TUGAS BESAR IF3070

Dasar Intelegensi Artifisial

Pencarian Solusi Diagonal Magic Cube dengan Local Search

Disusun oleh:

Regina Deva Carissa	18222040
Kezia Caren Cahyadi	18222041
Ananda Farhan Raihandra	18222084

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

I. Deskripsi Persoalan

Tugas Kecil ini memiliki tujuan untuk menyelesaikan masalah Diagonal Magic Cube dengan menggunakan algoritma *local search*. Diagonal Magic Cube adalah sebuah kubus berukuran NxNxN yang tersusun dari angka 1 sampai N pangkat 3 tanpa pengulangan (dengan N sebagai panjang sisi) dan diacak. Setiap angka disusun hingga memenuhi syarat, yaitu:

1. Jumlah angka pada setiap baris sama dengan magic number
2. Jumlah angka pada setiap kolom sama dengan magic number
3. Jumlah angka pada setiap tiang harus sama dengan magic number
4. Jumlah angka pada seluruh diagonal kubus sama dengan magic number.

Implementasi dari local search digunakan untuk memperbaiki susunan dari Magic Cube sehingga memenuhi syarat yang ada. Algoritma yang digunakan adalah Steepest Ascent Hill-Climbing, Hill-Climbing with Sideways Move, Random Restart Hill-Climbing, Stochastic Hill-Climbing, Simulated Annealing, dan Genetic Algorithm.

II. Pembahasan

1. Pemilihan Objective Function

Objective function adalah fungsi yang digunakan untuk mengevaluasi suatu state dari kubus. Objective function ini harus mengembalikan nilai numerik yang menggambarkan seberapa jauh solusi dari penerapan *local search* dengan solusi yang optimal. Rumus yang bisa digunakan adalah:

$$f(kubus) = \sum_{baris} |\text{sum(baris)} - \text{MN}| + \sum_{kolom} |\text{sum(kolom)} - \text{MN}| + \sum_{tiang} |\text{sum(tiang)} - \text{MN}| + \sum_{diagonal} |\text{sum(diagonal)} - \text{MN}|$$

MN disini merupakan *magic number* dari kubus. *Magic number* sendiri dapat dihitung menggunakan rumus berikut:

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

Nilai yang ingin dicapai oleh fungsi ini adalah 0, yang berarti kubus memenuhi semua kondisi magic cube, yaitu setiap baris, kolom, tiang, dan diagonal memiliki jumlah yang sama dengan magic number. Function ini menggunakan cara yang sederhana, yaitu menghitung selisih antara sum pada elemen-elemen kubus dan magic number untuk memastikan kondisi ideal dari kubus. Selain itu, dengan mengevaluasi seluruh kolom, baris, tiang, hingga diagonal, maka dapat dipastikan bahwa seluruh aspek kubus diperiksa. Dengan meminimalkan selisih jumlah angka pada setiap baris, kolom, tiang, dan diagonal terhadap magic number, maka solusi yang diinginkan dapat dicapai.

2. Penjelasan Algoritma Research

2.1. Steepest Ascent Hill-climbing

2.1.1. Source code

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

N = 5
max_iteration = 20000
magicSum = N * (N**3 + 1) // 2

def randomize_cube():
    angka = list(range(1, N**3 + 1))
    random.shuffle(angka)
    return np.array(angka).reshape(N, N, N)

def count_objective(cube):
    row_sum_error = np.sum(np.abs(np.sum(cube, axis=0) - magicSum))
    column_sum_error = np.sum(np.abs(np.sum(cube, axis=1) - magicSum))
    depth_sum_error = np.sum(np.abs(np.sum(cube, axis=2) - magicSum))

    diag1_sum = sum(cube[i, i, i] for i in range(N))
    diag2_sum = sum(cube[i, i, N - i - 1] for i in range(N))
    diag3_sum = sum(cube[i, N - i - 1, i] for i in range(N))
    diag4_sum = sum(cube[N - i - 1, i, i] for i in range(N))

    diag_error = abs(diag1_sum - magicSum) + abs(diag2_sum - magicSum)
+ \
                 abs(diag3_sum - magicSum) + abs(diag4_sum - magicSum)

    return row_sum_error + column_sum_error + depth_sum_error +
diag_error

def search_bestNeighbor(cube):
    best_neighbor = cube.copy()
    best_objective = count_objective(cube)

    for _ in range(100):
        neighbor = cube.copy()

        idx1 = tuple(np.random.randint(0, N, size=3))
        idx2 = tuple(np.random.randint(0, N, size=3))
```

```
        while idx1 == idx2:
            idx2 = tuple(np.random.randint(0, N, size=3))

        neighbor[idx1], neighbor[idx2] = neighbor[idx2],
neighbor[idx1]

        neighbor_objective = count_objective(neighbor)

        if neighbor_objective < best_objective:
            best_neighbor = neighbor
            best_objective = neighbor_objective
        elif neighbor_objective == best_objective and best_neighbor is
cube:
            best_neighbor = neighbor

    return best_neighbor, best_objective

def steepest_ascent_hill_climbing():
    initial_cube = randomize_cube()
    cube = initial_cube.copy()
    best_objective = count_objective(cube)
    history = [best_objective]
    start_time = time.time()

    for iterasi in range(max_iteration):
        kubus_baru, objective_baru = search_bestNeighbor(cube)

        if objective_baru >= best_objective:
            break

        cube = kubus_baru
        best_objective = objective_baru
        history.append(best_objective)

    duration = time.time() - start_time
    return initial_cube, cube, best_objective, history, duration,
iterasi

def run_experiment():
    initial_cube, last_cube, last_objective, history, duration,
iterasi = steepest_ascent_hill_climbing()

    print("State Awal Kubus:\n", initial_cube)
    print("\nState Akhir Kubus:\n", last_cube)
    print("\nNilai Objective Function Akhir:", last_objective)
```

```
    print("Durasi Proses Pencarian:", duration, "detik")
    print("Jumlah Iterasi Hingga Berhenti:", iterasi)

    plt.plot(history, label="Objective Function")
    plt.xlabel("Iterasi")
    plt.ylabel("Nilai Objective Function")
    plt.title("Performa Steepest Ascent Hill-Climbing dengan Kondisi
Flat")
    plt.legend()
    plt.show()

