The plot illustrates the optimization process. The objective function value starts high (around 3500) and rapidly decreases in a periodic, oscillatory manner, indicating the search space being explored by the algorithm. The x-axis represents the number of iterations, and the y-axis represents the objective function value or fitness.</p>

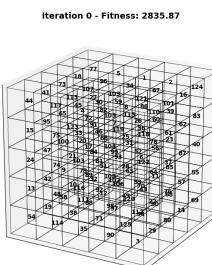
## Bukti Eksperimen

```
Local Search Algorithms:  
1. Steepest Hill Climbing  
2. Hill Climbing with Sideways Move  
3. Random Restart Hill Climbing  
4. Stochastic Hill Climbing  
5. Simulated Annealing  
6. Genetic Algorithm  
7. Back to Main Menu  
Choose the algorithm you want to use: 3  
  
Do you want to keep the replay of the cube solving process? (y/n): y  
Enter the output file name (without extension): random_restart_1  
The file will be saved as 'random_restart_1.json' in the 'result/' directory.  
  
Input the maximum restart (default 20): 20  
Input the maximum iteration (default 1000): 1000  
  
Restart: 0 - Jumlah Iterasi: 115  
Restart: 1 - Jumlah Iterasi: 130  
Restart: 2 - Jumlah Iterasi: 140  
Restart: 3 - Jumlah Iterasi: 109  
Restart: 4 - Jumlah Iterasi: 124  
Restart: 5 - Jumlah Iterasi: 93  
Restart: 6 - Jumlah Iterasi: 90  
Restart: 7 - Jumlah Iterasi: 121  
Restart: 8 - Jumlah Iterasi: 87  
Restart: 9 - Jumlah Iterasi: 135  
Restart: 10 - Jumlah Iterasi: 75  
Restart: 11 - Jumlah Iterasi: 123  
Restart: 12 - Jumlah Iterasi: 131  
Restart: 13 - Jumlah Iterasi: 115  
Restart: 14 - Jumlah Iterasi: 76  
Restart: 15 - Jumlah Iterasi: 92  
Restart: 16 - Jumlah Iterasi: 85  
Restart: 17 - Jumlah Iterasi: 97  
Restart: 18 - Jumlah Iterasi: 130  
Restart: 19 - Jumlah Iterasi: 104  
  
Final cube state is reached after 2191 iterations in 1267.81 seconds.  
  
Final cube state after climbing:  
Layer 1:  
[[ 86 31 118 53 7]  
 [121 82 11 10 105]  
 [ 46 65 92 51 64]  
 [ 12 119 5 84 96]  
 [ 44 26 89 117 36]]  
  
Layer 2:  
[[ 35 23 112 122 28]  
 [ 6 45 74 80 94]  
 [123 76 52 15 42]  
 [ 78 108 56 48 37]  
 [ 68 81 16 41 114]]  
  
Layer 3:  
[[ 83 110 22 2 115]  
 [ 57 32 97 73 39]  
 [ 25 77 59 93 66]  
 [103 1 43 102 69]  
 [ 60 104 87 50 27]]  
  
Layer 4:  
[[ 17 75 13 120 88]  
 [106 49 34 40 67]  
 [ 91 111 90 21 19]  
 [ 71 79 54 55 47]  
 [ 24 14 116 62 107]]  
  
Layer 5:  
[[ 95 58 61 29 63]  
 [ 3 101 99 98 8]  
 [ 38 4 20 125 113]  
 [ 70 30 124 9 85]  
 [109 100 18 33 72]]  
  
Final Fitness: 147.71
```

## 2. Eksperimen II

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[ 32 116 58 19 54]  
[106 12 88 38 114]  
[ 43 120 97 71 35]  
[ 68 50 94 125 90]  
[ 69 14 80 29 3]]  
  
Layer 2:  
[[ 84 103 9 42 13]  
[ 93 51 89 10 48]  
[122 63 102 92 111]  
[ 85 11 70 49 98]  
[ 55 57 76 22 112]]  
  
Layer 3:  
[[ 91 33 100 47 24]  
[119 66 17 21 74]  
[118 31 99 81 79]  
[ 23 56 6 28 108]  
[ 40 87 37 64 53]]  
  
Layer 4:  
[[ 27 45 65 95 15]  
[ 59 82 72 123 75]  
[ 86 115 7 46 20]  
[ 39 60 36 25 104]  
[ 83 62 61 78 52]]  
  
Layer 5:  
[[ 77 18 73 41 44]  
[ 5 96 107 113 117]  
[ 1 34 109 30 4]  
[ 2 67 121 8 105]  
[124 16 101 110 26]]  
  
Initial Fitness: 2835.87
```



### State Akhir Kubus

```

Final cube state after climbing:
Layer 1:
[[ 81  73  55  45  59]
 [ 26  22  27 118 114]
 [ 16  90  76  37  96]
 [109  12  65 103  25]
 [ 68 116  95  10  17]]

Layer 2:
[[ 23  18 124  41  98]
 [106  97  43  46  29]
 [ 30 115  91  36  44]
 [ 61   1  48  79 110]
 [104  84  7 121  21]]

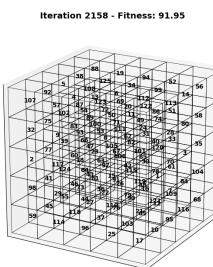
Layer 3:
[[100  93  39  77   2]
 [111   8  47  60 117]
 [ 31  82  67  63  64]
 [ 33 120  85  42  15]
 [ 35   3  70  78 119]]

Layer 4:
[[ 13  87 102  75  32]
 [ 69  72  89  83   9]
 [122  11  71  53  66]
 [ 51  74  24  62 105]
 [ 58  80  28  40 101]]

Layer 5:
[[ 88  38    5  92 107]
 [ 19 125 108    4  57]
 [ 94  34    6 123  54]
 [ 52  99 112  20  50]
 [ 56  14 113  86  49]]

```

Final Fitness: 91.95



Durasi Pencarian	1240.10
------------------	---------

Banyak Iterasi Per Restart	<pre> Restart: 0 - Jumlah Iterasi: 109 Restart: 1 - Jumlah Iterasi: 79 Restart: 2 - Jumlah Iterasi: 111 Restart: 3 - Jumlah Iterasi: 113 Restart: 4 - Jumlah Iterasi: 89 Restart: 5 - Jumlah Iterasi: 118 Restart: 6 - Jumlah Iterasi: 90 Restart: 7 - Jumlah Iterasi: 76 Restart: 8 - Jumlah Iterasi: 85 Restart: 9 - Jumlah Iterasi: 118 Restart: 10 - Jumlah Iterasi: 162 Restart: 11 - Jumlah Iterasi: 94 Restart: 12 - Jumlah Iterasi: 105 Restart: 13 - Jumlah Iterasi: 112 Restart: 14 - Jumlah Iterasi: 141 Restart: 15 - Jumlah Iterasi: 111 Restart: 16 - Jumlah Iterasi: 148 Restart: 17 - Jumlah Iterasi: 72 Restart: 18 - Jumlah Iterasi: 106 Restart: 19 - Jumlah Iterasi: 100 </pre>
Plot Nilai Objektif	<p>The plot illustrates the performance of an optimization algorithm. The vertical axis represents the Objective Function (Fitness Value), ranging from 0 to 3500. The horizontal axis represents the Iterations, ranging from 0 to 2000. The data points show a clear downward trend over time, indicating progress towards a minimum. However, the algorithm exhibits significant local minima, often dropping to values near zero before rapidly increasing again, which suggests it might be stuck in local optima or experiencing high-frequency oscillations.</p>

## Bukti Eksperimen

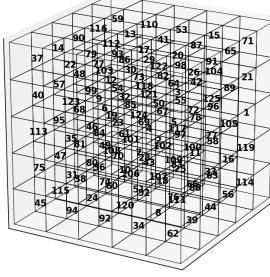
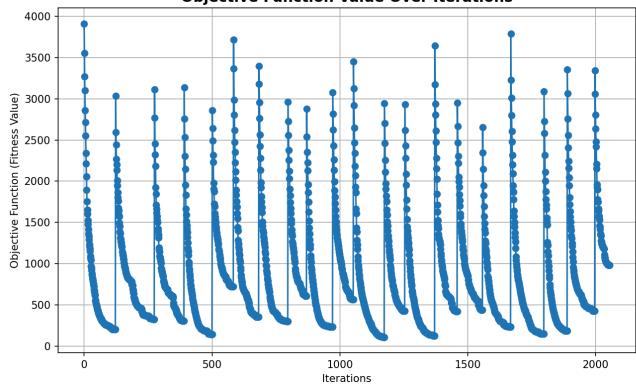
```
Local Search Algorithms:  
1. Steepest Hill Climbing  
2. Hill Climbing with Sideways Move  
3. Random Restart Hill Climbing  
4. Stochastic Hill Climbing  
5. Simulated Annealing  
6. Genetic Algorithm  
7. Back to Main Menu  
Choose the algorithm you want to use: 3  
  
Do you want to keep the replay of the cube solving process? (y/n): y  
Enter the output file name (without extension): random_restart_2  
The file will be saved as 'random_restart_2.json' in the 'result/' directory.  
  
Input the maximum restart (default 20): 20  
Input the maximum iteration (default 1000): 1000  
  
Restart: 0 - Jumlah Iterasi: 109  
Restart: 1 - Jumlah Iterasi: 79  
Restart: 2 - Jumlah Iterasi: 111  
Restart: 3 - Jumlah Iterasi: 113  
Restart: 4 - Jumlah Iterasi: 89  
Restart: 5 - Jumlah Iterasi: 118  
Restart: 6 - Jumlah Iterasi: 90  
Restart: 7 - Jumlah Iterasi: 76  
Restart: 8 - Jumlah Iterasi: 85  
Restart: 9 - Jumlah Iterasi: 118  
Restart: 10 - Jumlah Iterasi: 162  
Restart: 11 - Jumlah Iterasi: 94  
Restart: 12 - Jumlah Iterasi: 105  
Restart: 13 - Jumlah Iterasi: 112  
Restart: 14 - Jumlah Iterasi: 141  
Restart: 15 - Jumlah Iterasi: 111  
Restart: 16 - Jumlah Iterasi: 148  
Restart: 17 - Jumlah Iterasi: 72  
Restart: 18 - Jumlah Iterasi: 106  
Restart: 19 - Jumlah Iterasi: 100  
  
Final cube state is reached after 2158 iterations in 1240.10 seconds.  
  
Final cube state after climbing:  
Layer 1:  
[[ 81  73  55  45  59]  
 [ 26  22  27 118 114]  
 [ 16  90  76  37  96]  
 [109  12  65 103  25]  
 [ 68 116  95  10  17]]  
  
Layer 2:  
[[ 23  18 124  41  98]  
 [106  97  43  46  29]  
 [ 30 115  91  36  44]  
 [ 61   1  48  79 110]  
 [104  84    7 121  21]]  
  
Layer 3:  
[[100  93  39  77   2]  
 [111   8  47  60 117]  
 [ 31  82  67  63  64]  
 [ 33 120  85  42  15]  
 [ 35   3  70  78 119]]  
  
Layer 4:  
[[ 13  87 102  75  32]  
 [ 69  72  89  83   9]  
 [122  11  71  53  66]  
 [ 51  74  24  62 105]  
 [ 58  80  28  40 101]]  
  
Layer 5:  
[[ 88  38    5  92 107]  
 [ 19 125 108    4  57]  
 [ 94  34    6 123  54]  
 [ 52  99 112  20  50]  
 [ 56  14 113  86  49]]  
  
Final Fitness: 91.95
```

### 3. Eksperimen III

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[103 54 109 78 90]  
 [ 35 23 87 49 28]  
 [ 57 118 120 121 50]  
 [122 31 105 33 19]  
 [ 66 72 76 69 102]]  
  
Layer 2:  
[[ 21 1 68 29 7]  
 [ 67 83 13 81 111]  
 [ 42 38 95 115 47]  
 [ 80 12 114 59 10]  
 [ 20 46 6 52 112]]  
  
Layer 3:  
[[ 75 44 14 73 64]  
 [ 8 61 34 17 4]  
 [117 123 58 119 24]  
 [ 89 22 71 91 11]  
 [ 3 43 51 70 26]]  
  
Layer 4:  
[[ 36 56 9 124 5]  
 [ 18 15 79 53 88]  
 [ 60 96 32 86 48]  
 [ 2 55 82 92 116]  
 [ 27 113 94 45 85]]  
  
Layer 5:  
[[101 104 41 108 110]  
 [ 65 25 30 39 93]  
 [ 99 125 40 84 77]  
 [ 97 37 100 63 62]  
 [ 98 16 74 107 106]]  
  
Initial Fitness: 3910.25
```

