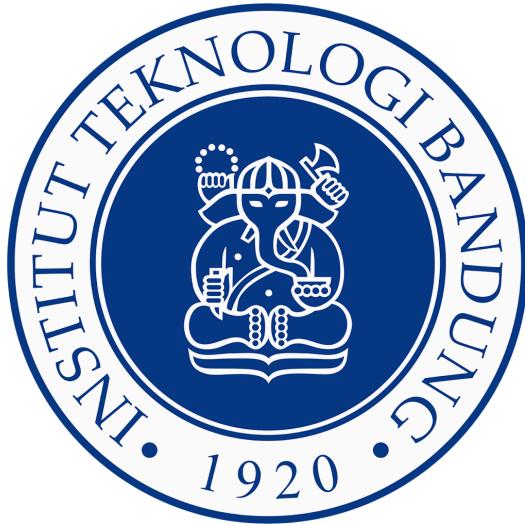


LAPORAN TUGAS BESAR 1
IF3070 Dasar Inteligensi Artifisial

“Pencarian Solusi Diagonal Magic Cube dengan Local Search”



Dosen:

Dr. Fariska Zakhralativa Ruskanda, S.T., M.T.

Oleh:

Owen Tobias Sinurat (13522131)

Ahmad Thoriq Saputra (13522141)

Muhammad Fatihul Irhab (13522143)

Axel Santadi Warih (13522155)

PROGRAM STUDI TEKNIK INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
SEMESTER II TAHUN 2023/2024

BAB I

DESKRIPSI PERSOALAN

Diagonal Magic Cube adalah kubus yang tersusun dari angka-angka yang unik, mulai dari 1 hingga n^3 , di mana n adalah panjang sisi kubus tersebut. Pada persoalan ini, ukuran kubus yang akan digunakan adalah $5 \times 5 \times 5$, yang artinya terdapat total 125 angka yang diatur dalam susunan tiga dimensi. Magic Cube memiliki properti yang khusus, di mana terdapat sebuah angka yang disebut dengan magic number. Magic number ini tidak harus termasuk dalam rentang angka yang tersusun di dalam kubus, namun merupakan nilai yang dihasilkan dari jumlah angka-angka yang ada di dalam kubus berdasarkan aturan tertentu.

64	18	37	11	
20	9	62	39	11
45	56	3	26	39
1	47	28	54	26
54	26	39	46	
1	47	28	54	54
40	58	13	19	19
30	4	55	41	53
59	21	34	16	41
16	36	29	49	

Terdapat beberapa karakteristik utama yang harus dipenuhi oleh sebuah Diagonal Magic Cube:

1. **Jumlah setiap baris:** Jumlah semua angka di setiap baris pada kubus harus sama dengan magic number.
2. **Jumlah setiap kolom:** Sama halnya dengan baris, jumlah semua angka di setiap kolom juga harus sama dengan magic number.
3. **Jumlah setiap tiang:** Tiang dalam konteks kubus ini adalah susunan angka yang memanjang secara vertikal di sepanjang kubus, dan jumlah angka pada setiap tiang juga harus sama dengan magic number.
4. **Jumlah diagonal ruang:** Kubus memiliki beberapa diagonal ruang, yang merupakan garis diagonal yang terbentuk dari sudut ke sudut yang melintasi kubus tiga dimensi. Jumlah angka di sepanjang semua diagonal ruang ini harus sama dengan magic number.
5. **Jumlah diagonal bidang:** Kubus $5 \times 5 \times 5$ juga memiliki beberapa bidang dua dimensi yang bisa dipotong secara horizontal, vertikal, dan diagonal. Jumlah angka di sepanjang semua diagonal dalam bidang-bidang ini juga harus sama dengan magic number.