run_experiment()
```

2.1.2. Deskripsi Fungsi

1. randomize_cube()

   Fungsi ini dibuat untuk membentuk suatu *cube* dengan letak angka-angka yang selalu *random* setiap eksperimen berlangsung, dengan kata lain merupakan inisialisasi kubus sebelum dilakukan local_search. Fungsi ini menggunakan variabel angka (1-126, dengan jumlah angka ada 125) lalu di *shuffle* untuk membentuk kubus 5x5x5.
2. count_objective()

   Fungsi ini dibuat untuk menghitung *objective function* atau seberapa dekat keadaan kubus sekarang dengan keadaan ideal. Fungsi menghitung jumlah eror dari kolom, baris hingga tiang dengan magic sum (angka yang seharusnya ideal), dan juga menghitung error diagonal. Jumlah error tersebut merupakan angka *objective function* yang didapatkan pada iterasi terakhir, dalam kasus ini saat tidak ada lagi tetangga yang lebih baik dari current state atau keadaan flat.
3. search_BestNeighbor()

   Fungsi ini mencari tetangga/neighbor terbaik dari kubus saat ini dengan cara mencoba menukar dua elemen acak dalam kubus dan menghitung nilai *objective function* untuk tetangga tersebut. Apabila nilai *objective function* dari tetangga lebih kecil, maka tetangga tersebut akan masuk dalam variabel best_neighbor.
4. steepest_ascent_hill_climbing()

   Fungsi ini merupakan fungsi utama dalam algoritma, karena menjalankan implementasi *local search* Steepest Ascent Hill Climbing dalam menyelesaikan kasus kubus ini. Fungsi ini akan mencari solusi paling optimal dari kubus dengan meminimalkan *objective function.* Fungsi dimulai dengan membuat inisialisasi kubus dengan angka-angka yang disusun secara acak, lalu melakukan iterasi sebanyak mungkin dengan tiap iterasi mencari *best neighbor.* Apabila sudah tidak ada lagi tetangga yang lebih baik atau nilainya sama (flat), maka iterasi dihentikan dan fungsi ini juga berhenti.
5. run_experiment()

Fungsi ini merupakan fungsi untuk menjalankan eksperimen dan algoritma yang telah dibuat sebelumnya. Selain itu, fungsi ini juga akan mencetak hasil-hasil yang diperlukan, seperti state awal dan state akhir dari kubus, lama iterasi, jumlah iterasi yang digunakan, *objective function* akhir, dan juga menampilkan grafik dari iterasi dan objective function yang dihasilkan.

2.2. Simulated Annealing

2.2.1. Source Code

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

N = 5
magicSum = N * (N**3 + 1) / 2

def count_objective(cube):
    objective = 0
    for layer in cube:
        for row in layer:
            objective += abs(sum(row) - magicSum)
    for layer in cube:
        for column in range(N):
            column_sum = sum(layer[row][column] for row in range(N))
            objective += abs(column_sum - magicSum)
    for row in range(N):
        for column in range(N):
            pillar_sum = sum(cube[layer][row][column] for layer in range(N))
            objective += abs(pillar_sum - magicSum)
    return objective

def simulated_annealing(initial_cube, suhu_awal, cooldown, max_iteration,
interval=1000):
    cube = np.copy(initial_cube)
    best_cube = np.copy(cube)
    best_objective = count_objective(cube)

    suhu = suhu_awal
    objectiveNow = count_objective(cube)
```

```
    plot_objective = []
    decay_values = []
    stuck_count = 0
    startTime = time.time()

    # Menampilkan state awal kubus
    print("State Awal Kubus:")
    print(initial_cube)
    print("\n")  # Memberi spasi antara state awal dan proses simulasi

    for i in range(max_iteration):
        if suhu <= 0.1:
            suhu = 0.1

        plot_objective.append(objectiveNow)

        new_cube = np.copy(cube)
        x1, y1, z1 = random.randint(0, N-1), random.randint(0, N-1), random.randint(0,
N-1)
        x2, y2, z2 = random.randint(0, N-1), random.randint(0, N-1), random.randint(0,
N-1)

        while (x1, y1, z1) == (x2, y2, z2):  # Hindari pertukaran elemen yang sama
            x2, y2, z2 = random.randint(0, N-1), random.randint(0, N-1),
random.randint(0, N-1)

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

        new_objective = count_objective(new_cube)
        delta_e = objectiveNow - new_objective

        decay_value = math.exp(delta_e / suhu) if suhu > 0 else 0
        decay_values.append(decay_value)

        if delta_e > 0 or decay_value > random.random():
            cube = new_cube
            objectiveNow = new_objective
        else:
            stuck_count += 1  # Increment stuck_count if new configuration is not
accepted

        if objectiveNow < best_objective:
            best_cube = np.copy(cube)
```

```
            best_objective = objectiveNow

        suhu *= cooldown

        if (i + 1) % interval == 0:
            print(f"Iterasi {i+1}, Suhu {suhu:.4f}, Objective Saat Ini {objectiveNow},
Objective Terbaik {best_objective}")

    endTime = time.time()
    durasi = endTime - startTime

    print("\nState Akhir Kubus:")
    print(best_cube)
    print(f"Jumlah Stuck: {stuck_count}")
    print(f"Durasi Total: {durasi:.2f} detik")

    plt.figure()
    plt.plot(plot_objective, label="Nilai Objective")
    plt.xlabel("Iterasi")
    plt.ylabel("Nilai Fungsi Objective")
    plt.title("Perkembangan Fungsi Objective selama Iterasi")
    plt.legend()
    plt.show()

    plt.figure()
    decay_values = [val if val < 1e300 else 1e300 for val in decay_values]
    plt.plot(decay_values, label="e^(Delta E / T)")
    plt.xlabel("Iterasi")
    plt.ylabel("Nilai $e^{\\Delta E / T}$")
    plt.title("$e^{\\Delta E / T}$ Selama Iterasi")
    plt.legend()
    plt.show()

    return best_cube, best_objective, durasi, stuck_count

# Menghasilkan kubus acak awal
initial_cube = np.arange(1, N**3 + 1)
np.random.shuffle(initial_cube)
initial_cube = initial_cube.reshape(N, N, N)

initial_temp = 100.0
cooldown = 0.99
max_iteration = 10000
interval = 1000
```

```
best_cube, best_objective, durasi, stuck_count = simulated_annealing(initial_cube, initial_temp, cooldown, max_iteration, interval)

print("Nilai Objective Terbaik yang Ditemukan:", best_objective)
print("Konfigurasi Kubus Terbaik:")
print(best_cube)
```

2.2.2. Deksripsi Fungsi

1. count_objective()

   Fungsi ini digunakan untuk menghitung *objective function,* seberapa dekat keadaan kubus sekarang dengan keadaan ideal. Fungsi menghitung jumlah eror dari kolom, baris hingga tiang dengan magic sum (angka yang seharusnya ideal), dan nilai total dari semua error merupakan *objective function* yang dari kubus.
2. simulated_annealing()

   Fungsi ini merupakan fungsi utama yang menjalankan algoritma yang sebelumnya sudah dibuat, yaitu *local search* simulated annealing untuk menyelesaikan kubus. Fungsi ini juga mencetak plot dari delta E, iterasi dan objective function (grafik dan angka), jumlah stuck di local optima, hingga state awal, akhir, dan state terbaik dari kubus.

2.3. Genetic Algorithm

2.3.1. Source Code

```python
import numpy as np
import random
import time
from copy import deepcopy


N = 5
MAGIC_NUMBER = N * (N**3 + 1) // 2
POPULATION_SIZE = 50
MAX_ITERATIONS = 500
MUTATION_RATE = 0.1


def calculate_objective(cube):
    error = 0
    for i in range(N):
        error += abs(sum(cube[i, :, :].flatten()) - MAGIC_NUMBER)
        error += abs(sum(cube[:, i, :].flatten()) - MAGIC_NUMBER)
        error += abs(sum(cube[:, :, i].flatten()) - MAGIC_NUMBER)
```

```
    diag1 = np.sum([cube[i, i, i] for i in range(N)])
    diag2 = np.sum([cube[i, i, N - i - 1] for i in range(N)])
    error += abs(diag1 - MAGIC_NUMBER) + abs(diag2 - MAGIC_NUMBER)
    return error


def initialize_population():
    population = []
    for _ in range(POPULATION_SIZE):
        individual = np.arange(1, N**3 + 1)
        np.random.shuffle(individual)
        cube = individual.reshape((N, N, N))
        population.append(cube)
    return population


def tournament_selection(population, k=5):
    selected = random.sample(population, k)
    return min(selected, key=calculate_objective)


def crossover(parent1, parent2):
    idx1, idx2 = sorted(random.sample(range(N**3), 2))
    child1, child2 = deepcopy(parent1.flatten()),
deepcopy(parent2.flatten())
    child1[idx1:idx2], child2[idx1:idx2] =
parent2.flatten()[idx1:idx2], parent1.flatten()[idx1:idx2]
    # Bentuk ulang ke dalam bentuk 3D
    return child1.reshape((N, N, N)), child2.reshape((N, N, N))


def mutate(individual):
    flat = individual.flatten()
    for i in range(len(flat)):
        if random.random() < MUTATION_RATE:
            idx1, idx2 = random.sample(range(len(flat)), 2)
            flat[idx1], flat[idx2] = flat[idx2], flat[idx1]
    return flat.reshape((N, N, N))


def genetic_algorithm():
    population = initialize_population()
    best_solution = None
    best_fitness = float('inf')
```

```
    fitness_over_time = []
    start_time = time.time()

    for iteration in range(MAX_ITERATIONS):
        fitness_scores = [calculate_objective(ind) for ind in
population]
        min_fitness = min(fitness_scores)
        fitness_over_time.append(min_fitness)

        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution =
population[fitness_scores.index(min_fitness)]

        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1 = tournament_selection(population)
            parent2 = tournament_selection(population)
            child1, child2 = crossover(parent1, parent2)
            child1, child2 = mutate(child1), mutate(child2)
            new_population.extend([child1, child2])

        population = new_population[:POPULATION_SIZE]

        print(f"Iterasi {iteration}: Best Fitness = {best_fitness}")

        if best_fitness == 0:
            break
    duration = time.time() - start_time
    print("Hasil akhir:")
    print("Objective Function Akhir:", best_fitness)
    print("Durasi:", duration)
    print("State akhir dari kubus:", best_solution)
    return best_solution, fitness_over_time

best_solution, fitness_over_time = genetic_algorithm()
```