	<p style="text-align: center;"><b>Iteration 0 - Fitness: 3910.25</b></p>
State Akhir Kubus	<pre> Final cube state after climbing: Layer 1: [[ 70  36  38 115  45]  [ 43 106  60  24  94]  [ 25  18  52 120  92]  [ 63  88 111   8  34]  [114  56  44  39  62]]   Layer 2: [[ 23  84  81  47  75]  [  4 101 108  80  31]  [ 97 102  28  10  74]  [ 58  11  83 107  51]  [119  16  33  66  76]]   Layer 3: [[ 54    9  68  95 113]  [121  85  19  46  35]  [ 55  67  69  61  49]  [ 96  78 117   2  7]  [  1 105  27 100 109]]   Layer 4: [[ 93  77  48  57  40]  [ 29  30  12  99 123]  [ 98  82 118  32   6]  [104  42   3  50 124]  [ 21  89 125  72   5]]   Layer 5: [[ 59 116  90  14  37]  [110  13 112  79  22]  [ 53  41  17  86 103]  [ 15  87  20 122  73]  [ 71  65  91  26  64]]   Final Fitness: 114.67 </pre>

	<p style="text-align: center;"><b>Iteration 2054 - Fitness: 114.67</b></p> 
Durasi Pencarian	1827.07 detik
Banyak Iterasi Per Restart	2054
Plot Nilai Objektif	<p style="text-align: center;"><b>Objective Function Value Over Iterations</b></p> 
Bukti Eksperimen	<pre> Do you want to keep the replay of the cube solving process? (y/n): y Enter the output file name (without extension): rr_var3 Local Search Algorithms: 1. Steepest Hill Climbing 2. Hill Climbing with Sideways Move 3. Random Restart Hill Climbing 4. Stochastic Hill Climbing 5. Simulated Annealing 6. Genetic Algorithm 7. Back to Main Menu Choose the algorithm you want to use: 3 </pre>

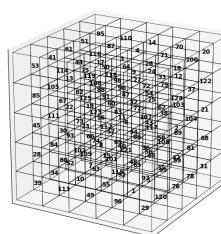
## D. Stochastic Hill-climbing

## 1. Eksperimen I

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[ 8 109 52 34 39]  
[121 101 63 10 113]  
[ 46 65 92 55 49]  
[ 59 99 5 1 96]  
[ 31 78 76 120 29]]  
  
Layer 2:  
[[ 16 23 93 84 28]  
[ 6 57 74 102 80]  
[117 86 40 15 7]  
[ 89 108 56 48 115]  
[ 68 81 42 35 97]]  
  
Layer 3:  
[[ 88 125 22 111 45]  
[ 47 60 123 73 30]  
[ 25 77 11 112 66]  
[103 38 43 94 69]  
[ 21 104 2 58 36]]  
  
Layer 4:  
[[ 17 75 13 105 85]  
[ 64 118 106 82 67]  
[ 24 27 90 91 19]  
[ 12 79 54 32 83]  
[122 37 124 62 107]]  
  
Layer 5:  
[[ 95 51 61 41 53]  
[110 87 116 44 114]  
[ 14 4 3 50 119]  
[ 70 71 26 9 98]  
[ 20 100 18 33 72]]  
  
Initial Fitness: 3886.38
```

Iteration 0 - Fitness: 3886.38



### State Akhir Kubus

```

Final cube state after climbing:
Layer 1:
[[ 73  84 100  41  27]
 [111  44  31  99  13]
 [ 8  91  37  50 117]
 [ 65  46 110  15  47]
 [ 55  29  18 120 113]]

Layer 2:
[[ 11  32 106  94  36]
 [ 48  67  25  98  88]
 [123  68  89   7  26]
 [ 93  82  17  81  30]
 [ 9  42 116  35 102]]

Layer 3:
[[101  38  45  43  78]
 [ 6 105  54 114  23]
 [ 83  88  61  12  66]
 [ 87  10  40  95 118]
 [ 57 103  85  58  21]]

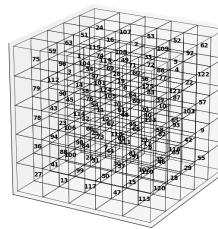
Layer 4:
[[ 74 104    3 112  79]
 [ 49  69  97  14  90]
 [ 72  60  19 124  76]
 [ 4  77 125  64  70]
 [122  22 121  39  20]]

Layer 5:
[[ 24  51  63  59  75]
 [107  16 115    1  96]
 [ 53   2 108 119  34]
 [ 52 109  33  71  28]
 [ 62  92    5  86  56]]

Final Fitness: 371.17

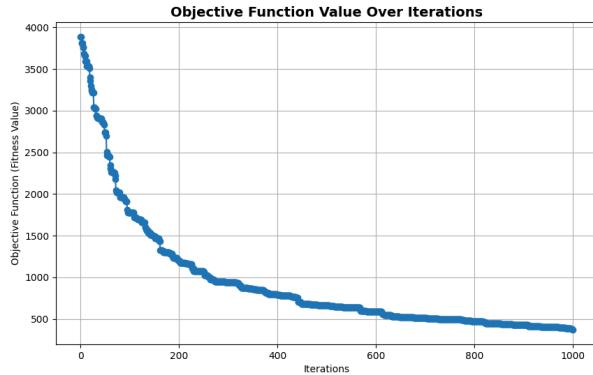
```

Iteration 1000 - Fitness: 371.17



Durasi Pencarian	90.01
Banyak Iterasi	1000

Plot Nilai Objektif



Bukti Eksperimen

```
Local Search Algorithms:  
1. Steepest Hill Climbing  
2. Hill Climbing with Sideways Move  
3. Random Restart Hill Climbing  
4. Stochastic Hill Climbing  
5. Simulated Annealing  
6. Genetic Algorithm  
7. Back to Main Menu  
Choose the algorithm you want to use: 4
```

```
Do you want to keep the replay of the cube solving process? (y/n): y  
Enter the output file name (without extension): stochastic_1  
The file will be saved as 'stochastic_1.json' in the 'result/' directory.
```

```
Input the maximum iteration (default 1000): 1000
```

```
Final cube state is reached after 1000 iterations in 90.01 seconds.
```

```
Final cube state after climbing:
```

```
Layer 1:
```

```
[[ 73 84 100 41 27]  
[111 44 31 99 13]  
[ 8 91 37 50 117]  
[ 65 46 110 15 47]  
[ 55 29 18 120 113]]
```

```
Layer 2:
```

```
[[ 11 32 106 94 36]  
[ 48 67 25 98 88]  
[123 68 89 7 26]  
[ 93 82 17 81 30]  
[ 9 42 116 35 102]]
```

```
Layer 3:
```

```
[[101 38 45 43 78]  
[ 6 105 54 114 23]  
[ 83 80 61 12 66]  
[ 87 10 40 95 118]  
[ 57 103 85 58 21]]
```

```
Layer 4:
```

```
[[ 74 104 3 112 79]  
[ 49 69 97 14 90]  
[ 72 60 19 124 76]  
[ 4 77 125 64 70]  
[122 22 121 39 20]]
```

```
Layer 5:
```

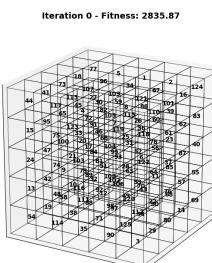
```
[[ 24 51 63 59 75]  
[107 16 115 1 96]  
[ 53 2 108 119 34]  
[ 52 109 33 71 28]  
[ 62 92 5 86 56]]
```

```
Final Fitness: 371.17
```

## 2. Eksperimen II

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[ 32 116 58 19 54]  
[106 12 88 38 114]  
[ 43 120 97 71 35]  
[ 68 50 94 125 90]  
[ 69 14 80 29 3]]  
  
Layer 2:  
[[ 84 103 9 42 13]  
[ 93 51 89 10 48]  
[122 63 102 92 111]  
[ 85 11 70 49 98]  
[ 55 57 76 22 112]]  
  
Layer 3:  
[[ 91 33 100 47 24]  
[119 66 17 21 74]  
[118 31 99 81 79]  
[ 23 56 6 28 108]  
[ 40 87 37 64 53]]  
  
Layer 4:  
[[ 27 45 65 95 15]  
[ 59 82 72 123 75]  
[ 86 115 7 46 20]  
[ 39 60 36 25 104]  
[ 83 62 61 78 52]]  
  
Layer 5:  
[[ 77 18 73 41 44]  
[ 5 96 107 113 117]  
[ 1 34 109 30 4]  
[ 2 67 121 8 105]  
[124 16 101 110 26]]  
  
Initial Fitness: 2835.87
```



### State Akhir Kubus

```

Final cube state after climbing:
Layer 1:
[[ 41 125 26 29 86]
 [ 32 3 102 84 77]
 [120 52 89 19 30]
 [ 57 60 12 119 73]
 [ 69 38 98 70 35]]

Layer 2:
[[ 90 27 59 40 75]
 [ 97 49 113 74 2]
 [ 1 83 20 114 111]
 [ 33 58 79 48 47]
 [ 93 81 14 21 123]]

Layer 3:
[[ 99 50 106 53 15]
 [117 10 44 22 92]
 [ 46 31 71 43 124]
 [ 17 110 28 116 72]
 [ 62 109 61 68 18]]

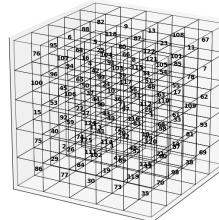
Layer 4:
[[ 25 16 94 96 100]
 [ 66 91 42 65 45]
 [121 115 39 24 37]
 [ 85 54 51 56 36]
 [ 7 78 55 63 112]]

Layer 5:
[[ 82 88 6 95 76]
 [ 9 118 5 64 107]
 [ 13 87 80 104 4]
 [108 23 122 8 103]
 [ 67 11 101 105 34]]

Final Fitness: 315.64

```

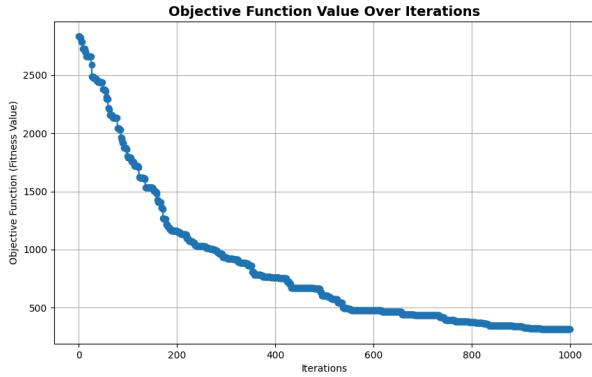
Iteration 1000 - Fitness: 315.64



Durasi Pencarian 95.47

Banyak Iterasi 1000

Plot Nilai Objektif



Bukti Eksperimen

```
Local Search Algorithms:  
1. Steepest Hill Climbing  
2. Hill Climbing with Sideways Move  
3. Random Restart Hill Climbing  
4. Stochastic Hill Climbing  
5. Simulated Annealing  
6. Genetic Algorithm  
7. Back to Main Menu  
Choose the algorithm you want to use: 4
```

```
Do you want to keep the replay of the cube solving process? (y/n): y  
Enter the output file name (without extension): stochastic_2  
The file will be saved as 'stochastic_2.json' in the 'result/' directory.
```

```
Input the maximum iteration (default 1000): 1000
```

```
Final cube state is reached after 1000 iterations in 95.47 seconds.
```

```
Final cube state after climbing:
```

```
Layer 1:
```

```
[[ 41 125 26 29 86]  
[ 32 3 102 84 77]  
[120 52 89 19 30]  
[ 57 60 12 119 73]  
[ 69 38 98 70 35]]
```

```
Layer 2:
```

```
[[ 90 27 59 40 75]  
[ 97 49 113 74 2]  
[ 1 83 20 114 111]  
[ 33 58 79 48 47]  
[ 93 81 14 21 123]]
```

```
Layer 3:
```

```
[[ 99 50 106 53 15]  
[117 10 44 22 92]  
[ 46 31 71 43 124]  
[ 17 110 28 116 72]  
[ 62 109 61 68 18]]
```

```
Layer 4:
```

```
[[ 25 16 94 96 100]  
[ 66 91 42 65 45]  
[121 115 39 24 37]  
[ 85 54 51 56 36]  
[ 7 78 55 63 112]]
```

```
Layer 5:
```

```
[[ 82 88 6 95 76]  
[ 9 118 5 64 107]  
[ 13 87 80 104 4]  
[108 23 122 8 103]  
[ 67 11 101 105 34]]
```

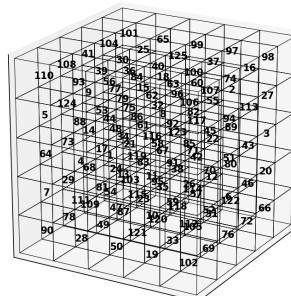
```
Final Fitness: 315.64
```

### 3. Eksperimen III

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[103 54 109 78 90]  
[ 35 23 87 49 28]  
[ 57 118 120 121 50]  
[122 31 105 33 19]  
[ 66 72 76 69 102]]  
  