Tujuan dari persoalan ini adalah untuk menemukan susunan angka dalam kubus sedemikian rupa sehingga semua kondisi di atas terpenuhi, menggunakan algoritma local search. Kubus awal dimulai dengan angka 1 hingga 125 yang disusun secara acak. Setiap langkah dalam pencarian solusi dilakukan dengan menukar posisi dua angka pada kubus tersebut. Dengan

menggunakan berbagai algoritma local search, solusi optimal yang memenuhi semua aturan di atas harus dicari.

Proses pencarian solusi dilakukan dengan pendekatan local search, yang mencakup beberapa metode seperti:

- **Steepest Ascent Hill-climbing:** Algoritma ini memilih langkah yang memberikan peningkatan terbesar terhadap solusi saat ini, berusaha untuk mencapai solusi optimal dengan menaiki "bukit" nilai fungsi objektif.
- **Hill-climbing with Sideways Move:** Sama seperti hill-climbing biasa, tetapi mengizinkan pergerakan ke arah yang tidak meningkatkan nilai fungsi objektif, untuk menghindari macet di puncak lokal.
- **Random Restart Hill-climbing:** Setelah setiap kali terjebak di puncak lokal, algoritma ini memulai kembali dari posisi acak baru, dengan harapan menemukan solusi global optimal.
- **Stochastic Hill-climbing:** Algoritma ini memilih langkah secara acak di antara peningkatan yang tersedia, sehingga lebih fleksibel dibandingkan steepest ascent.
- **Simulated Annealing:** Algoritma ini meniru proses pendinginan dalam metalurgi, di mana probabilitas menerima langkah buruk menurun seiring waktu, memungkinkan eksplorasi solusi yang lebih beragam.
- **Genetic Algorithm:** Pendekatan ini menggunakan seleksi alam dan operasi genetik seperti crossover dan mutasi untuk menghasilkan solusi baru dari populasi solusi.

Dengan menggunakan metode-metode tersebut, diharapkan bisa ditemukan solusi optimal untuk susunan Diagonal Magic Cube berukuran 5x5x5 yang memenuhi seluruh kondisi di atas.

BAB II

PEMBAHASAN

2.1 Pemilihan Objective Function

Magic number dalam konteks magic cube adalah nilai yang harus dicapai oleh setiap baris, kolom, dan diagonal agar kubus tersebut dapat dianggap sebagai magic cube. Untuk cube berukuran n , magic number ini dihitung dengan rumus:

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

Magic number untuk Diagonal Magic Cube berukuran 5x5x5 adalah 315. Artinya, untuk dapat disebut sebagai magic cube, setiap baris, kolom, dan diagonal di dalam cube tersebut harus memiliki jumlah angka yang sama dengan 315.

Objective function adalah fungsi yang digunakan untuk mengevaluasi kualitas dari konfigurasi yang ada. Dalam konteks diagonal magic cube, objective function bertujuan untuk meminimalkan selisih antara jumlah setiap baris, kolom, dan diagonal dengan magic number. Untuk cube berukuran n , objective function dapat dihitung dengan rumus:

$$f(state) = \sum_{i=1}^n |jumlah_angka(i) - M|$$

Penjelasan:

- M = magic number dari magic cube tersebut
- $jumlah_angka(i)$ = jumlah angka dari baris, kolom, tiang, atau diagonal ke- i pada state tertentu dari cube tersebut.

Alasan:

Rumus $f(state)$ tersebut dirancang untuk mengevaluasi seberapa baik konfigurasi dari sebuah magic cube mendekati solusi yang optimal. Dengan mengukur perbedaan antara M (magic number) dan jumlah setiap baris, kolom, tiang atau diagonal dalam suatu konfigurasi, kita bisa mengetahui seberapa jauh nilai-nilai tersebut dari solusi sempurna (magic cube yang benar).

```
● ● ●

def fitness(cube):
    score = 0

    for i in range(N):
        score += abs(np.sum(cube[i, :, :]) - MAGIC_CONSTANT*N)
        score += abs(np.sum(cube[:, i, :]) - MAGIC_CONSTANT*N)
        score += abs(np.sum(cube[:, :, i]) - MAGIC_CONSTANT*N)

    for i in range(N):
        score += abs(np.sum(cube[i, :, :].diagonal()) - MAGIC_CONSTANT)
        score += abs(np.sum(np.fliplr(cube[i, :, :]).diagonal()) - MAGIC_CONSTANT)
        score += abs(np.sum(cube[:, i, :].diagonal()) - MAGIC_CONSTANT)
        score += abs(np.sum(np.fliplr(cube[:, i, :]).diagonal()) - MAGIC_CONSTANT)
        score += abs(np.sum(cube[:, :, i].diagonal()) - MAGIC_CONSTANT)
        score += abs(np.sum(np.fliplr(cube[:, :, i]).diagonal()) - MAGIC_CONSTANT)

    score += abs(np.sum([cube[i, i, i] for i in range(N)]) - MAGIC_CONSTANT)
    score += abs(np.sum([cube[i, i, N - i - 1] for i in range(N)]) - MAGIC_CONSTANT)
    return score
```

Fungsi fitness(cube) : fungsi akan menerima input yang berupa array 5x5x5 yang sudah berisikan konfigurasi awal dari state. Lalu fungsi ini akan menghitung nilai objective function berdasarkan dengan rumus yang telah dijelaskan diatas, lalu mengembalikan nilai objective function tersebut.

2.2 Penjelasan Implementasi Algoritma Local Search

2.2.1 Hill Climb

2.2.1.1 Steepest Ascent

Algoritma steepest ascent hill climbing digunakan untuk menemukan solusi optimal dengan mengevaluasi semua state tetangga dari posisi saat ini dan memilih yang memiliki nilai heuristic terbaik (dalam hal ini, nilai fitness terendah). Prosesnya berulang hingga tidak ada perbaikan lebih lanjut pada state saat ini. Jika semua tetangga lebih buruk atau sama, algoritma berhenti dan mengembalikan hasilnya.

```
● ● ●

import numpy as np
from itertools import product
from cube import fitness
import time

#menghasilkan semua kemungkinan state tetangga dengan menukar 2 angka
def generate_neighbors(cube):
    neighbors = []

    N = len(cube) # Assuming cube is a 3D numpy array with shape (N, N, N)

    indices = np.array(list(product(range(N), repeat=3)))

    for idx in range(len(indices)):
        i, j, k = indices[idx]
        for di, dj, dk in product([-1, 0, 1], repeat=3):
            if di == dj == dk == 0:
                continue
            ni, nj, nk = i + di, j + dj, k + dk
            if 0 <= ni < N and 0 <= nj < N and 0 <= nk < N:
                new_cube = cube.copy()
                new_cube[i, j, k], new_cube[ni, nj, nk] = new_cube[ni, nj, nk], new_cube[i, j, k]
                neighbors.append(new_cube)

    return neighbors

# Hill Climbing Algorithm
def steepest_ascent_hill_climbing(cube):
    current_state = cube
    current_heuristic = fitness(current_state)
    iterations = 1 # Track the number of iterations
    fitnesses = []

    start = time.time()
    while True:
        neighbors = generate_neighbors(current_state)
        best_neighbor = None
        best_heuristic = current_heuristic

        # Find the neighbor with the highest heuristic score
        for neighbor in neighbors:
            neighbor_heuristic = fitness(neighbor)
            if neighbor_heuristic < best_heuristic:
                best_heuristic = neighbor_heuristic
                best_neighbor = neighbor

        # If no better neighbor is found, terminate
        if best_neighbor is None:
            break

        # Move to the best neighbor
        current_state = best_neighbor
        current_heuristic = best_heuristic
        print(iterations, current_heuristic)
        fitnesses.append(current_heuristic)
        iterations += 1

    endtime = time.time()
    time_taken = endtime - start
    return current_state, current_heuristic, fitnesses, time_taken, iterations
```

Fungsi generate_neighbors(cube): Fungsi ini digunakan untuk menghasilkan semua kemungkinan state tetangga dari sebuah cube dengan menukar dua angka yang berdekatan.