2.3.2. Deskripsi Fungsi

1. calculate_objective(cube)

   Menghitung nilai fitness kubus dengan menjumlahkan selisih antara jumlah angka pada setiap baris, kolom, tiang, dan diagonal utama dengan magic number. Semakin kecil nilai ini, semakin baik solusi (konfigurasi angka dalam kubus) tersebut.
2. initialize_population()

Membentuk populasi awal untuk algoritma genetika. Fungsi ini menghasilkan sejumlah individu (kubus) yang masing-masing terdiri dari angka acak dari 1 hingga N3N^3N3 tanpa pengulangan. Populasi awal ini menjadi dasar bagi proses evolusi algoritma.

3. tournament_selection(population, k=5)

   Memilih *parent* dari populasi menggunakan metode *tournament selection*. Fungsi ini memilih *k* individu secara acak dari populasi, lalu memilih individu dengan *fitness* terbaik di antara mereka sebagai *parent*. Pemilihan ini bertujuan untuk memberikan peluang lebih besar bagi individu yang lebih kuat untuk berkembang biak.

4. crossover(parent1, parent2)

   Membuat dua *child* baru dari dua *parent* dengan melakukan *partially matched crossover* (PMX). Dua posisi dalam kubus dipilih secara acak, lalu bagian di antara kedua posisi ini dipertukarkan antara *parent1* dan *parent2* untuk membentuk *child*. Proses ini menghasilkan keturunan yang memiliki karakteristik gabungan dari kedua *parent*.

5. mutate(individual)

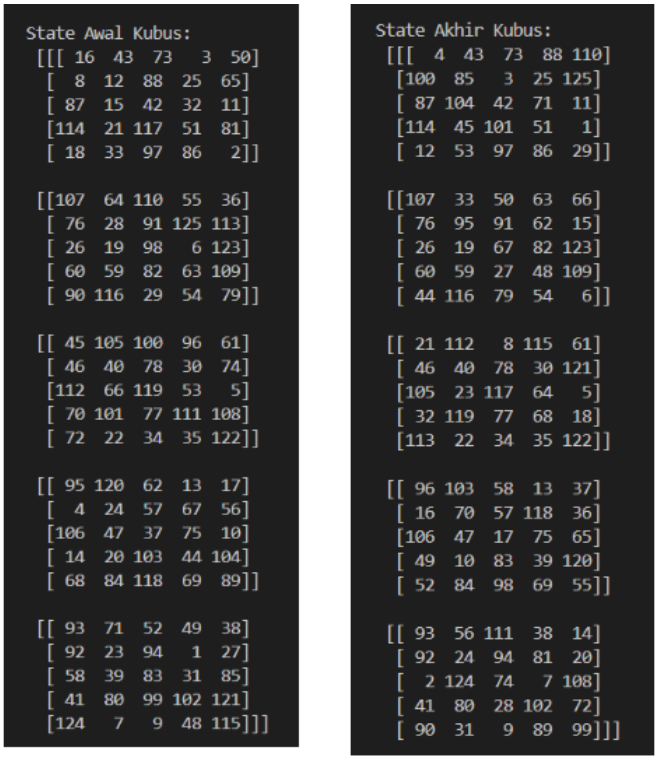
   Memperkenalkan variasi pada individu dengan menukar posisi dua angka secara acak dalam kubus. Mutasi dilakukan dengan probabilitas tertentu untuk memastikan keberagaman dalam populasi dan membantu algoritma menghindari kebuntuan pada *local optimum*.

6. generic_algorithm()

   Fungsi utama yang menjalankan algoritma genetika. Fungsi ini menginisialisasi populasi, lalu menjalankan beberapa iterasi yang meliputi seleksi, *crossover*, mutasi, dan evaluasi populasi baru. Fungsi akan berhenti ketika menemukan solusi yang ideal atau mencapai batas iterasi, lalu mengembalikan solusi terbaik yang ditemukan dan riwayat *fitness* selama iterasi.

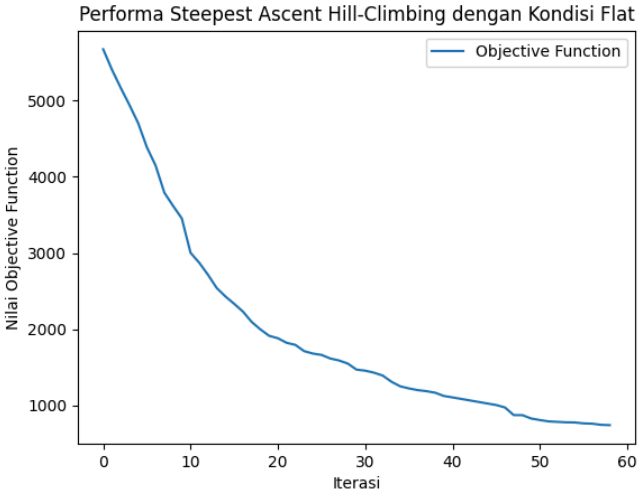
3. Hasil Eksperimen dan Analisis

3.1. Eksperimen Steepest Ascent Hill-Climbing



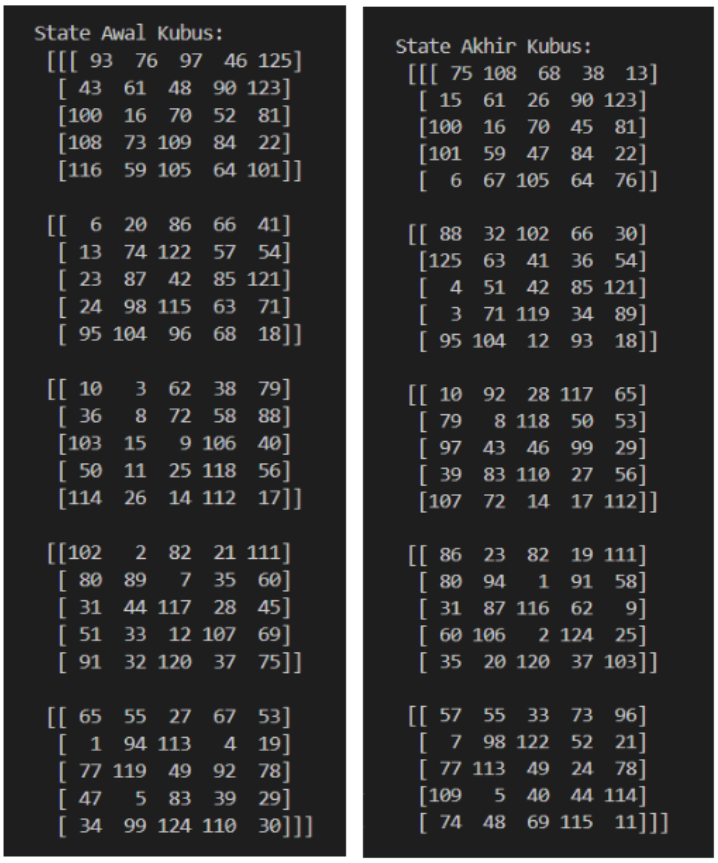
Gambar State Awal dan Akhir percobaan 1

Pada Percobaan 1 dengan algoritma Steepest Ascent Hill-Climbing, nilai objective function akhir adalah 740 dengan durasi sekitar 1,66 detik dan telah melalui 58 iterasi hingga berhenti. berikut grafik nilai objective function dengan banyaknya iterasi yang sudah dilewati