Layer 2:  
[[ 21 1 68 29 7]  
[ 67 83 13 81 111]  
[ 42 38 95 115 47]  
[ 80 12 114 59 10]  
[ 20 46 6 52 112]]  
  
Layer 3:  
[[ 75 44 14 73 64]  
[ 8 61 34 17 4]  
[117 123 58 119 24]  
[ 89 22 71 91 11]  
[ 3 43 51 70 26]]  
  
Layer 4:  
[[ 36 56 9 124 5]  
[ 18 15 79 53 88]  
[ 60 96 32 86 48]  
[ 2 55 82 92 116]  
[ 27 113 94 45 85]]  
  
Layer 5:  
[[101 104 41 108 110]  
[ 65 25 30 39 93]  
[ 99 125 40 84 77]  
[ 97 37 100 63 62]  
[ 98 16 74 107 106]]  
  
Initial Fitness: 3910.25
```

Iteration 0 - Fitness: 3910.25



State Akhir Kubus

Final cube state after climbing:

Layer 1:  
[[ 72 54 78 42 83]  
[ 18 23 105 64 124]  
[ 71 118 52 9 47]  
[111 13 40 120 19]  
[ 55 113 8 80 63]]

Layer 2:  
[[ 44 43 121 77 21]  
[ 67 82 34 30 90]  
[ 56 69 50 115 29]  
[ 39 62 99 37 87]  
[114 49 2 68 102]]

Layer 3:  
[[ 1 123 3 66 125]  
[ 97 122 61 41 7]  
[ 76 32 59 91 31]  
[ 65 14 58 103 93]  
[112 45 110 22 38]]

Layer 4:  
[[107 36 53 117 10]  
[ 75 4 79 108 35]  
[ 89 84 48 12 95]  
[ 6 116 33 57 94]  
[ 27 85 119 16 98]]

Layer 5:  
[[109 73 24 11 74]  
[ 92 86 25 60 51]  
[ 28 5 70 106 100]  
[ 81 104 96 26 15]  
[ 17 20 101 88 46]]

Final Fitness: 236.57

	<b>Iteration 1000 - Fitness: 236.57</b> 
Durasi Pencarian	179.12 detik
Banyak Iterasi	1000
Plot Nilai Objektif	
Bukti Eksperimen	<p>Local Search Algorithms:</p> <ol style="list-style-type: none"> <li>1. Steepest Hill Climbing</li> <li>2. Hill Climbing with Sideways Move</li> <li>3. Random Restart Hill Climbing</li> <li>4. Stochastic Hill Climbing</li> <li>5. Simulated Annealing</li> <li>6. Genetic Algorithm</li> <li>7. Back to Main Menu</li> </ol> <p>Choose the algorithm you want to use: 4</p> <p>Do you want to keep the replay of the cube solving process? (y/n): n</p> <p>Input the maximum iteration (default 1000): 1000</p>

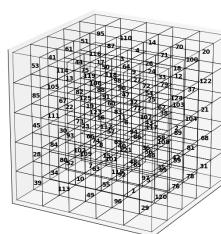
## E. Simulated Annealing

## 1. Eksperimen I

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[ 8 109 52 34 39]  
[121 101 63 10 113]  
[ 46 65 92 55 49]  
[ 59 99 5 1 96]  
[ 31 78 76 120 29]]  
  
Layer 2:  
[[ 16 23 93 84 28]  
[ 6 57 74 102 80]  
[117 86 40 15 7]  
[ 89 108 56 48 115]  
[ 68 81 42 35 97]]  
  
Layer 3:  
[[ 88 125 22 111 45]  
[ 47 60 123 73 30]  
[ 25 77 11 112 66]  
[103 38 43 94 69]  
[ 21 104 2 58 36]]  
  
Layer 4:  
[[ 17 75 13 105 85]  
[ 64 118 106 82 67]  
[ 24 27 90 91 19]  
[ 12 79 54 32 83]  
[122 37 124 62 107]]  
  
Layer 5:  
[[ 95 51 61 41 53]  
[110 87 116 44 114]  
[ 14 4 3 50 119]  
[ 70 71 26 9 98]  
[ 20 100 18 33 72]]  
  
Initial Fitness: 3886.38
```

Iteration 0 - Fitness: 3886.38



### State Akhir Kubus

Final cube state after climbing:

Layer 1:

```
[[ 61  59 100  68  32]
 [ 55 117  34  58  37]
 [  9  57  54  97  99]
 [116  49 121   1  53]
 [105  24  21  98  85]]
```

Layer 2:

```
[[ 33  29 109 106  45]
 [  6  84  43  83 113]
 [125  62  36  51  10]
 [ 47  46 115  63  28]
 [ 92  91   4  20 123]]
```

Layer 3:

```
[[ 89  96  38   5  94]
 [103  22  71 119   3]
 [ 78  93  50   7  73]
 [  2  11  77 112 114]
 [ 41 101  65  74  18]]
```

Layer 4:

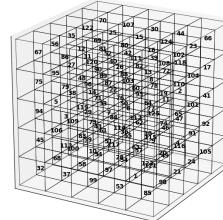
```
[[ 81  31  27  95  75]
 [ 42  40  90  48  79]
 [ 39  82  88  87  14]
 [118  72   8  60  76]
 [ 17 104 110  19  64]]
```

Layer 5:

```
[[ 70 122  35  56  67]
 [107  25  69  12  86]
 [ 30  15  80  52 120]
 [ 44 124  16 111  26]
 [ 66  23 102 108  13]]
```

Final Fitness: 149.03

Iteration 2000 - Fitness: 149.03



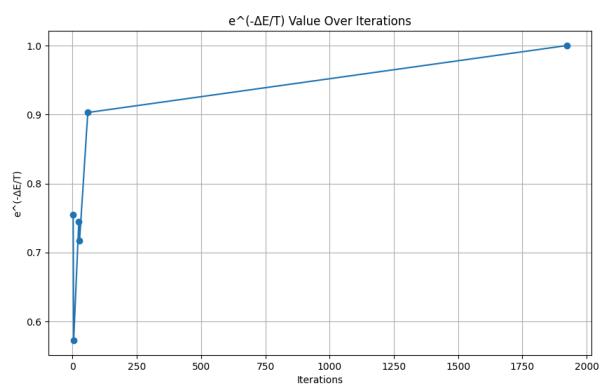
Durasi Pencarian

386.15

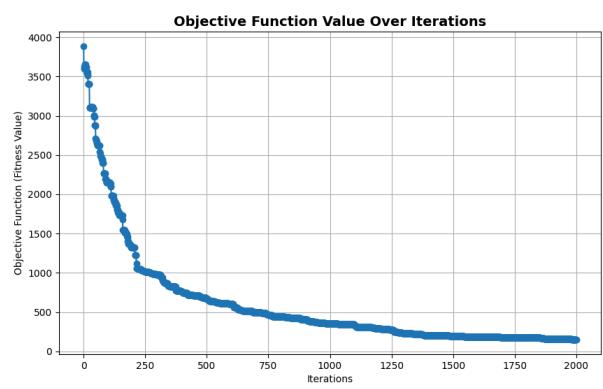
Banyak Iterasi

2000

Plot Probabilitas



Plot Nilai Objektif



## Bukti Eksperimen

```
Do you want to keep the replay of the cube solving process? (y/n): y
Enter the output file name (without extension): sa_1
The file will be saved as 'sa_1.json' in the 'result/' directory.

Input the threshold (default 0.5): 0.5

Loading...
Choose cooling schedule you want to use from options below!
1. Default (Quadratic Multiplicative Monotonic)
2. Linear Multiplicative Monotonic
3. Logarithmic Multiplicative
4. Exponential Multiplicative Monotonic
Input your choice (1-4) : 1
Worse moves accepted : 6

Final cube state is reached after 2000 iterations in 386.15 seconds.

Final cube state after climbing:
Layer 1:
[[ 61  59 100  68  32]
 [ 55 117  34  58  37]
 [  9  57  54  97  99]
 [116  49 121   1  53]
 [105  24  21  98  85]]

Layer 2:
[[ 33  29 109 106  45]
 [  6  84  43  83 113]
 [125  62  36  51  10]
 [ 47  46 115  63  28]
 [ 92  91   4  20 123]]

Layer 3:
[[ 89  96  38   5  94]
 [103  22  71 119   3]
 [ 78  93  50   7  73]
 [  2  11  77 112 114]
 [ 41 101  65  74  18]]

Layer 4:
[[ 81  31  27  95  75]
 [ 42  40  90  48  79]
 [ 39  82  88  87  14]
 [118  72   8  60  76]
 [ 17 104 110  19  64]]

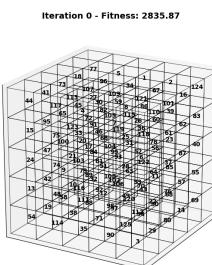
Layer 5:
[[ 70 122  35  56  67]
 [107  25  69  12  86]
 [ 30  15  80  52 120]
 [ 44 124  16 111  26]
 [ 66  23 102 108  13]]
```

Final Fitness: 149.03

## 2. Eksperimen II

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[ 32 116 58 19 54]  
[106 12 88 38 114]  
[ 43 120 97 71 35]  
[ 68 50 94 125 90]  
[ 69 14 80 29 3]]  
  
Layer 2:  
[[ 84 103 9 42 13]  
[ 93 51 89 10 48]  
[122 63 102 92 111]  
[ 85 11 70 49 98]  
[ 55 57 76 22 112]]  
  
Layer 3:  
[[ 91 33 100 47 24]  
[119 66 17 21 74]  
[118 31 99 81 79]  
[ 23 56 6 28 108]  
[ 40 87 37 64 53]]  
  
Layer 4:  
[[ 27 45 65 95 15]  
[ 59 82 72 123 75]  
[ 86 115 7 46 20]  
[ 39 60 36 25 104]  
[ 83 62 61 78 52]]  
  
Layer 5:  
[[ 77 18 73 41 44]  
[ 5 96 107 113 117]  
[ 1 34 109 30 4]  
[ 2 67 121 8 105]  
[124 16 101 110 26]]  
  
Initial Fitness: 2835.87
```



### State Akhir Kubus

```

Final cube state after climbing:
Layer 1:
[[ 63  97  47   9  99]
 [ 79    5  89  78  81]
 [ 56 120  46  66  18]
 [ 19    2 117 121  65]
 [ 88 105  11  54  52]]

Layer 2:
[[ 76 183    7  33  98]
 [107  49 125  39   8]
 [ 80  12  43 118  57]
 [ 24  92 104  32  72]
 [ 26  74  35  45 113]]

Layer 3:
[[ 51  77  41  68 106]
 [ 4 123  40  31  71]
 [ 90  16  67  82  73]
 [112  62  61  27  42]
 [ 48  28 116 101  37]]

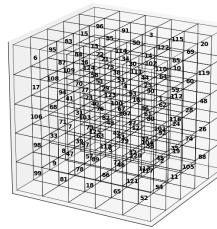
Layer 4:
[[ 22  36 109 108  17]
 [ 34  75  58  70  94]
 [102 111  53  29  23]
 [ 10  64  25  93 100]
 [119  60  59   1  86]]

Layer 5:
[[ 96  15  83  95   6]
 [ 91  55  13  84  87]
 [  3  50 114  21 124]
 [115 122  14  30  38]
 [ 20  69  85 110  44]]

Final Fitness: 146.49

```

Iteration 2000 - Fitness: 146.49



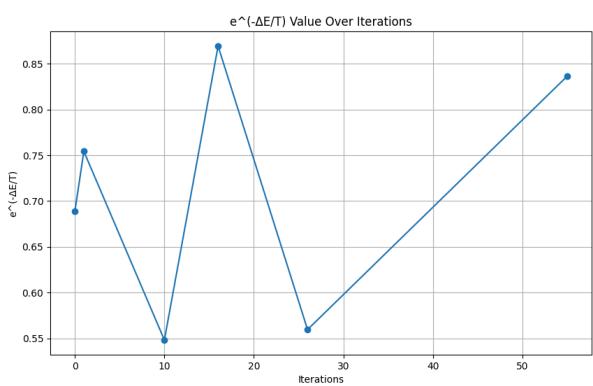
Durasi Pencarian

368.49

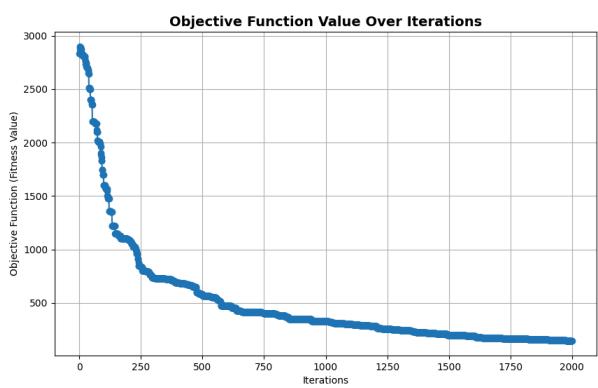
Banyak Iterasi

2000

Plot Probabilitas



Plot Nilai Objektif



## Bukti Eksperimen