Fungsi steepest_ascent_hill_climbing(cube): Implementasi utama dari algoritma steepest ascent hill climbing. Algoritma ini mengevaluasi state awal dan mencari tetangga dengan heuristic terbaik (dalam konteks ini, fitness terendah) untuk menemukan solusi optimal.

2.2.1.2 Sideways Move

Algoritma *sideways move hill climbing* adalah variasi dari algoritma *hill climbing* yang memungkinkan pergerakan lateral jika tidak ditemukan perbaikan pada nilai *heuristic*. Dalam implementasi ini, algoritma dapat melakukan beberapa langkah lateral yang ditentukan oleh `max_side_moves` sebelum berhenti. Algoritma ini membantu menghindari jebakan pada *plateau* dalam ruang pencarian solusi.

```
●●●

import numpy as np
from itertools import product
from concurrent.futures import ThreadPoolExecutor
from cube import fitness
import time

# menghasilkan semua kemungkinan state tetangga dengan menukar 2 angka
def generate_neighbors(cube):
    neighbors = []
    N = len(cube) # Assuming cube is a 3D numpy array with shape (N, N, N)

    indices = np.array(list(product(range(N), repeat=3)))

    def swap_elements(idx):
        i, j, k = indices[idx]
        local_neighbors = []
        for di, dj, dk in product([-1, 0, 1], repeat=3):
            if di == dj == dk == 0:
                continue
            ni, nj, nk = i + di, j + dj, k + dk
            if 0 <= ni < N and 0 <= nj < N and 0 <= nk < N:
                new_cube = cube.copy()
                new_cube[i, j, k], new_cube[ni, nj, nk] = new_cube[ni, nj, nk], new_cube[i, j, k]
                local_neighbors.append(new_cube)
        return local_neighbors

    with ThreadPoolExecutor() as executor:
        results = executor.map(swap_elements, range(len(indices)))

    for result in results:
        neighbors.extend(result)

    return neighbors

# Hill Climbing Algorithm
def sideways_move_hill_climbing(cube, max_side_moves):
    current_state = cube
    current_heuristic = fitness(current_state)
    iterations = 1 # Inisialisasi variabel iterations
    no_improvement_count = 0 # Counter untuk melacak iterasi tanpa perbaikan
    max_no_improvement = max_side_moves # Batas iterasi tanpa perbaikan
    fitnesses = []

    start = time.time()
    while True:
        neighbors = generate_neighbors(current_state)
        best_neighbor = None
        best_heuristic = current_heuristic

        # Find the neighbor with the highest heuristic score
        for neighbor in neighbors:
            neighbor_heuristic = fitness(neighbor)
            if neighbor_heuristic <= best_heuristic:
                best_heuristic = neighbor_heuristic
                best_neighbor = neighbor

        # If no better neighbor is found, increment no_improvement_count
        if best_neighbor is None or best_heuristic == current_heuristic:
            no_improvement_count += 1
        else:
            no_improvement_count = 0

        # If no improvement for max_no_improvement iterations, terminate
        if no_improvement_count >= max_no_improvement:
            break

        # Move to the best neighbor
        if best_neighbor is not None:
            current_state = best_neighbor
            current_heuristic = best_heuristic

        print(iterations, current_heuristic)
        fitnesses.append(current_heuristic)
        iterations += 1

    end = time.time()
    time_taken = end - start
    return current_state, current_heuristic, fitnesses, time_taken, iterations
```

Fungsi generate_neighbor(cube): Fungsi ini digunakan untuk menghasilkan semua kemungkinan *state* tetangga dari sebuah *cube* dengan menukar dua elemen yang berdekatan. Pendekatan ini menggunakan *multithreading* untuk mempercepat proses pembentukan tetangga.