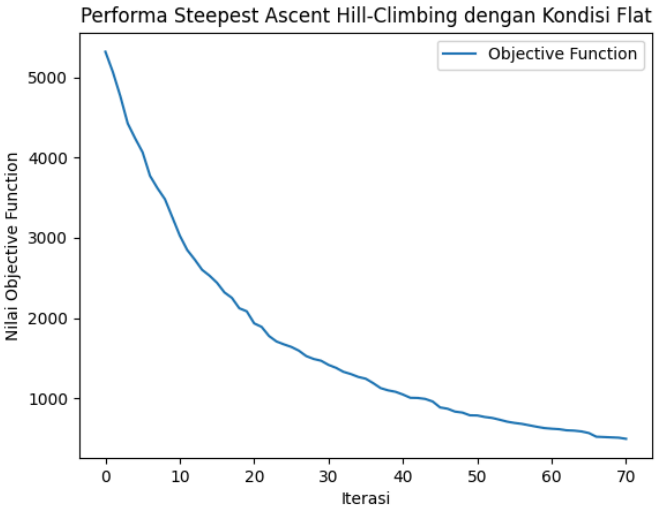
Gambar Plot percobaan 1

Percobaan 2 dengan algoritma yang sama, nilai objective function yang dihasilkan adalah 495, durasi proses pencariannya hingga 1,5 detik. sedangkan jumlah iterasi yang digunakan adalah 70



Gambar State Awal dan Akhir percobaan 2

Sedangkan untuk grafik plot antara nilai objective function dan jumlah iterasi.



Gambar Plot percobaan 2

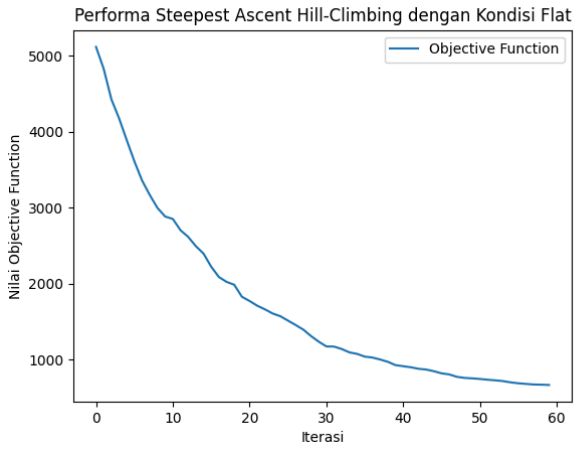
Percobaan terakhir untuk algoritma ini, nilai objective function nya adalah 668 dengan jumlah iterasi sampai 59. selain itu proses pencarian yang dibutuhkan adalah 1,19 detik

```
State Awal Kubus:
[[[ 60 113  47  16 108]
  [ 54  34  72  78 105]
  [ 30  24  96  19  77]
  [117  10   8  88 109]
  [124  53  70  76 125]]

 [[ 92  93   9  89   6]
  [  5  12  71  44  28]
  [ 63  64  39   7 120]
  [ 14  29  58  35  59]
  [101  52  26   3  31]]

 [[ 91  68   4 102 121]
  [ 81  41   2  55  56]
  [ 40  23  84 116   1]
  [ 79  85  37  27 110]
  [ 66  49  67  20  45]]

 [[ 95  22  38  33 111]
  [ 25  87 118 114 103]
  [107  86 115  11  18]
  [ 36  17 112  51  97]
  [119  90  74  69  21]]

 [[ 61  48 100  15  83]
  [ 62  98  80  99  50]
  [104  57  94  13 122]
  [ 73  65 123  75  43]
  [ 46 106  82  42  32]]]
```

```
State Akhir Kubus:
[[[ 84 113  92  16  41]
  [ 77  37  59 123   3]
  [ 30  98 109  19  60]
  [117  10   1  88  96]
  [  5  53  62  76 124]]

 [[ 34 104  80  89   6]
  [ 72  12  64  44 125]
  [ 91  71  24   7 120]
  [ 20 114 115  43  28]
  [101  13  36 121  31]]

 [[ 33  47   4 102 122]
  [ 75 108  55  21  56]
  [ 65   2 111 116   8]
  [ 79 112  39  27  45]
  [ 74  49  78  14  97]]

 [[ 95  22  42  93  63]
  [ 25  32 118  29 103]
  [ 81  86  17 119  18]
  [ 26  68  67  51 110]
  [ 87  90  61  69  23]]

 [[ 70  48 100  15  83]
  [ 66  94   9  99  50]
  [ 54  57  58  52 105]
  [ 73  11  85 107  35]
  [ 46 106  82  38  40]]]
```

State Awal dan Akhir dari percobaan 3

Berikut juga adalah plot dari percobaan ini



Gambar Plot percobaan 3

berikut jika nilainya disandingkan secara bersamaan

Percobaan ke-	Nilai Objective Function	Jumlah Iterasi	Durasi Proses Pencarian (Dibulatkan ke atas)
1	740	58	1,66 detik
2	495	70	1,5 detik
3	668	59	1,19 detik

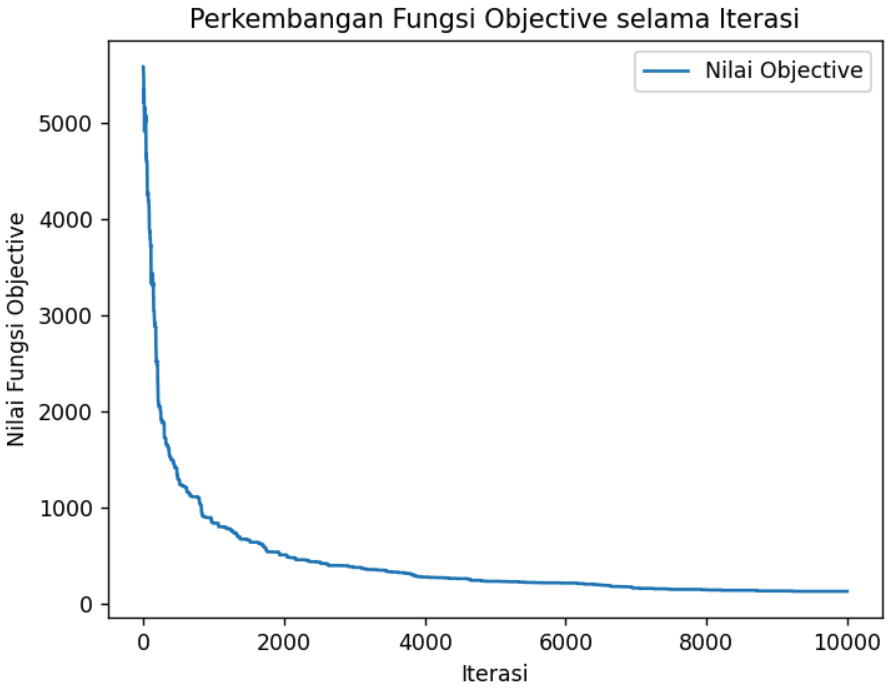
Dapat dilihat, bahwa jumlah iterasi yang rendah dapat menghasilkan nilai objective function yang tinggi. sebaliknya, jumlah iterasi yang tinggi menghasilkan nilai objective yang rendah.