```
Do you want to keep the replay of the cube solving process? (y/n): y
Enter the output file name (without extension): sa_2
The file will be saved as 'sa_2.json' in the 'result/' directory.
```

```
Input the threshold (default 0.5): 0.5
```

```
Loading...
```

```
Choose cooling schedule you want to use from options below!
```

1. Default (Quadratic Multiplicative Monotonic)
2. Linear Multiplicative Monotonic
3. Logarithmic Multiplicative
4. Exponential Multiplicative Monotonic

```
Input your choice (1-4) : 1
```

```
Worse moves accepted : 6
```

```
Final cube state is reached after 2000 iterations in 368.49 seconds.
```

```
Final cube state after climbing:
```

```
Layer 1:
```

```
[[ 63 97 47 9 99]
 [ 79 5 89 78 81]
 [ 56 120 46 66 18]
 [ 19 2 117 121 65]
 [ 88 105 11 54 52]]
```

```
Layer 2:
```

```
[[ 76 103 7 33 98]
 [107 49 125 39 8]
 [ 80 12 43 118 57]
 [ 24 92 104 32 72]
 [ 26 74 35 45 113]]
```

```
Layer 3:
```

```
[[ 51 77 41 68 106]
 [ 4 123 40 31 71]
 [ 90 16 67 82 73]
 [112 62 61 27 42]
 [ 48 28 116 101 37]]
```

```
Layer 4:
```

```
[[ 22 36 109 108 17]
 [ 34 75 58 70 94]
 [102 111 53 29 23]
 [ 10 64 25 93 100]
 [119 60 59 1 86]]
```

```
Layer 5:
```

```
[[ 96 15 83 95 6]
 [ 91 55 13 84 87]
 [ 3 50 114 21 124]
 [115 122 14 30 38]
 [ 20 69 85 110 44]]
```

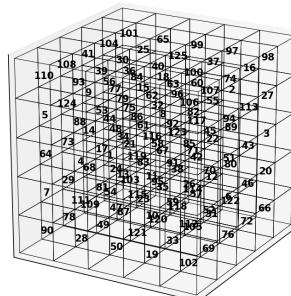
```
Final Fitness: 146.49
```

### 3. Eksperimen III

State Awal Kubus

```
Initial Cube State:  
Layer 1:  
[[103 54 109 78 90]  
[ 35 23 87 49 28]  
[ 57 118 120 121 50]  
[122 31 105 33 19]  
[ 66 72 76 69 102]]  
  
Layer 2:  
[[ 21 1 68 29 7]  
[ 67 83 13 81 111]  
[ 42 38 95 115 47]  
[ 80 12 114 59 10]  
[ 20 46 6 52 112]]  
  
Layer 3:  
[[ 75 44 14 73 64]  
[ 8 61 34 17 4]  
[117 123 58 119 24]  
[ 89 22 71 91 11]  
[ 3 43 51 70 26]]  
  
Layer 4:  
[[ 36 56 9 124 5]  
[ 18 15 79 53 88]  
[ 60 96 32 86 48]  
[ 2 55 82 92 116]  
[ 27 113 94 45 85]]  
  
Layer 5:  
[[101 104 41 108 110]  
[ 65 25 30 39 93]  
[ 99 125 40 84 77]  
[ 97 37 100 63 62]  
[ 98 16 74 107 106]]  
  
Initial Fitness: 3910.25
```

Iteration 0 - Fitness: 3910.25



### State Akhir Kubus

Final cube state after climbing:

Layer 1:  
[[ 49 28 115 67 32]  
[ 14 111 104 17 60]  
[ 76 20 83 103 30]  
[122 123 8 1 107]  
[ 35 55 3 118 85]]

Layer 2:  
[[ 41 50 86 92 43]  
[ 77 70 9 81 73]  
[ 18 11 90 93 120]  
[105 48 119 53 5]  
[ 66 109 12 2 79]]

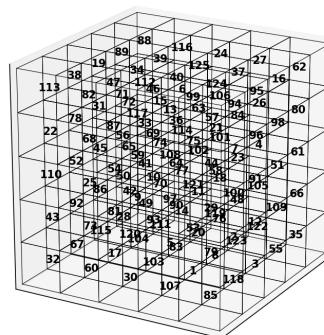
Layer 3:  
[[ 33 56 45 52 110]  
[114 74 59 54 25]  
[101 102 64 10 42]  
[ 4 23 58 121 97]  
[ 61 51 91 100 29]]

Layer 4:  
[[112 71 31 78 22]  
[ 6 15 117 87 68]  
[106 63 36 69 65]  
[ 26 84 21 75 108]  
[ 80 98 96 7 44]]

Layer 5:  
[[ 88 89 19 38 113]  
[116 39 34 47 82]  
[ 24 125 40 46 72]  
[ 27 37 124 99 13]  
[ 62 16 95 94 57]]

Final Fitness: 137.67

Iteration 2000 - Fitness: 137.67

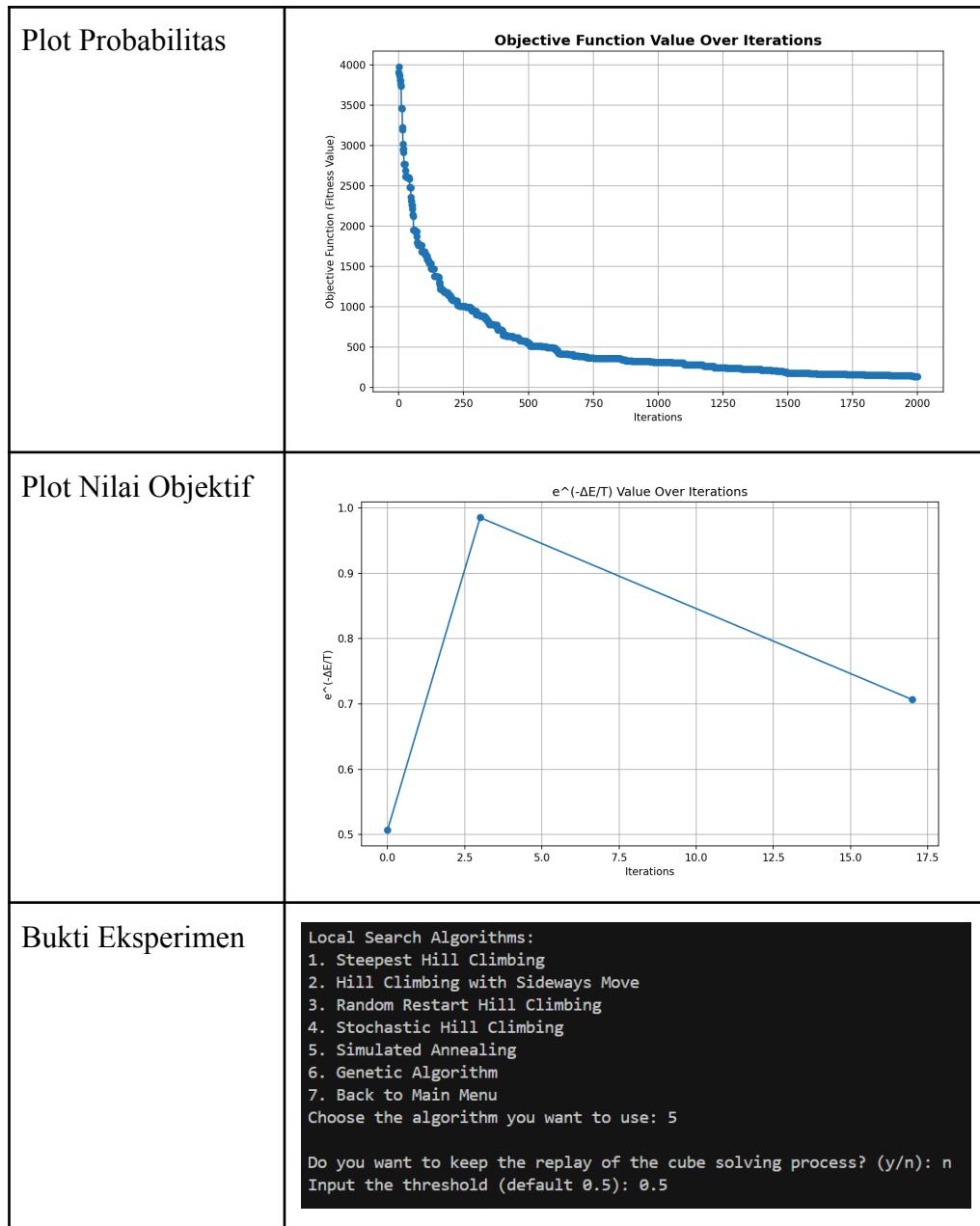


Durasi Pencarian

660.14 detik

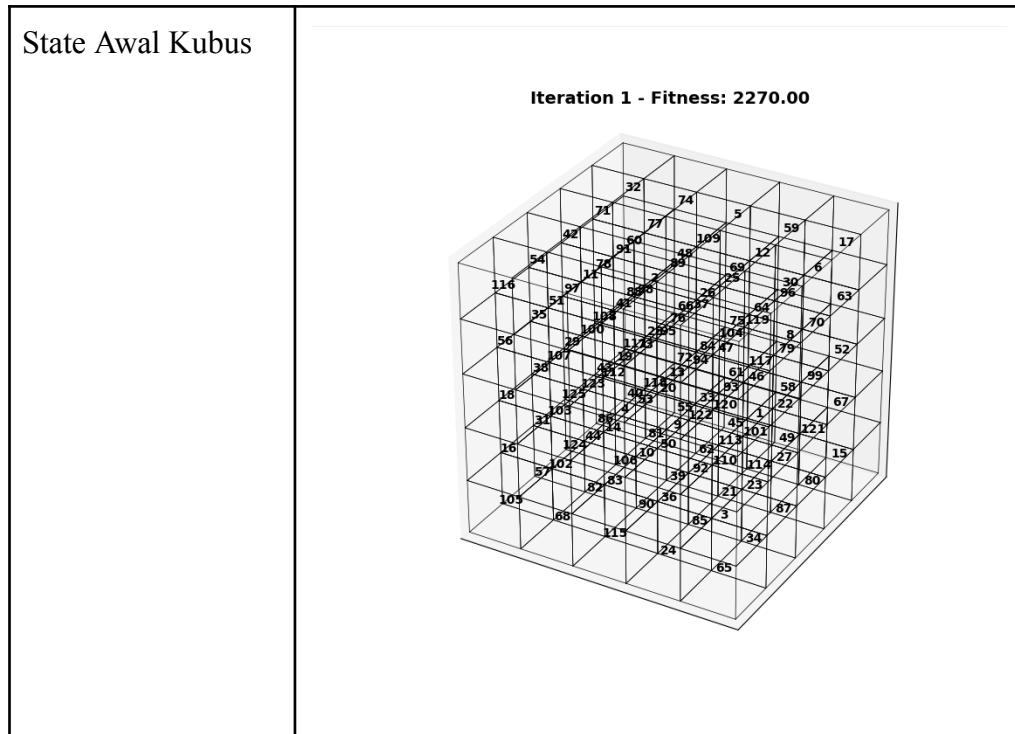
Banyak Iterasi

2000



## F. Genetic Algorithm

## 1. Eksperimen I



State Akhir Kubus

Final cube state after climbing:

Layer 1:

```
[[ 73 100 34 91 75]
 [ 57 82 90 10 7]
 [ 56 106 39 9 49]
 [ 38 6 98 61 101]
 [ 71 68 45 77 87]]
```

Layer 2:

```
[[ 71 87 45 40 15]
 [ 31 44 54 120 23]
 [125 4 42 104 37]
 [ 43 118 12 1 121]
 [ 22 17 123 28 119]]
```

Layer 3:

```
[[105 15 79 27 87]
 [119 23 46 99 69]
 [ 87 37 79 30 59]
 [ 22 121 70 58 67]
 [ 27 119 63 76 32]]
```

Layer 4:

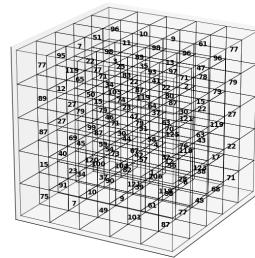
```
[[ 3 77 65 12 89]
 [35 83 34 50 27]
 [97 41 87 74 79]
 [78 2 80 64 47]
 [79 79 15 30 63]]
```

Layer 5:

```
[[ 96 51 7 95 77]
 [ 10 11 98 22 119]
 [ 9 98 89 25 71]
 [ 61 96 13 93 22]
 [ 77 96 47 71 22]]
```

Final Fitness: 1028.62

Iteration 999 - Fitness: 1028.62



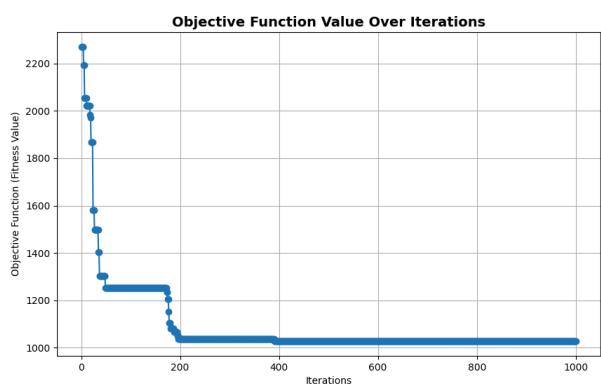
Durasi Pencarian

88.38 detik

Banyak Iterasi

1000

Plot Nilai Objektif



## Bukti Eksperimen

- 2. Hill Climbing with Sideways Move
- 3. Random Restart Hill Climbing
- 4. Stochastic Hill Climbing
- 5. Simulated Annealing
- 6. Genetic Algorithm
- 7. Back to Main Menu

Choose the algorithm you want to use: 6

Do you want to keep the replay of the cube solving process? (y/n): y  
Enter the output file name (without extension): ga\_1  
The file will be saved as 'ga\_1.json' in the 'result/' directory.

Input the population size: 5  
Input the maximum iteration: 1000  
Input the mutation rate (Between 0 and 1): 0.1

Final cube state is reached after 999 iterations in 88.38 seconds.

Final cube state after climbing:

Layer 1:

```
[[ 73 100 34 91 75]
 [ 57 82 90 10  7]
 [ 56 106 39  9 49]
 [ 38   6 98  61 101]
 [ 71 68 45 77 87]]
```

Layer 2:

```
[[ 71 87 45 40 15]
 [ 31 44 54 120 23]
 [125   4 42 104 37]
 [ 43 118 12   1 121]
 [ 22 17 123 28 119]]
```

Layer 3:

```
[[105 15 79 27 87]
 [119 23 46 99 69]
 [ 87 37 79 30 59]
 [ 22 121 70 58 67]
 [ 27 119 63 76 32]]
```

Layer 4:

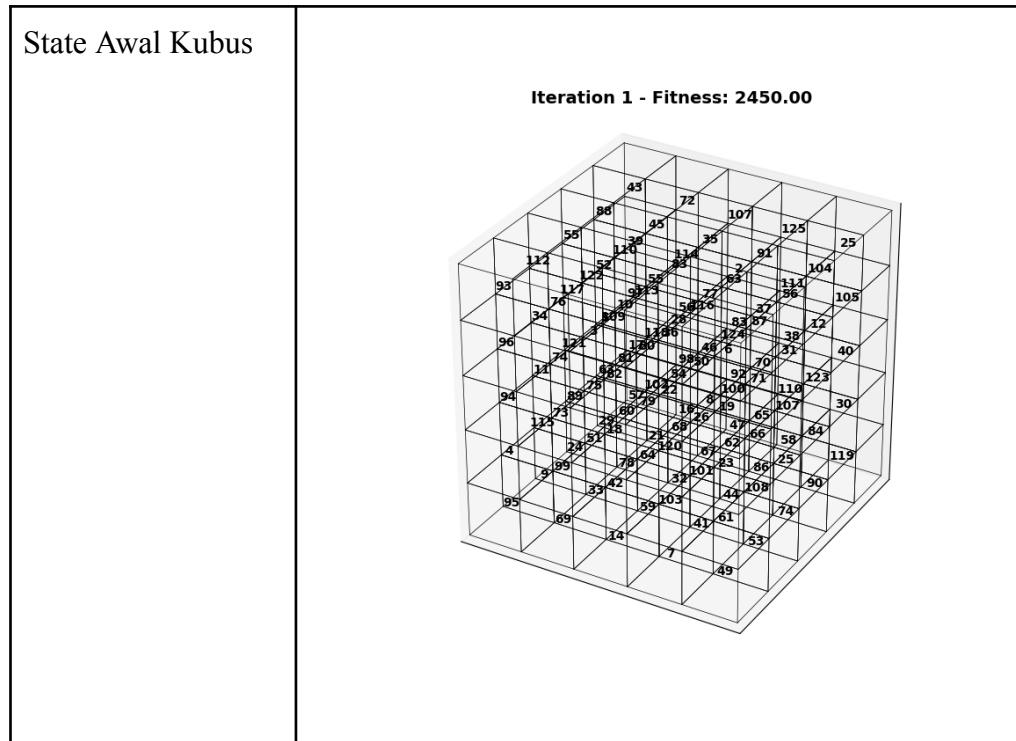
```
[[ 3 77 65 12 89]
 [35 83 34 50 27]
 [97 41 87 74 79]
 [78 2 80 64 47]
 [79 79 15 30 63]]
```

Layer 5:

```
[[ 96 51   7 95 77]
 [ 10 11 98 22 119]
 [  9 98 89 25 71]
 [ 61 96 13 93 22]
 [ 77 96 47 71 22]]
```

Final Fitness: 1028.62

## 2. Eksperimen II



State Akhir Kubus

Final cube state after climbing:

Layer 1:

```
[[ 95  54  14   7 103]
 [101    9  41 119   30]
 [ 24  78  32  44 122]
 [ 29  21  67  86  65]
 [ 73  76 109  36 107]]
```

Layer 2:

```
[[  4 108  42  57  30]
 [ 72  33  81  72  99]
 [ 89  60  68  75  25]
 [116 102    8 118  84]
 [ 17 100  92  76  30]]
```

Layer 3:

```
[[ 94  66  18 120  40]
 [ 6 113  79  26  93]
 [ 76  55  43  63  25]
 [ 5  88  46  70 123]
 [ 93  43 119  58   6]]
```

Layer 4:

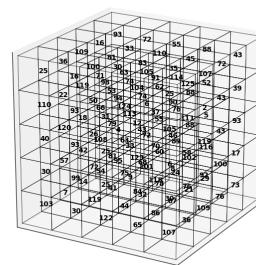
```
[[ 30  71 119  22 110]
 [105  74  53  50  93]
 [114  62  74 124  31]
 [ 52  88  90  37  12]
 [ 39  43  2 111 105]]
```

Layer 5:

```
[[ 93  16 109  36  25]
 [ 72  33  81 100  16]
 [ 55 110  83  63  98]
 [ 88  45  35  91 104]
 [ 43  72 107 125  25]]
```

Final Fitness: 1294.76

Iteration 999 - Fitness: 1294.76



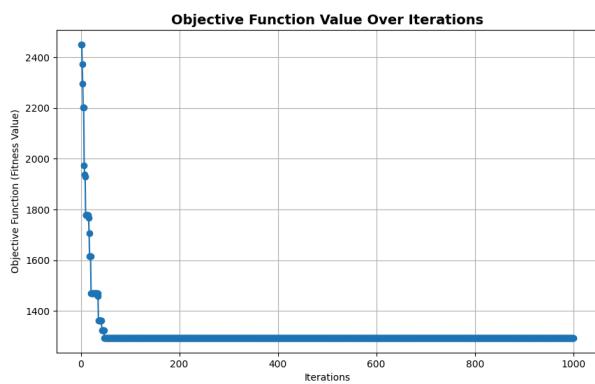
Durasi Pencarian

87.21 detik

Banyak Iterasi

1000

Plot Nilai Objektif



## Bukti Eksperimen

```
2. Hill Climbing with Sideways Move
3. Random Restart Hill Climbing
4. Stochastic Hill Climbing
5. Simulated Annealing
6. Genetic Algorithm
7. Back to Main Menu
```

```
Choose the algorithm you want to use: 6
```

```
Do you want to keep the replay of the cube solving process? (y/n): y
Enter the output file name (without extension): ga_2
The file will be saved as 'ga_2.json' in the 'result/' directory.
```

```
Input the population size: 10
Input the maximum iteration: 1000
Input the mutation rate (Between 0 and 1): 0.1
```

```
Final cube state is reached after 999 iterations in 87.21 seconds.
```

```
Final cube state after climbing:
```

```
Layer 1:
```

```
[[ 95  54  14   7 103]
 [101    9  41 119  30]
 [ 24  78  32  44 122]
 [ 29  21  67  86  65]
 [ 73  76 109  36 107]]
```

```
Layer 2:
```

```
[[  4 108  42  57  30]
 [ 72  33  81  72  99]
 [ 89  60  68  75  25]
 [116 102    8 118  84]
 [ 17 100  92  76  30]]
```

```
Layer 3:
```

```
[[ 94  66  18 120  40]
 [ 6 113  79  26  93]
 [ 76  55  43  63  25]
 [ 5  88  46  70 123]
 [ 93  43 119  58   6]]
```

```
Layer 4:
```

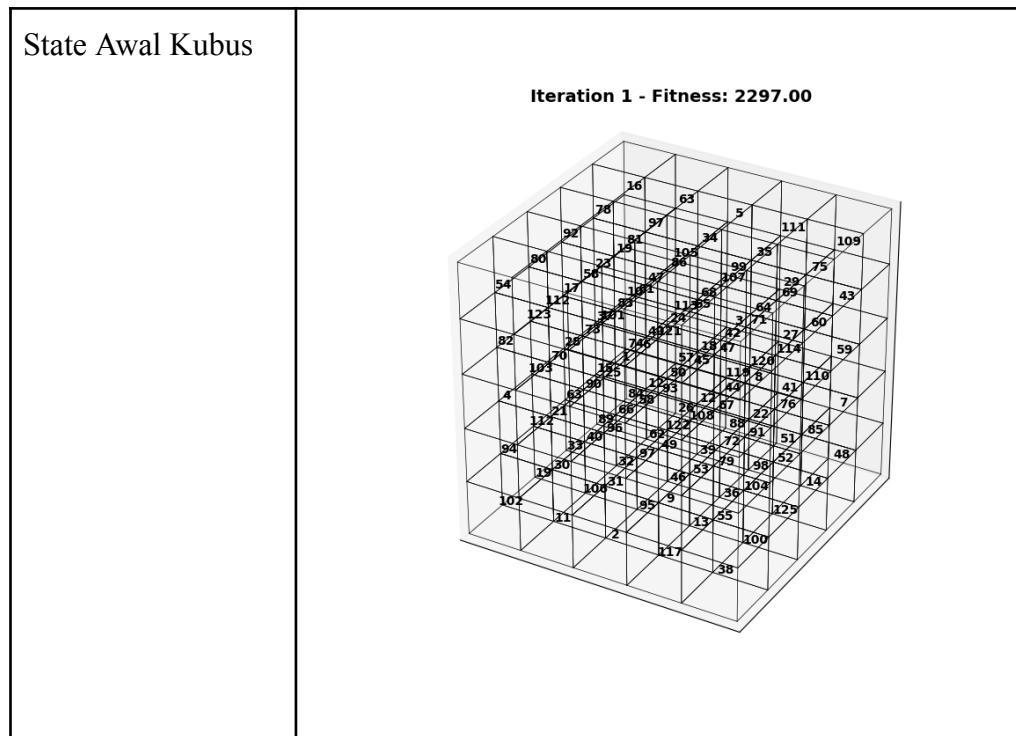
```
[[ 30  71 119  22 110]
 [105  74  53  50  93]
 [114  62  74 124  31]
 [ 52  88  90  37  12]
 [ 39  43    2 111 105]]
```

```
Layer 5:
```

```
[[ 93  16 109  36  25]
 [ 72  33  81 100  16]
 [ 55 110  83  63  98]
 [ 88  45  35  91 104]
 [ 43  72 107 125  25]]
```

```
Final Fitness: 1294.76
```

### 3. Eksperimen III



State Akhir Kubus

Final cube state after climbing:

Layer 1:

```
[[ 19  27  63  38 120]
 [ 78 120    8  91    3]
 [103  36  17  14  69]
 [ 39  59  37  90  45]
 [ 58  78 125  48  58]]
```

Layer 2:

```
[[ 84  88  16  12  30]
 [ 32  32 110  54  68]
 [ 47  68  62 109  16]
 [ 14  41  49  60  99]
 [ 99  45  49  62  99]]
```

Layer 3:

```
[[ 25  56  79  87  80]
 [110  65  93  55    6]
 [ 13  96  28 110    3]
 [ 33  37  51  53  69]
 [ 95  27  19  28 122]]
```

Layer 4:

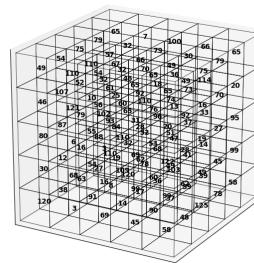
```
[[ 67  54  52 107  46]
 [ 70  46  61  10 121]
 [ 36  18  92  60 102]
 [114  73  74  76  31]
 [ 20  70  16  92  20]]
```

Layer 5:

```
[[ 65  79  75  54  49]
 [  7  32  57 110 110]
 [100  79  66  32  32]
 [ 66  30  49  65  65]
 [ 65  79  75  49  65]]
```

Final Fitness: 666.53

Iteration 999 - Fitness: 666.53



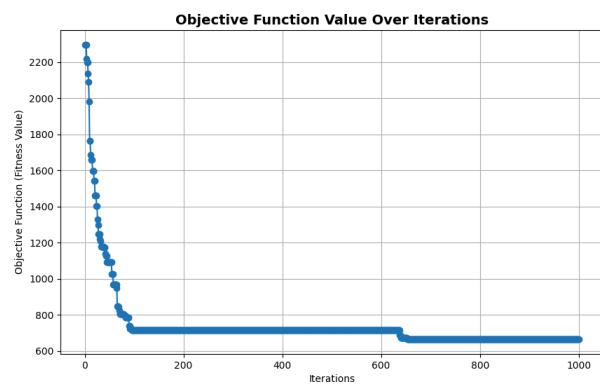
Durasi Pencarian

89.48 detik

Banyak Iterasi

1000

Plot Nilai Objektif



## Bukti Eksperimen

- 2. Hill Climbing with Sideways Move
- 3. Random Restart Hill Climbing
- 4. Stochastic Hill Climbing
- 5. Simulated Annealing
- 6. Genetic Algorithm
- 7. Back to Main Menu

Choose the algorithm you want to use: 6

Do you want to keep the replay of the cube solving process? (y/n): y  
Enter the output file name (without extension): ga\_3  
The file will be saved as 'ga\_3.json' in the 'result/' directory.

Input the population size: 15  
Input the maximum iteration: 1000  
Input the mutation rate (Between 0 and 1): 0.1

Final cube state is reached after 999 iterations in 89.48 seconds.

Final cube state after climbing:

Layer 1:

```
[[ 19  27  63  38 120]
 [ 78 120   8  91   3]
 [103  36  17  14  69]
 [ 39  59  37  90  45]
 [ 58  78 125  48  58]]
```

Layer 2:

```
[[ 84  88  16  12  30]
 [ 32  32 110  54  68]
 [ 47  68  62 109  16]
 [ 14  41  49  60  99]
 [ 99  45  49  62  99]]
```

Layer 3:

```
[[ 25  56  79  87  80]
 [110  65  93  55   6]
 [ 13  96  28 110   3]
 [ 33  37  51  53  69]
 [ 95  27  19  28 122]]
```

Layer 4:

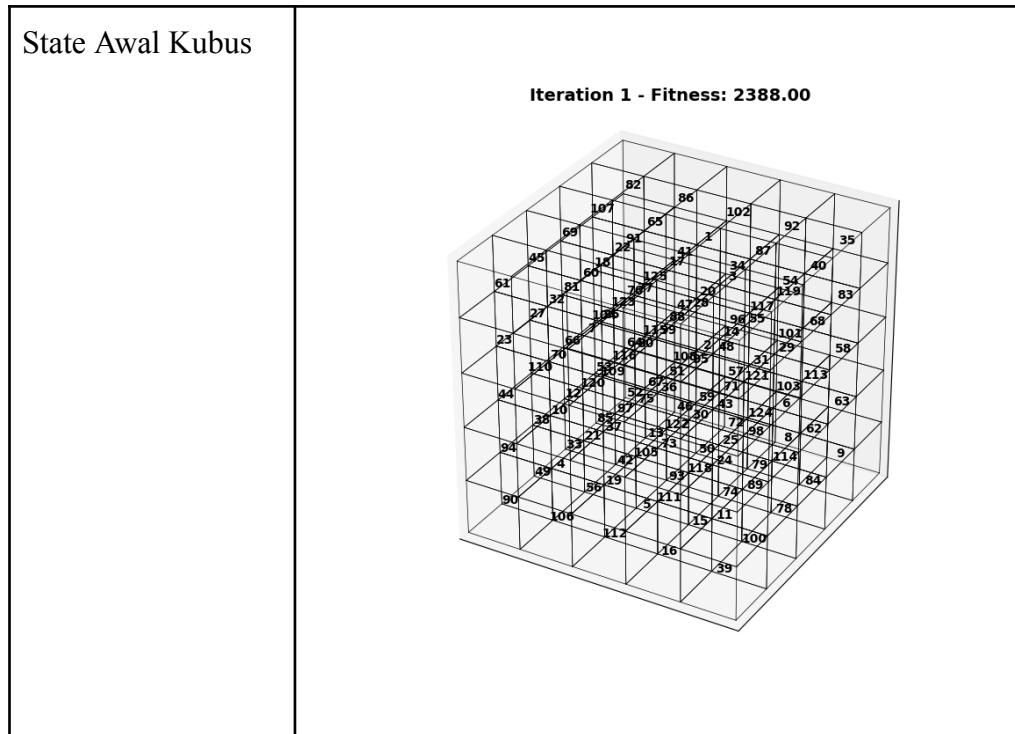
```
[[ 67  54  52 107  46]
 [ 70  46  61  10 121]
 [ 36  18  92  60 102]
 [114  73  74  76  31]
 [ 20  70  16  92  20]]
```

Layer 5:

```
[[ 65  79  75  54  49]
 [ 7  32  57 110 110]
 [100  79  66  32  32]
 [ 66  30  49  65  65]
 [ 65  79  75  49  65]]
```

Final Fitness: 666.53

## 4. Eksperimen IV



State Akhir Kubus

Final cube state after climbing:

Layer 1:

```
[[ 76 108 57 34 73]
 [ 97 32 26 112 69]
 [ 17 78 93 74 29]
 [122 13 6 79 104]
 [ 5 56 92 47 97]]
```

Layer 2:

```
[[ 89 15 19 99 45]
 [ 45 28 105 29 103]
 [ 30 76 20 86 88]
 [ 69 84 52 92 35]
 [ 82 108 57 45 47]]
```

Layer 3:

```
[[ 9 46 100 92 61]
 [ 60 86 1 92 37]
 [119 50 51 71 9]
 [ 65 115 10 31 113]
 [ 86 8 96 9 58]]
```

Layer 4:

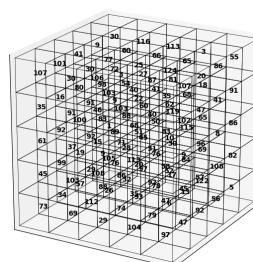
```
[[ 72 106 80 16 35]
 [ 27 54 103 91 91]
 [ 81 41 22 103 83]
 [ 18 68 62 40 68]
 [ 91 41 47 102 83]]
```

Layer 5:

```
[[ 30 9 41 101 107]
 [116 60 77 30 30]
 [113 66 25 3 98]
 [ 3 65 124 87 40]
 [ 55 86 20 107 35]]
```

Final Fitness: 716.60

Iteration 1499 - Fitness: 716.60



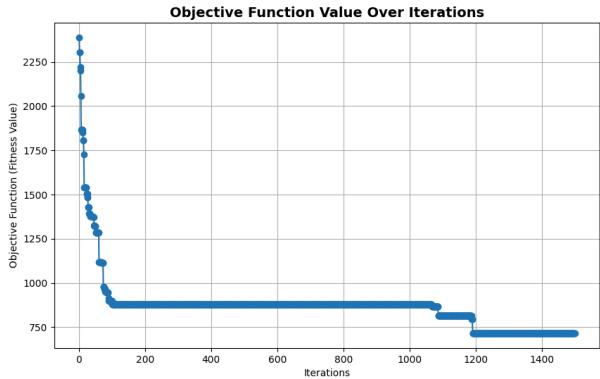
Durasi Pencarian

225.64 detik

Banyak Iterasi

1500

Plot Nilai Objektif



Bukti Eksperimen

2. Hill Climbing with Sideways Move

3. Random Restart Hill Climbing

4. Stochastic Hill Climbing

5. Simulated Annealing

6. Genetic Algorithm

7. Back to Main Menu

Choose the algorithm you want to use: 6

```
Do you want to keep the replay of the cube solving process? (y/n): y
Enter the output file name (without extension): ga_4
The file will be saved as 'ga_4.json' in the 'result/' directory.
```

```
Input the population size: 15
```

```
Input the maximum iteration: 1500
```

```
Input the mutation rate (Between 0 and 1): 0.1
```

Final cube state is reached after 1499 iterations in 225.64 seconds.

Final cube state after climbing:

Layer 1:

```
[[ 76 108 57 34 73]
 [ 97 32 26 112 69]
 [ 17 78 93 74 29]
 [122 13 6 79 104]
 [ 5 56 92 47 97]]]
```

Layer 2:

```
[[ 89 15 19 99 45]
 [ 45 28 105 29 103]
 [ 30 76 20 86 88]
 [ 69 84 52 92 35]
 [ 82 108 57 45 47]]]
```

Layer 3:

```
[[ 9 46 100 92 61]
 [ 60 86 1 92 37]
 [119 50 51 71 9]
 [ 65 115 10 31 113]
 [ 86 8 96 9 58]]]
```

Layer 4:

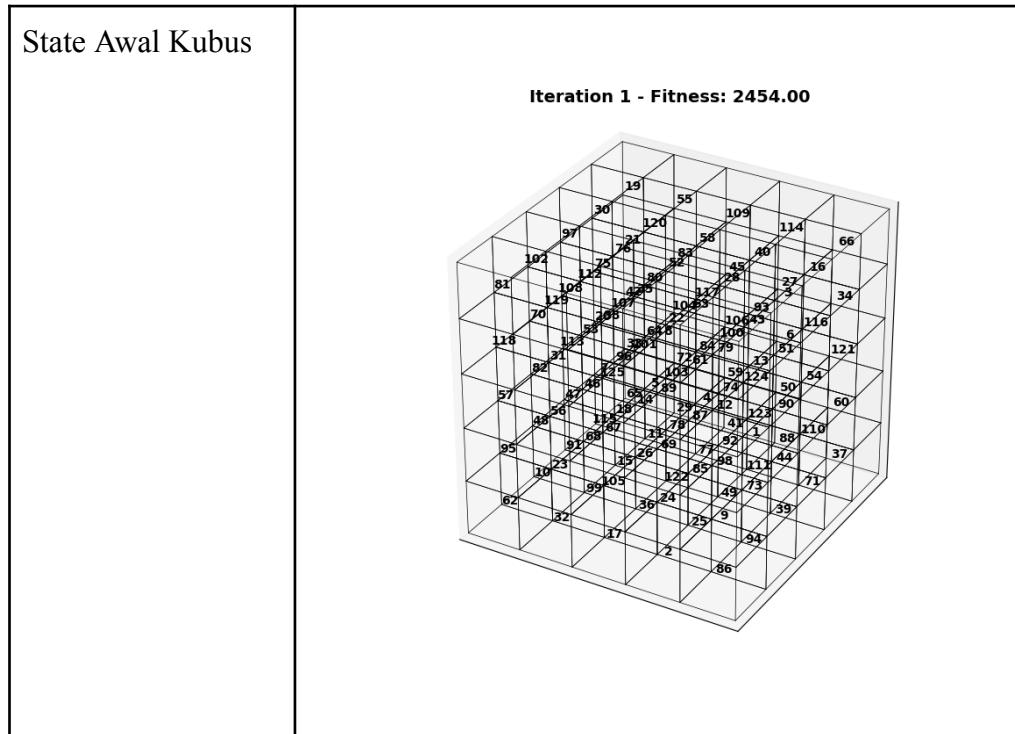
```
[[ 72 106 80 16 35]
 [ 27 54 103 91 91]
 [ 81 41 22 103 83]
 [ 18 68 62 40 68]
 [ 91 41 47 102 83]]]
```

Layer 5:

```
[[ 30 9 41 101 107]
 [116 60 77 30 30]
 [113 66 25 3 98]
 [ 3 65 124 87 40]
 [ 55 86 20 107 35]]]
```

Final Fitness: 716.60

## 5. Eksperimen V



State Akhir Kubus

Final cube state after climbing:

Layer 1:

```
[[ 36  81  47  73  98]
 [ 15  72  67  51 102]
 [102  56  67  63  43]
 [ 86  11  77  92  71]
 [ 79  56  67  79  43]]
```

Layer 2:

```
[[ 41  89  58  39  59]
 [ 75  85  80  41  79]
 [ 52   9  78  64  75]
 [ 45   9 117  98  74]
 [109  98  41  69  41]]
```

Layer 3:

```
[[73 30 86 98 51]
 [94 98 43 98  1]
 [66 98 51 22 74]
 [71 88 66 40 54]
 [51 50 79  6 98]]
```

Layer 4:

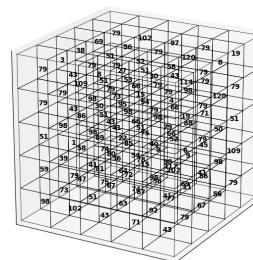
```
[[ 79    8 109  79  79]
 [ 51  53  79  98  43]
 [ 43  79  15  95  51]
 [ 79  98    4  79  66]
 [ 79 120  79  19  79]]
```

Layer 5:

```
[[ 79  69  38    3  79]
 [102  56  51  79  43]
 [ 97  79  52  27  51]
 [ 79 120  58  30  66]
 [ 19    8  79 114  79]]
```

Final Fitness: 533.05

Iteration 1999 - Fitness: 533.05



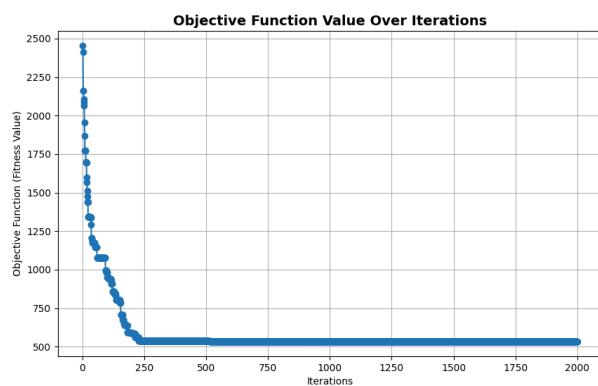
Durasi Pencarian

403.00 detik

Banyak Iterasi

2000

Plot Nilai Objektif



## Bukti Eksperimen

- 2. Hill Climbing with Sideways Move
- 3. Random Restart Hill Climbing
- 4. Stochastic Hill Climbing
- 5. Simulated Annealing
- 6. Genetic Algorithm
- 7. Back to Main Menu

Choose the algorithm you want to use: 6

Do you want to keep the replay of the cube solving process? (y/n): y  
Enter the output file name (without extension): ga\_5  
The file will be saved as 'ga\_5.json' in the 'result/' directory.

Input the population size: 15  
Input the maximum iteration: 2000  
Input the mutation rate (Between 0 and 1): 0.1

Final cube state is reached after 1999 iterations in 403.00 seconds.

Final cube state after climbing:

Layer 1:

```
[[ 36  81  47  73  98]
 [ 15  72  67  51 102]
 [102  56  67  63  43]
 [ 86  11  77  92  71]
 [ 79  56  67  79  43]]
```

Layer 2:

```
[[ 41  89  58  39  59]
 [ 75  85  80  41  79]
 [ 52  9  78  64  75]
 [ 45  9 117  98  74]
 [109  98  41  69  41]]
```

Layer 3:

```
[[73 30 86 98 51]
 [94 98 43 98  1]
 [66 98 51 22 74]
 [71 88 66 40 54]
 [51 50 79  6 98]]
```

Layer 4:

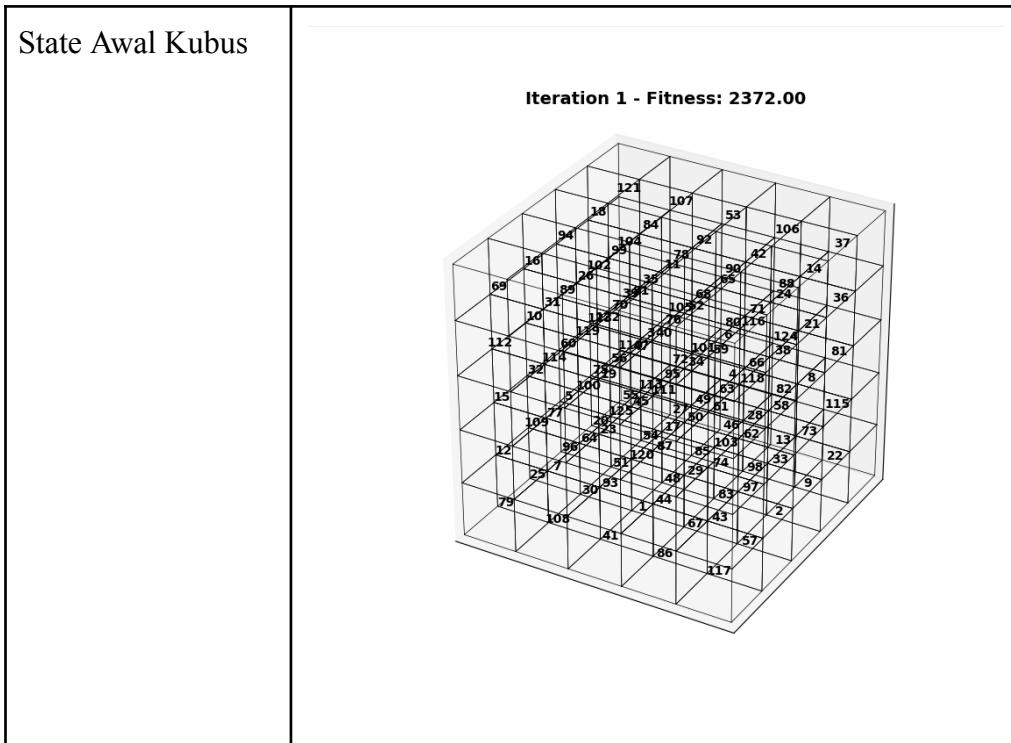
```
[[ 79   8 109  79  79]
 [ 51  53  79  98  43]
 [ 43  79  15  95  51]
 [ 79  98   4  79  66]
 [ 79 120  79  19  79]]
```

Layer 5:

```
[[ 79  69  38   3  79]
 [102  56  51  79  43]
 [ 97  79  52  27  51]
 [ 79 120  58  30  66]
 [ 19   8  79 114  79]]
```

Final Fitness: 533.05

## 6. Eksperimen VI



State Akhir Kubus

Final cube state after climbing:

Layer 1:

```
[[ 58 101 80 30 72]
 [ 97 4 106 87 25]
 [ 53 85 20 83 65]
 [ 37 54 41 119 26]
 [ 96 124 37 33 107]]
```

Layer 2:

```
[[ 43 59 93 37 74]
 [ 81 64 57 29 62]
 [ 24 110 42 121 4]
 [ 1 88 84 28 125]
 [115 37 40 27 115]]
```

Layer 3:

```
[[ 15 77 23 116 59]
 [ 32 116 87 50 81]
 [100 11 95 63 53]
 [123 3 80 66 17]
 [ 37 117 72 33 22]]
```

Layer 4:

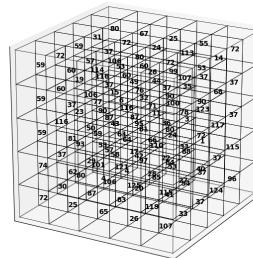
```
[[106 114 19 60 59]
 [ 60 60 37 106 37]
 [ 99 44 76 6 90]
 [ 37 35 90 71 87]
 [ 37 68 90 78 36]]
```

Layer 5:

```
[[ 80 31 59 72 59]
 [ 67 72 37 57 60]
 [ 25 24 80 53 116]
 [ 55 113 72 26 49]
 [ 72 14 53 107 37]]
```

Final Fitness: 741.81

Iteration 2499 - Fitness: 741.81



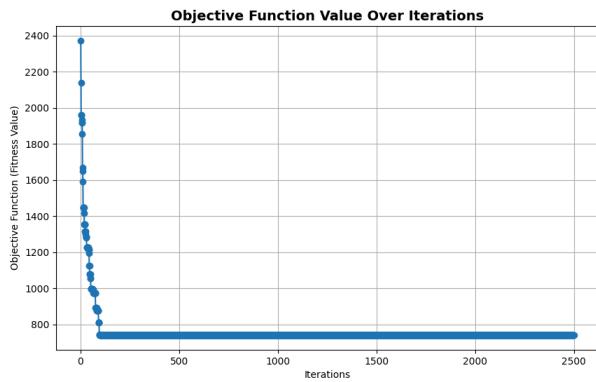
Durasi Pencarian

596.34 detik

Banyak Iterasi

2500

Plot Nilai Objektif



Bukti Eksperimen

2. Hill Climbing with Sideways Move  
3. Random Restart Hill Climbing  
4. Stochastic Hill Climbing  
5. Simulated Annealing  
6. Genetic Algorithm  
7. Back to Main Menu  
Choose the algorithm you want to use: 6

Do you want to keep the replay of the cube solving process? (y/n): y  
Enter the output file name (without extension): ga\_6  
The file will be saved as 'ga\_6.json' in the 'result/' directory.

Input the population size: 15  
Input the maximum iteration: 2500  
Input the mutation rate (Between 0 and 1): 0.1

Final cube state is reached after 2499 iterations in 596.34 seconds.

Final cube state after climbing:

Layer 1:  
[[ 58 101 80 30 72]  
[ 97 4 106 87 25]  
[ 53 85 20 83 65]  
[ 37 54 41 119 26]  
[ 96 124 37 33 107]]

Layer 2:  
[[ 43 59 93 37 74]  
[ 81 64 57 29 62]  
[ 24 110 42 121 4]  
[ 1 88 84 28 125]  
[115 37 40 27 115]]

Layer 3:  
[[ 15 77 23 116 59]  
[ 32 116 87 50 81]  
[100 11 95 63 53]  
[123 3 80 66 17]  
[ 37 117 72 33 22]]

Layer 4:  
[[106 114 19 60 59]  
[ 60 60 37 106 37]  
[ 99 44 76 6 90]  
[ 37 35 90 71 87]  
[ 37 68 90 78 36]]

Layer 5:  
[[ 80 31 59 72 59]  
[ 67 72 37 57 60]  
[ 25 24 80 53 116]  
[ 55 113 72 26 49]  
[ 72 14 53 107 37]]

Final Fitness: 741.81

## 4. Analisis

### A. Seberapa dekat tiap-tiap algoritma bisa mendekati global optima? Mengapa hasilnya demikian?

#### 1. Algoritma Steepest Ascent Hill Climbing

Algoritma Steepest Ascent dalam tiga kali percobaannya menghasilkan nilai 147.71, 417.71, dan 207.32 dalam kisaran iterasi di bawah 105 – 120 dengan nilai kondisi awal yang berada dalam rentang 2000 – 3000.

Algoritma Steepest Ascent memberikan nilai kondisi terminasi yang beragam dikarenakan algoritma ini hanya menerima *successor state* bernilai lebih besar sehingga algoritma ini lebih rentan terjebak dalam hasil yang memiliki nilai lokal optimum. Algoritma Steepest Ascent tidak dapat keluar dari kondisi terjebak dalam lokal optimum sehingga memberikan hasil yang tidak dapat diprediksi dan lebih beragam akibat akan langsungnya mengalami terminasi apabila sudah menemui lokal optimum.

#### 2. Algoritma Hill Climbing with Sideways Move

Algoritma Hill Climbing with Sideways Move dalam tiga kali percobaannya menghasilkan nilai 147.71, 417.41, dan 206.77 dalam rentang iterasi 100-120. Algoritma Steepest Ascent memberikan nilai kondisi terminasi yang lebih beragam dan cenderung lebih mendekati global optima karena adanya kemungkinan melakukan eksplorasi pada kondisi yang mempertemukan permasalahan dengan situasi *successor state* memiliki nilai yang sama dengan *current state*. Kondisi ini memungkinkan permasalahan untuk keluar dari kondisi keterjebakan dalam lokal optimum.

#### 3. Algoritma Random Restart Hill Climbing

Algoritma Random Restart Hill Climbing memberikan kembalian *state* dengan nilai 147.71, 91.95, dan 114.67 yang berada dalam kurun iterasi lebih dari 2000 iterasi. Algoritma ini mengembalikan hasil yang cenderung mendekati global optima dikarenakan kemampuan yang

dimilikinya dalam mengulang kembali percobaan saat menemui kondisi terjebak dalam lokal optimum memberikan kesempatan eksplorasi solusi yang lebih luas sehingga dapat memperbesar peluang tercapainya solusi yang bersifat global optimum. Namun, algoritma ini memerlukan waktu yang cukup lama dalam pengeksekusiannya, pada percobaan yang dilakukan penulis, pengeksekusian algoritma ini memakan waktu di atas 1200 detik karena iterasi maksimum yang dilakukan mencapai 20 kali pengulangan sehingga memakan waktu dalam melakukan Steepest Ascent Hill Climb. Sehingga, algoritma ini cocok digunakan untuk penyelesaian masalah yang lebih mengutamakan efektivitas solusi dibandingkan efisiensi waktu.

#### 4. Algoritma Stochastic Hill Climbing

Algoritma Stochastic Hill Climbing menghasilkan kembalian yang memiliki nilai 371.17, 315.64, dan 236.57 dalam rentang 1000 iterasi. Hal ini disebabkan oleh algoritma stochastic hill climbing yang hanya dapat menerima nilai terbaik dari *successor state* yang dipilih secara random dari seluruh *successor* yang dimiliki oleh suatu state. Faktor penentuan *successor state* yang bersifat acak (tidak didasarkan pada suatu nilai yang diharapkan) disertai dengan terbatasnya fleksibilitas algoritma dalam menerima *successor state* yang ada membuat *state* kembalian yang dihasilkan cenderung kurang mendekati global optima.

#### 5. Algoritma Simulated Annealing

Algoritma Simulated Annealing menghasilkan kembalian yang memiliki nilai 149.03, 137.67, dan 146.69 dalam rentang iterasi 2000 iterasi serta memakan waktu 300 – 600 detik. Hasil yang dikembalikan cenderung mendekati global optimum karena *simulated annealing* memberikan fleksibilitas lebih untuk permasalahan melakukan eksplorasi solusi sehingga memungkinkan permasalahan keluar dari kondisi keterjebakan dalam kondisi lokal optimum berkat peran dari nilai probabilitas.

#### 6. Algoritma Genetic Algorithm

Algoritma Genetic Algorithm mengembalikan *state* yang bernilai 500 – 700 dan mengisyaratkan kurang dekatnya nilai dari *state* kembalian dengan global optimum. Hal ini disebabkan masih dapat dioptimalkannya fungsi yang melibatkan proses *crossover* dan *mutation* dari algoritma Genetic Algorithm yang dibuat.

## B. Bagaimana perbandingan hasil pencarian tiap-tiap algoritma dengan algoritma local search yang lain?

### 1. Algoritma Steepest Ascent Hill Climbing

Pada Eksperimen I dan II, algoritma ini menunjukkan hasil state akhir yang sama persis dengan algoritma with sideways move, dengan waktu yang kurang lebih hampir sama. Hasil state akhir juga merupakan **yang terbaik** diantara algoritma-algoritma lainnya dan dengan waktu **yang tercepat**, meski pada Eksperimen II jadi yang terburuk. Pada Eksperimen III juga tidak jauh berbeda dengan sideways move.

### 2. Algoritma Hill Climbing with Sideways Move

Pada Eksperimen I dan II, algoritma ini menunjukkan hasil state akhir yang sama persis dengan algoritma steepest ascent, dengan waktu yang kurang lebih hampir sama. Hasil state akhir juga merupakan yang **terbaik** diantara algoritma-algoritma lainnya dan dengan waktu yang **tercepat**, meski pada Eksperimen II jadi yang terburuk. Mengapa bisa sama? Hal yang paling mungkin dalam pencarian tidak terlalu banyak shoulder effect (optimum lokal rata) sehingga terlihat sama saja. Pada Eksperimen III, mendapat nilai yang lebih baik dari steepest ascent dengan perbedaan 2 iterasi

### 3. Algoritma Random Restart Hill Climbing

Pada Eksperimen 1, algoritma ini menunjukkan hasil yang sama saja (**terbaik tercepat**) dengan 2 algoritma sebelumnya padahal telah 20 kali iterasi hingga memakan waktu hingga 20 menit (asumsi setiap restart 1 menit).

Pada Eksperimen 2 dan 3, algoritma ini menunjukkan hasil **terbaik** dibandingkan yang lainnya namun tentu dengan total lebih dari 2000 iterasi dan **waktu yang sangat panjang**.

#### 4. Algoritma Stochastic Hill Climbing

Pada Eksperimen 1, Algoritma ini **tidak menunjukkan hasil yang paling baik, padahal waktunya lebih lama** dari steepest ascent dan sideways move dengan 1000 kali iterasi. Pada eksperimen 2 dan 3 juga tidak jauh berbeda, dengan 1000 iterasi hanya dapat mencapai 300 iterasi yang lebih tinggi dari yang lainnya. Namun, per iterasinya tentu lebih cepat karena bersifat random.

#### 5. Algoritma Simulated Annealing

Pada Eksperimen 1 dan 3, Algoritma ini **hampir paling baik** mendekati steepest dan sideways climbing, **hanya saja dengan waktu lebih lama** karena hingga 2000 iterasi. Sementara pada eksperimen 2 menunjukkan hasil yang kurang baik. Seharusnya, untuk simulated annealing maksimum iterasi dapat ditingkatkan karena nilainya seringkali cukup baik hasilnya seperti random restart yang cukup baik dalam eksperimen kami namun dengan waktu yang lebih singkat.

#### 6. Algoritma Genetic Algorithm

Karena kurang optimalnya heuristik yang dapat lebih ditingkatkan dengan fungsi crossover harusnya lebih ditingkatkan. Dengan waktu dan iterasi yang sama dengan stochastic, hasil pencarian masih dapat dikatakan kurang baik dan mendekati optimum.

### C. Bagaimana perbandingan durasi proses pencarian tiap algoritma relatif terhadap algoritma lainnya?

Algoritma yang memiliki durasi proses pencarian paling lama adalah algoritma random restart karena penggunaan strategi eksplorasi yang intensif. Algoritma ini menggunakan pendekatan pencarian berulang dengan memulai kembali dari titik

awal yang berbeda sebanyak 20 kali. Setiap kali restart dilakukan, algoritma melakukan pencarian lokal dengan metode steepest hill climb, yang bertujuan menemukan solusi optimal dalam ruang pencarian lokal di setiap iterasi. Namun, proses ini memerlukan waktu yang signifikan, terutama karena setiap iterasi steepest hill climb membutuhkan banyak langkah perhitungan untuk mencapai puncak lokalnya sebelum melakukan restart berikutnya. Oleh sebab itu, akumulasi dari semua langkah ini menghasilkan durasi pencarian yang lebih lama dibandingkan algoritma lainnya.

**D. Seberapa konsisten hasil akhir yang didapatkan dari tiap-tiap eksperimen yang dilakukan**

Pada tiap-tiap eksperimen, hasil akhir steepest ascent dan sideways selalu konsisten memiliki nilai yang mirip karena sedikitnya kasus shoulder optimum local pada kasus cube ini. Random restart pada eksperimen kami juga cenderung sangat konsisten memiliki nilai yang paling baik, namun tentu memakan waktu. Pencarian stochastic tidak konsisten karena sifatnya yang sangat random, namun cukup dikatakan bersifat rata-rata hasil capaian pada setiap eksperimennya (tidak pernah terbaik maupun sangat terburuk). Pencarian simulated annealing meski bersifat random namun cukup terarah, namun sayang pada salah satu pencarian menunjukkan performa yang kurang, padahal performanya sangat bagus pada eksperimen lainnya, menunjukkan kurangnya konsistensi simulated annealing.

**E. Bagaimana pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada Genetic Algorithm?**

a. Pengaruh banyak iterasi

Semakin banyak iterasi yang dilakukan pada genetic algorithm, semakin besar peluang algoritma untuk mendekati solusi yang lebih baik karena di setiap iterasi dilakukan seleksi antara individu yang terbaik, crossover antarindividu, dan mutasi dengan peluang tertentu untuk menghasilkan anak yang lebih baik. Iterasi yang semakin banyak akan membuat anak memiliki fitness value yang lebih stabil dan memungkinkan algoritma untuk terhindar untuk berhenti di solusi lokal tanpa mencapai solusi global. Selain itu, mekanisme crossover yang kami terapkan di setiap iterasi dengan menukar baris atau kolom atau pilar terburuk (memiliki deviasi yang paling jauh dari 315) pada induk 1

dengan baris atau kolom atau pilar terbaik (memiliki deviasi yang paling dekat dari 315) pada induk 2 bertujuan untuk menghasilkan anak dengan kualitas yang lebih baik.

Namun, pada jumlah iterasi yang cukup tinggi, algoritma bisa saja stuck di solusi lokal tanpa menemukan solusi global. Pada awalnya, iterasi yang semakin meningkat akan meningkatkan peluang eksplorasi solusi baru, tetapi setelah beberapa iterasi, peningkatan signifikan akan berkurang dan stabil di sekitar nilai tertentu, di mana peningkatan jumlah iterasi hanya memberikan perbaikan kecil pada solusi. Jumlah iterasi yang tinggi memungkinkan algoritma untuk terus memperbaiki solusi selama cukup lama, tetapi pada titik tertentu, peningkatan kinerja menjadi stagnan.

Pada hasil eksperimen, diantara iterasi berjumlah 1500, 2000, dan 2500, didapatkan performa algoritma yang lebih baik pada jumlah iterasi 2000. Pada awal, algoritma bertujuan untuk mengeksplorasi ruang pencarian untuk menemukan beragam solusi yang berpotensi baik. Pada jumlah iterasi 2000, algoritma mencapai nilai yang stagnan, di mana algoritma mulai memfokuskan pencarinya pada solusi yang sudah ditemukan. Ketika jumlah iterasi ditingkatkan hingga 2500, algoritma telah menghabiskan banyak waktu untuk mengeksplor solusi yang telah ditemukan, tetapi tidak mendapatkan hasil yg signifikan karena ruang pencarian sudah jenuh atau konvergen ke solusi lokal.

#### b. Pengaruh banyak populasi

Ukuran populasi yang besar akan memungkinkan algoritma untuk mengeksplorasi solusi ruang pencarian yang lebih luas sehingga meningkatkan peluang untuk menghasilkan anak dengan fitness mendekati 0. Populasi yang besar juga cenderung memiliki variasi individu yang lebih tinggi, sehingga algoritma memiliki lebih banyak kemungkinan solusi untuk dipertimbangkan.

Pada hasil eksperimen, diantara populasi berjumlah 5, 10, dan 15, didapatkan performa algoritma yang lebih baik pada jumlah populasi 15. Hal ini sesuai dengan analisis di atas. Dengan populasi 15, algoritma dapat menjelajahi ruang pencarian yang lebih luas karena memiliki lebih banyak individu yang

dengan variasi konfigurasi berbeda. Dengan lebih banyak individu, mekanisme seleksi roulette wheel menjadi lebih kompetitif sehingga probabilitas individu dengan fitness lebih tinggi dapat dipilih dan mempercepat konvergensi menuju solusi yang lebih baik.

## PENUTUP

### 1. Kesimpulan

Dari hasil observasi, didapatkan bahwa algoritma yang memberikan proporsi waktu dan nilai *state* hasil kembalian terbaik adalah algoritma *Simulated Annealing*. Algoritma ini memberikan hasil yang nilainya serupa dengan algoritma Random Restart yang dapat mencapai nilai paling mendekati global optimum, namun memiliki waktu eksekusi yang lebih cepat dibandingkan Random Restart untuk mendapatkan hasil dengan nilai *state* yang serupa.

### 2. Saran

Saran yang diberikan adalah:

1. Menambah maksimum iterasi simulated annealing hingga lebih dari 2000
2. Melakukan beberapa variasi algoritma genetic algorithm dengan heuristik yang terbaik
3. Melakukan lebih banyak percobaan dengan kombinasi parameter yang lebih variatif

### 3. Pembagian Tugas

Penanggung Jawab	Tugas	
Kayla Namira Mariadi	13522050	Genetic Algorithm
Andhita Naura Hariyano	13522060	Simulated Annealing
Salsabiila	13522062	Hill Climbing, Visualizer
Shulha	13522087	Genetic Algorithm

## REFERENSI

Çiçekli, İ., “Heuristic Search: Local Search Algorithms”. 2015. Diakses pada 2 Oktober dari [https://web.cs.hacettepe.edu.tr/~ilyas/Courses/VBM688/lec05\\_localssearch.pdf](https://web.cs.hacettepe.edu.tr/~ilyas/Courses/VBM688/lec05_localssearch.pdf)

Trump, D., “Magic Features of the Three-Dimensional Magic Cube”. Diakses pada 2 Oktober dari <https://www.magischvierkant.com/three-dimensional-eng/magic-features/>

Trump, D., “Magic Cubes 1”. Diakses pada 2 Oktober dari <https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>

Masayu L. K., “IF3170\_Materi03\_Seg01\_BeyondClassicalSearch\_LocalSearch” *IF3170 Intelelegensi Artifisial*, 2023. [Online] Diakses pada 2 Oktober 2024

Masayu L. K., “IF3170\_Materi03\_Seg03\_BeyondClassicalSearch\_HillClimbing” *IF3170 Intelelegensi Artifisial*, 2023. [Online] Diakses pada 2 Oktober 2024

Masayu L. K., “IF3170\_Materi03\_Seg04\_BeyondClassicalSearch\_SimulatedAnnealing” *IF3170 Intelelegensi Artifisial*, 2023. [Online] Diakses pada 2 Oktober 2024

Masayu L. K., “IF3170\_Materi03\_Seg05\_BeyondClassicalSearch\_GeneticAlgorithm”