Fungsi sideways_move_hill_climbing(cube, max_side_moves): Fungsi ini mengimplementasikan algoritma *hill climbing* dengan langkah samping (*sideways move*), yang mengizinkan pindah ke *state* dengan *heuristic* yang sama hingga batas tertentu (diatur dengan *max_side_moves*).

2.2.1.3 Random Restart

Algoritma *random restart hill climbing* adalah varian dari algoritma *hill climbing* yang mencoba mengatasi jebakan pada *local minima* dengan melakukan beberapa pengulangan dari posisi awal yang berbeda. Jika satu proses *hill climbing* berakhir tanpa menemukan solusi optimal global, algoritma akan memulai kembali dari *state* acak lainnya hingga jumlah percobaan yang ditentukan (*restart_limit*) tercapai. Hal ini meningkatkan peluang menemukan solusi yang lebih baik dibandingkan hanya melakukan satu pencarian lokal.

```
●●●

def generate_neighbors(cube):
    neighbors = []

    N = len(cube) # Assuming cube is a 3D numpy array with shape (N, N, N)
    indices = np.array(list(product(range(N), repeat=3)))

    for idx in range(len(indices)):
        i, j, k = indices[idx]
        for di, dj, dk in product([-1, 0, 1], repeat=3):
            if di == dj == dk == 0:
                continue
            ni, nj, nk = i + di, j + dj, k + dk
            if 0 <= ni < N and 0 <= nj < N and 0 <= nk < N:
                new_cube = cube.copy()
                new_cube[i, j, k], new_cube[ni, nj, nk] = new_cube[ni, nj, nk], new_cube[i, j, k]
                neighbors.append(new_cube)

    return neighbors

# Hill Climbing Algorithm
def random_restart_hill_climbing(restart_limit, cube):
    optimal_states = []
    iterations = [] # iterasi per restart
    wholeIterations = 1 # iterasi total dari algoritma
    fitnesses = [] # fitness total dari algoritma

    for _ in range(restart_limit):
        current_heuristic = fitness(cube)
        iteration = 1 # Track the number of iterations

        start = time.time()
        while True:
            #mulai time
            neighbors = generate_neighbors(current_state)
            best_neighbor = None
            best_heuristic = current_heuristic

            # Find the neighbor with the highest heuristic score
            for neighbor in neighbors:
                neighbor_heuristic = fitness(neighbor)
                if neighbor_heuristic < best_heuristic:
                    best_heuristic = neighbor_heuristic
                    best_neighbor = neighbor

            # If no better neighbor is found, terminate
            if best_neighbor is None:
                optimal_states.append(current_state)
                iterations.append(iteration)
                break

            # Move to the best neighbor
            current_state = best_neighbor
            current_heuristic = best_heuristic
            print(iteration, current_heuristic)
            fitnesses.append(current_heuristic)
            iteration += 1
            wholeIterations += 1

        #mencari state dengan heuristic terendah untuk di return
        best_state = min(optimal_states, key=fitness)
        end = time.time()
        time_taken = end - start
        #return state akhir, objective function yang dicapai, trus fitness sama wholeiterations buat bikin plot, durasi, banyak restart, sama iterasi per restart
        return best_state, current_heuristic, fitnesses, wholeIterations, time_taken, restart_limit, iterations
```

Fungsi generate_neighbors(cube): Fungsi ini bertanggung jawab untuk menghasilkan semua kemungkinan *state* tetangga dari kubus yang diberikan dengan menukar dua elemen di posisi yang berdekatan.

Fungsi random_restart_hill_climbing(restart_limit, cube): Fungsi ini mengimplementasikan algoritma *random restart hill climbing* yang melakukan beberapa restart dari *state* awal yang berbeda untuk menghindari *local optimum*.