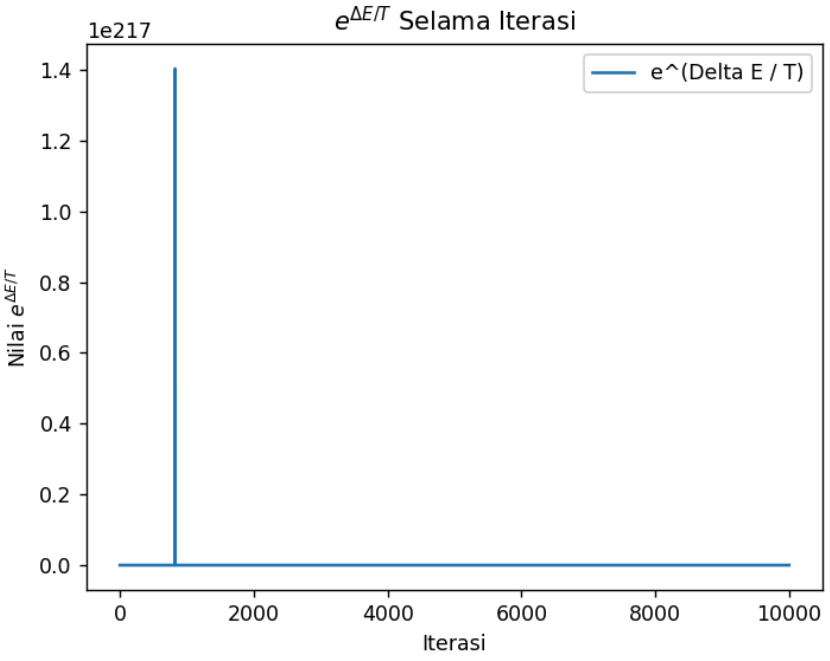
3.2. Eksperimen Simulated Annealing

Percobaan Algoritma pertama:

```
State Awal Kubus:
[[[ 60   1   7  89  35]
  [122  61  97  58  20]
  [ 56  31  49  82  59]
  [ 99  71  72  48  41]
  [ 11  95  52  62  51]]

 [[ 18  88   6  32  27]
  [ 13  93 105 113  40]
  [ 47 108  65 119  90]
  [117  79  43  42  22]
  [ 36 106  23  67  45]]

 [[ 33  74  87  46  80]
  [ 28  91 103  86   2]
  [ 39  75  77   8  96]
  [ 21   9  54 121 112]
  [ 38 125  98  84  55]]

 [[ 69  76  26 114   5]
  [ 94 115  92 124 109]
  [ 29  73 111 123  19]
  [  3  64  53  44  25]
  [ 16 101   4  70  17]]

 [[ 85  66 102 116 118]
  [ 83  14 100  57 104]
  [ 34  12  30  78 110]
  [107  50  81 120  63]
  [ 68  15  37  24  10]]]
```

```
State Akhir Kubus:
[[[ 91  72  89  29  35]
  [101  54  79  56  24]
  [ 10  16  71  95 122]
  [103 111  32  48  14]
  [  2  62  43  87 121]]

 [[  3 113  83  69  47]
  [119  36  76   7  78]
  [ 75  93 107   1  40]
  [ 19  67  20 123  88]
  [ 97   6  34 115  64]]

 [[ 18  63  74  59 100]
  [  4  70  52 124  68]
  [ 98  96  39  53  30]
  [ 80  25  85  38  86]
  [120  61  65  42  28]]

 [[112  44  45 108   5]
  [ 77 106  17  15  99]
  [104  33  60 110   9]
  [ 11  23 117  50 118]
  [ 13 109  73  31  84]]

 [[ 92  26  21  49 125]
  [ 12  51  94 116  46]
  [ 27  82  37  55 114]
  [102  90  58  57   8]
  [ 81  66 105  41  22]]]
```



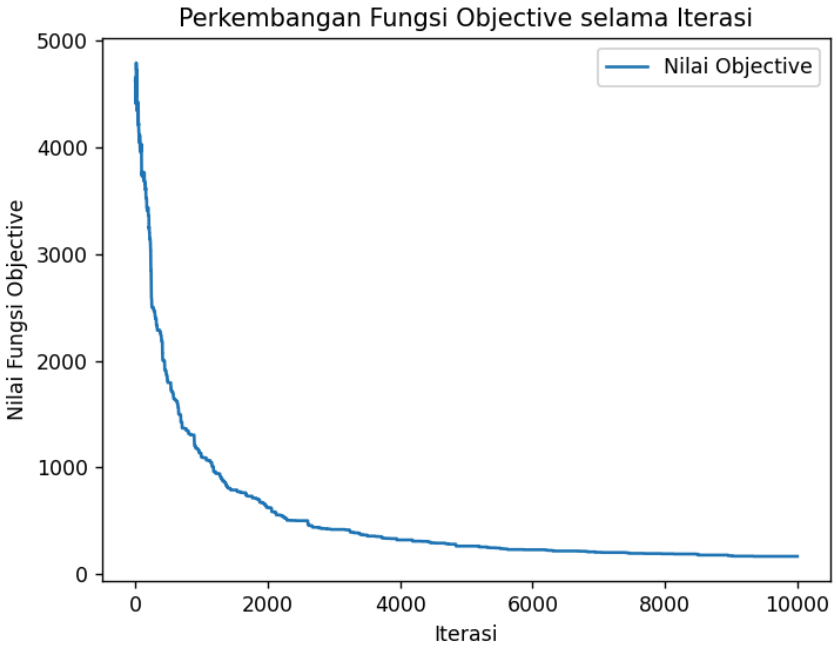


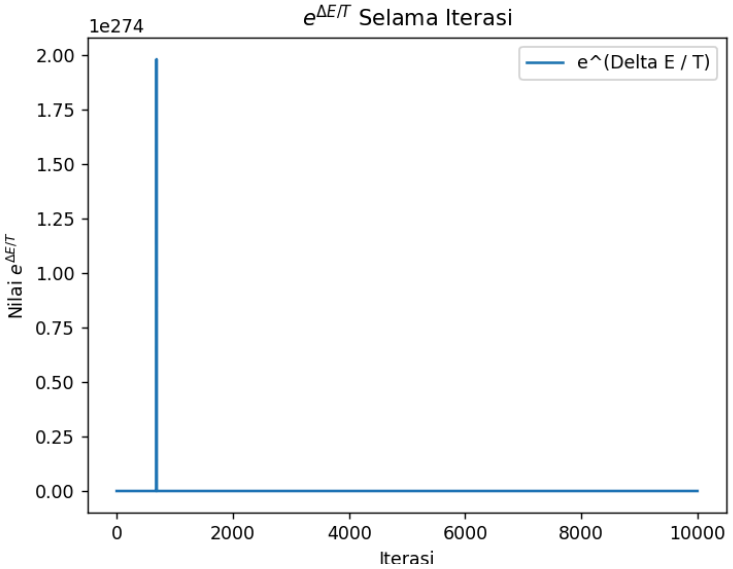

```
Jumlah Stuck: 9542
Durasi Total: 1.96 detik
Nilai Objective Terbaik yang Ditemukan: 136.0
```

Percobaan Algoritma kedua:

```
State Awal Kubus:
[[[ 15 104  75  23  61]
  [ 39   1  44  22  85]
  [107  60   7 116  93]
  [ 56  34  63 105 115]
  [ 14  17  84  36 101]]

 [[  2 102 118   4  38]
  [ 77  88 106  89  66]
  [ 29 114  52  45 125]
  [123   9  42  47  82]
  [ 65  10  86 113  95]]

 [[109  46 108  59  49]
  [ 55  50   5 122  81]
  [ 62  57  48  80 103]
  [ 37 124  35 112  53]
  [ 12  74  31  27 110]]

 [[ 71  87  25  78  26]
  [ 11  43   3   6  96]
  [ 21  76  98  18  33]
  [121  69  24  94  28]
  [119 120  13  91  97]]

 [[ 79  32  16  58  92]
  [ 64  99  41  72  90]
  [ 51  70 111 100  83]
  [  8  20  30  40  54]
  [ 19 117  67  73  68]]]
```

```
State Akhir Kubus:
[[[ 26  97 116  51  25]
  [101  40  44  56  73]
  [ 81 105   1  49  86]
  [ 85  53 104  65   8]
  [ 24  17  70  93 108]]

 [[ 32 119  95  11  58]
  [120  34  60  10  91]
  [ 75  68  55  99  15]
  [  3  92  59  88  71]
  [ 89   2  46 107  77]]

 [[ 28  35  87 125  39]
  [ 45 103  69  41  57]
  [ 30  36 110  20 122]
  [ 98  27  42 106  43]
  [114 115   5  23  54]]

 [[117  50   4  66  78]
  [ 18 102  19 109  67]
  [123  33  84  52  21]
  [ 38  64  96   7 111]
  [ 14  63 112  80  47]]

 [[113   9  13  61 121]
  [ 31  37 124 100  22]
  [  6  72  76  94  62]
  [ 90  79  16  48  83]
  [ 74 118  82  12  29]]]
```



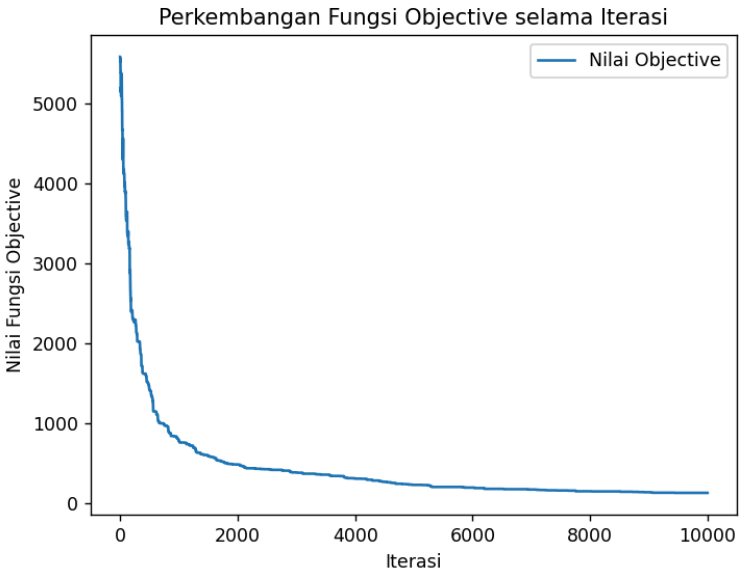


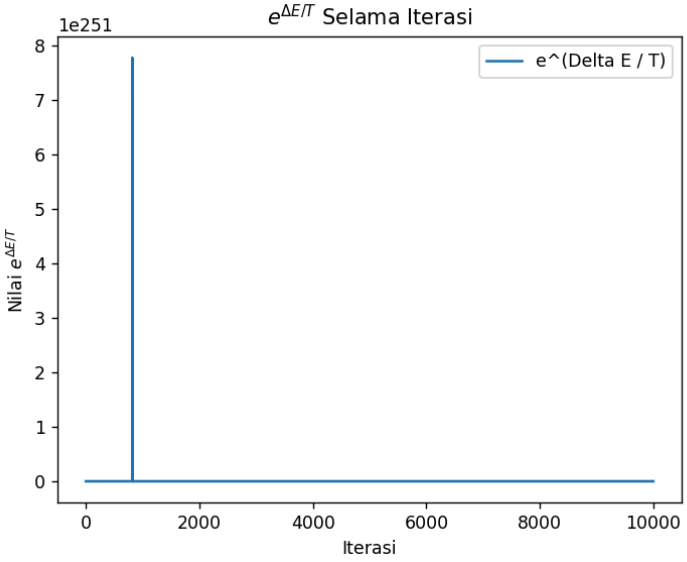

```
Jumlah Stuck: 9527
Durasi Total: 1.93 detik
Nilai Objective Terbaik yang Ditemukan: 168.0
```

Percobaan Algoritma ketiga:

```
State Awal Kubus:
[[[ 84  81  19  33  51]
  [ 30 109 110  48  47]
  [ 38  50  91   9 113]
  [ 29  17  52  78  13]
  [ 67 104 121  45  64]]

 [[  6   1  63  83  26]
  [ 87 101  90  97   8]
  [108  72  59 100  24]
  [ 89  39  35   2  70]
  [122   7 120  69   4]]

 [[ 36  23  61  21  34]
  [ 53  73 125  82  22]
  [ 42  80 106  56  31]
  [ 79  94  88  99  46]
  [103 105  66  49 123]]

 [[ 55  25  11  57  75]
  [ 40  85 115  20  16]
  [  3  43   5 102  58]
  [ 62  65 118  28  14]
  [ 15  96  77  10  44]]

 [[114  18  60  71 117]
  [ 68  98  93 107  32]
  [ 95 124  27  54 119]
  [111  12  86  41  37]
  [112 116  92  76  74]]]
```

```
State Akhir Kubus:
[[[ 82  99  52  68  13]
  [100  16  28  60 111]
  [119  30 103  44  19]
  [  8  96  45  92  75]
  [ 10  71  86  51  97]]

 [[  5   6 113  85 105]
  [ 41  95 108  57  17]
  [ 35  53  34  87 107]
  [115  66  26  22  77]
  [114  91  36  62  12]]

 [[ 65 117  58  42  33]
  [ 73   1 123   2 116]
  [ 49  93  55  79  39]
  [ 90  84  14 121   9]
  [ 40  20  67  72 118]]

 [[ 59  37  64  46 110]
  [ 21  69  50 106  70]
  [ 98  61  32 101  25]
  [ 76  47 120  43  29]
  [ 63 102  48  18  83]]

 [[104  56  24  74  54]
  [ 80 124   7  88  11]
  [ 15  78  94   4 125]
  [ 27  23 109  38 122]
  [ 89  31  81 112   3]]]
```



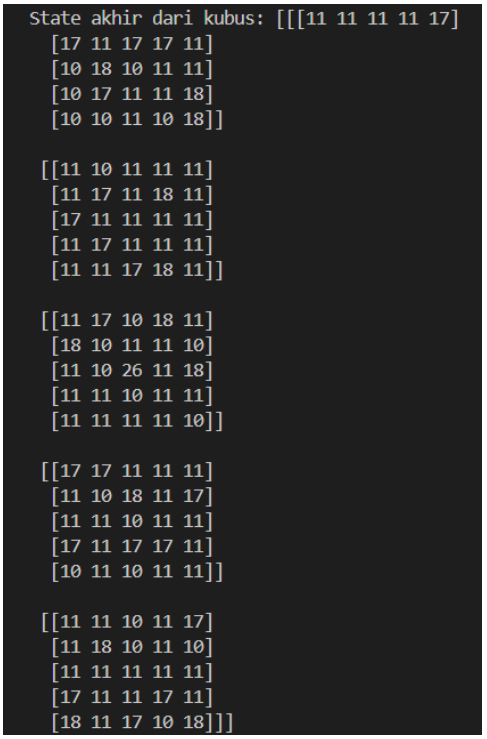



```
Jumlah Stuck: 9563
Durasi Total: 1.95 detik
Nilai Objective Terbaik yang Ditemukan: 124.0
```

Pencarian pertama menemukan *objective function* 136, dengan jumlah stuck 9542 dan durasi total 1.96 detik. Pencarian kedua menemukan *objective function* 168, dengan jumlah stuck 9527 dan durasi total 1.93 detik. Pencarian ketiga mendapatkan *objective function* 124 dengan jumlah stuck 9563 dan durasi total yang diperlukan adalah 1.95 detik.

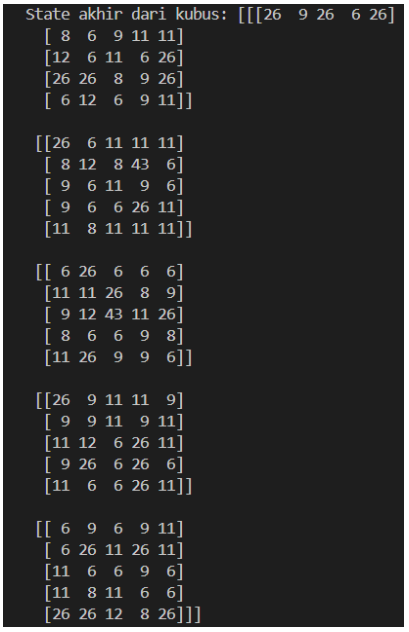
3.3. Eksperimen Genetic Algorithm

Pada percobaan ini, akan diubah 2 parameter pada algoritma. Tahap pertama adalah mengubah jumlah populasi dan membiarkan parameter yang lain tetap sama. jumlah populasi yang akan diuji adalah 25, 50, dan 75. Pada jumlah populasi 25 dan banyak iterasi 500, nilai objective function yang didapatkan adalah 515 dengan durasi 19,7 detik



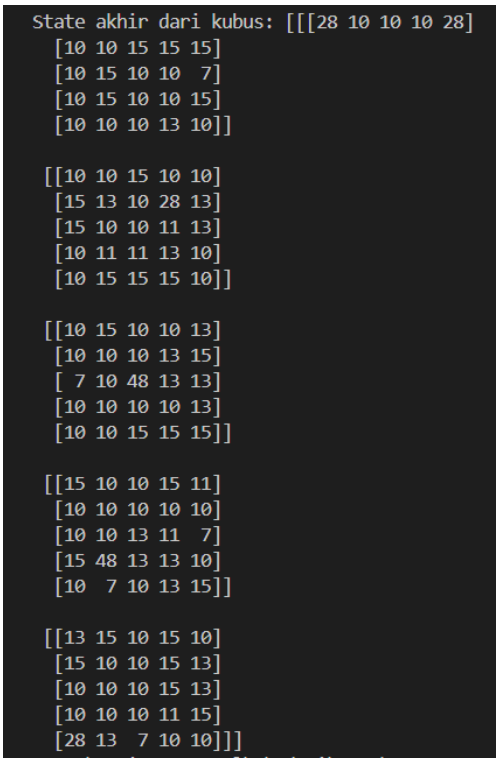
State Akhir dari percobaan 1 Variasi 1

untuk variasi berikutnya, masih menggunakan banyak iterasi 500 dengan jumlah populasi 50 mendapatkan nilai sebesar 482 dengan durasi sekitar 39 detik.



State Akhir dari percobaan 1 Variasi 2

Untuk Variasi terakhir pada percobaan 1, masih dengan banyak iterasi 500, jumlah populasi menjadi 75 mendapatkan nilai objective function sebesar 407 dengan durasi sekitar 56,25 detik

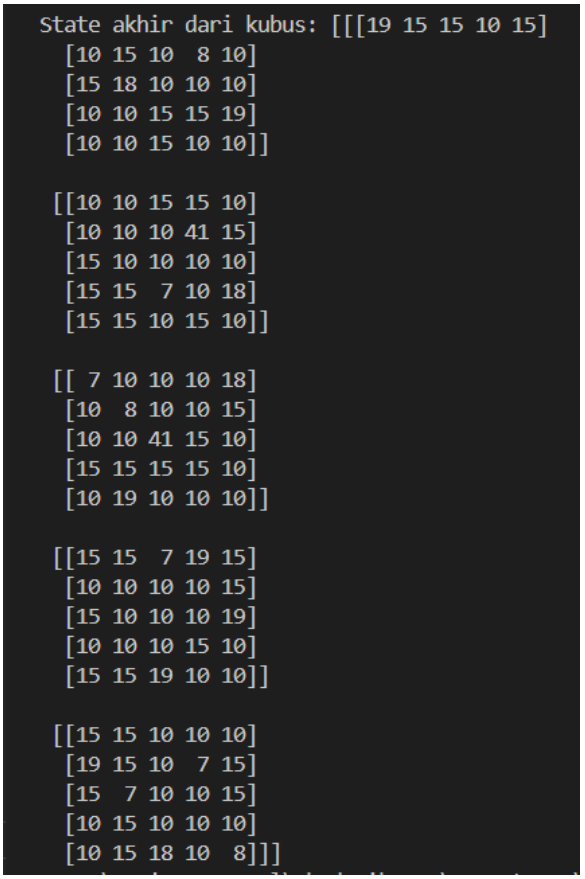


State Akhir dari percobaan 1 Variasi 3

berikut rangkuman data percobaan 1.

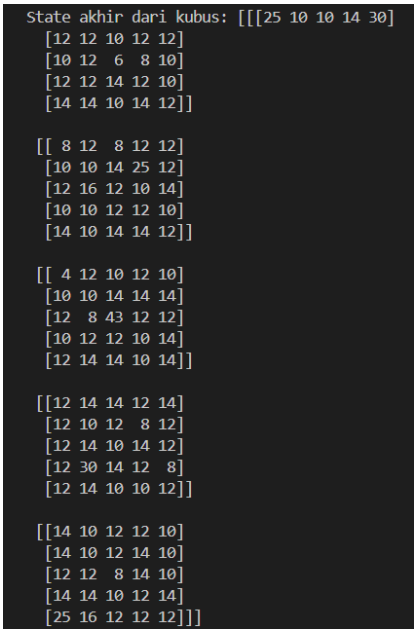
Variasi ke-	Jumlah Populasi	Banyak Iterasi	Nilai Objective Function	Durasi
1	25	500	515	19,7 detik
2	50	500	482	39 detik
3	75	500	407	56,25 detik

Untuk Percobaan 2, variasi akan dilakukan terhadap banyak iterasi dengan variasi antara 450, 350, 250 dan jumlah populasi konstan di 50. Pada variasi 1 dengan jumlah populasi 50 dan banyak iterasi 450 mendapatkan nilai objective function sebesar 492 dengan durasi sekitar 34,56 detik.



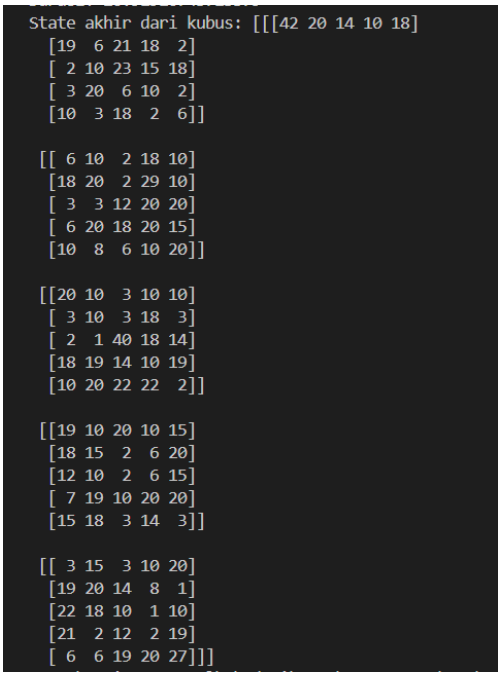
State Akhir dari percobaan 2 Variasi 1

Untuk variasi berikutnya, banyak iterasi diturunkan menjadi 350, nilai objective function yang didapatkan adalah 432 dengan durasi sekitar 26 detik.



State Akhir dari percobaan 2 Variasi 3

Untuk variasi berikutnya, banyak iterasi diturunkan menjadi 250, nilai objective function yang didapatkan adalah 490 dengan durasi 20,92 detik



State Akhir dari percobaan 2 Variasi 3

berikut rangkuman data dari percobaan 2

Variasi ke-	Jumlah Populasi	Banyak Iterasi	Nilai Objective Function	Durasi
1	50	450	492	34,56 detik
2	50	350	432	26 detik
3	50	250	490	20,92 detik

4. Analisis Hasil Eksperimen

1. Seberapa dekat tiap-tiap algoritma bisa mendekati global optima dan mengapa hasilnya demikian?

a. Steepest-Ascent Hill Climbing: Mendekati solusi optimal pada beberapa percobaan, tetapi sering kali berhenti di *local optima* karena sifatnya yang hanya memilih solusi terbaik dari *neighbors* terdekat dan tidak bisa berpindah dari *local minimum*. Steepest-Ascent Hill Climbing kurang efektif mendekati global optima

b. Simulated Annealing: Memiliki peluang lebih baik untuk mendekati global optimal karena memungkinkan pindah ke solusi yang kurang optimal pada proses awal untuk keluar dari *local minima*. Seiring dengan penurunan suhu, algoritma semakin selektif terhadap solusi yang memungkinkan pencarian menjadi lebih luas. Dengan konfigurasi suhu dan *cooling* yang tepat, Simulated Annealing lebih mampu mendekati global optima daripada Steepest-Ascent Hill Climbing.
c. Genetic Algorithm: Lebih fleksibel dalam eksplorasi ruang pencarian karena metode *selection, crossover,* dan *mutation* membuat keberagaman solusi yang tinggi. Hal itu membuat algoritma menghindari *local optima* dan memungkinkan mendekati global optima yang lebih baik dengan pengaturan *population size* dan jumlah iterasi yang tepat.

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

a. Steepest-Ascent Hill Climbing: Sering kali berhenti pada *local optimum* jika tidak ada jalur untuk memberikan solusi yang lebih baik. Hasil akhir bergantung pada solusi awal dan dapat terjebak pada hasil yang lebih jauh dari optimal jika berada di *local minima*.
b. Simulated Annealing: Lebih unggul dalam menghindari local minima karena penerimaan solusi yang kurang optimal di awal atau pada saat suhu tinggi. Hal ini membantu algoritma untuk menjelajahi ruang pencarian lebih luas.
c. Genetic Algorithm: Menghasilkan solusi yang lebih baik dengan eksplorasi ruang pencarian yang efektif, menjadikan algoritma ini lebih efektif dalam mendekati global optimum

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

a. Steepest-Ascent Hill Climbing: Durasi dari algoritma ini cenderung jauh lebih cepat dibandingkan kedua algoritma lainnya. Hal ini disebabkan karena *local search* ini tidak banyak melakukan eksplorasi, pencarian dilakukan secara lokal dan iterasi langsung berhenti apabila tidak ditemukan tetangga yang memiliki *objective function* yang lebih baik.
b. Simulated Annealing:Apabila dibandingkan dengan Steepest-Ascent Hill Climbing, maka algoritma ini berjalan lebih lambat. Hal ini disebabkan karena Simulated Annealing lebih melakukan eksplorasi, memungkinkan perpindahan sesekali ke solusi yang lebih buruk untuk lepas dari solusi optima lokal dan, membutuhkan pengurangan parameter suhu secara bertahap.
c. Genetic Algorithm: Algoritma ini memiliki durasi pencarian jauh lebih lama dibandingkan dengan algoritma-algoritma lain. Genetic algorithm menggunakan populasi solusi kandidat dan menerapkan operasi seperti mutasi, crossover, dan seleksi untuk mengembangkan solusi yang lebih baik. Karena jauh lebih kompleks dan pengembangannya melibatkan seluruh populasi solusi, maka durasi yang dibutuhkan jauh lebih panjang.

4. Seberapa konsisten hasil akhir yang didapatkan dari tiap-tiap eksperimen yang dilakukan?

Hasil yang didapatkan setiap kali dilakukan pencarian dapat berubah-ubah karena keadaan kubus yang berbeda tiap kali melakukan inisialisasi *(*angka disusun secara random). Tetapi, rata-rata memiliki hasil yang tidak terlalu berbeda jauh antara satu dengan lainnya, baik itu dari segi waktu pencarian hingga hasil *objective function* yang didapatkan.

Bisa dilihat kembali dari Steepest-ascent hill climbing, bahwa *objective function* cukup beragam bergantung pada jumlah interasi yang dilakukan. Hal ini berpengaruh pada keadaan atau *state* kubus pada awal mulanya dan proses pencarian yang dihentikan apabila tidak menemukan *neighbor* yang lebih bagus. Hal ini menyebabkan hasilnya lebih bervariasi dibandingkan dengan *local search lain.* Tetapi untuk algoritma lain, hasil yang didapatkan rata-rata konsisten dan tidak banyak berubah.

5. Bagaimana pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada Genetic Algorithma

a. Banyak Iterasi

   Semakin banyak iterasi yang dijalankan, Genetic Algorithm mempunyai kesempatan yang lebih besar dalam mendekati global optimum. Proses *selection, crossover,* dan *mutation* dapat berjalan lebih lama agar populasi bisa berevolusi untuk memperbaiki nilai *objective function*. Namun, peningkatan banyak iterasi mengakibatkan peningkatan durasi pencarian.

b. Jumlah populasi

   Ukuran populasi memengaruhi variasi solusi dalam pencarian. Peningkatan jumlah populasi dari 25 ke 75 secara signifikan telah menurunkan nilai *objective function*, yang menunjukkan solusi mendekati optimal. Populasi yang lebih besar meningkatkan keberagaman genetic dalam populasi, sehingga mengurangi risiko terjebak pada *local optima*. Peningkatan jumlah populasi juga mengakibatkan peningkatan durasi pencarian.

III. Kesimpulan

Algoritma *local search* Steepest-Ascent Hill Climbing memiliki waktu pencarian yang jauh lebih cepat, tetapi iterasi yang dilakukan lebih sedikit dan hasilnya sering terjebak pada *local optima* karena hanya melibatkan pencarian lokal dan tidak melakukan eksplorasi. Iterasi langsung dihentikan apabila tidak ada lagi tetangga yang lebih baik / flat. Oleh karena itu, hasil pencariannya kadang tidak konsisten bergantung pada *initial state* dari suatu permasalahan.

Algoritma Simulated Annealing memiliki waktu pencarian yang standar, dan memiliki kemungkinan lebih tinggi untuk keluar dari local optima dan bisa memilih tetangga yang lebih buruk dari *current state.* Algoritma ini memiliki ruang penjelajahan yang lebih luas.

Algoritma Genetic Algorithm jauh lebih kompleks dibandingkan yang lain, karena mempengaruhi populasi suatu solusi dan tidak hanya satu kandidat. Oleh karena itu, waktu yang dibutuhkan untuk mencari solusi lebih lama. Namun, algoritma ini lebih mungkin mendapatkan solusi global optima karena ruang pencarian yang luas.

Saran yang digunakan adalah penggunaan local search masing-masing dapat disesuaikan dengan permasalahan yang didapatkan. Untuk kubus ini, algoritma yang cocok adalah simulated annealing, karena mampu mengatasi jebakan lokal optima dan waktu yang dibutuhkan tidak terlalu banyak.

IV. Pembagian Tugas

Nama	**NIM**	**Pembagian Tugas**
Regina Deva Carissa	18222040	1. Membuat source code simulated annealing dan source code Steepest Ascent Hill Climbing 2. Menjawab pertanyaan analisis 3. Mengisi deskripsi fungsi simulated annealing dan source code Steepest Ascent Hill Climbing 4. Merapikan dokumen
Kezia Caren Cahyadi	18222041	1. Membuat dan mengimplementasikan source code Genetic Algorithm 2. Menjawab pertanyaan analisis 3. Mengisi deskripsi fungsi Genetic Algorithm 4. Merapikan dokumen

Nama	NIM	Pembagian Tugas
Ananda Farhan Raihandra	18222084	1. Merapihkan template dokumen 2. Melakukan Eksperimen Steepest Ascent Hill-Climbing 3. Melakukan Eksperimen Genetic Algorithm 4. Menambah fungsional durasi pada source code Genetic Algorithm 5. Menambahkan sedikit source code untuk Simulated Annealing

V. Referensi

Features of the magic cube - Magisch vierkant
Perfect Magic Cubes (trump.de)