2.2.1.4 Stochastic

Algoritma *stochastic hill climbing* adalah varian dari *hill climbing* yang memilih tetangga secara acak dari semua kemungkinan. Ini membantu algoritma menghindari jebakan pada *local maxima* atau *plateau* karena tidak sepenuhnya bergantung pada pilihan terbaik di setiap langkah. Algoritma ini melanjutkan pencarian hingga sejumlah upaya tanpa perbaikan mencapai batas tertentu (`max_attempts`), setelah itu ia berhenti jika tidak ada perbaikan lebih lanjut.

```
● ● ●

def generate_neighbors(cube):
    neighbors = []
    N = len(cube)

    indices = np.array(list(product(range(N), repeat=3)))

    for idx in range(len(indices)):
        i, j, k = indices[idx]
        for di, dj, dk in product([-1, 0, 1], repeat=3):
            if di == dj == dk == 0:
                continue
            ni, nj, nk = i + di, j + dj, k + dk
            if 0 <= ni < N and 0 <= nj < N and 0 <= nk < N:
                new_cube = cube.copy()
                new_cube[i, j, k], new_cube[ni, nj, nk] = new_cube[ni, nj, nk], new_cube[i, j, k]
                neighbors.append(new_cube)

    return neighbors

# Hill Climbing Algorithm
def stochastic_hill_climbing(cube):
    current_state = cube
    current_heuristic = fitness(current_state)
    iterations = 0 # Inisialisasi variabel iterations
    max_attempts = 100 # Batas maksimal pemilihan acak tanpa perbaikan
    attempts = 0 # Counter untuk melacak jumlah pemilihan acak tanpa perbaikan
    fitnesses = []

    start = time.time()
    while True:
        neighbors = generate_neighbors(current_state)

        # Ambil satu tetangga secara acak
        random_neighbor = random.choice(neighbors)
        random_neighbor_heuristic = fitness(random_neighbor)

        # Jika fitness tetangga lebih besar atau sama dengan fitness saat ini, ulangi pengambilan tetangga acak
        if random_neighbor_heuristic >= current_heuristic:
            attempts += 1
            if attempts >= max_attempts:
                break
            continue

        # Move to the random neighbor
        current_state = random_neighbor
        current_heuristic = random_neighbor_heuristic
        print(iterations, current_heuristic)
        fitnesses.append(current_heuristic)
        iterations += 1
        attempts = 0 # Reset counter jika ada perbaikan

    end = time.time()
    time_taken = end - start
    return current_state, current_heuristic, fitnesses, time_taken, iterations
```

Fungsi generate_neighbors(cube): Membuat semua kemungkinan *state* tetangga dari kubus yang diberikan dengan menukar dua elemen di posisi yang berdekatan.

Fungsi stochastic_hill_climbing(cube): Menerapkan algoritma *stochastic hill climbing* dengan memilih tetangga secara acak dan membandingkan *heuristic*-nya.

2.2.2 Simulated Annealing

Simulated annealing merupakan metode pencarian lokal yang menyerupai pencarian stochastic hill climbing, namun memiliki fleksibilitas untuk menerima solusi tetangga yang lebih buruk daripada solusi saat ini. Dalam proses ini, algoritma membuat successor dengan cara menukar dua buah posisi secara acak dan, pada kondisi tertentu dapat berpindah ke solusi yang kurang optimal untuk menghindari perangkap lokal maxima. Probabilitas menerima solusi yang lebih buruk dihitung menggunakan fungsi $e^{\Delta E/T}$, di mana ΔE menunjukkan selisih nilai objektif antara solusi suksesor dan solusi saat ini, dan T adalah suhu yang diatur oleh fungsi Schedule. Semakin rendah nilai T, semakin kecil kemungkinan untuk menerima solusi yang lebih buruk, yang mencerminkan proses pendinginan bertahap. Nilai hasil dari fungsi ini akan dibandingkan dengan bilangan konstan yang ditentukan. Jika hasil perhitungan lebih besar, neighbor terpilih akan menjadi solusi baru. Pendekatan ini memungkinkan eksplorasi solusi yang lebih luas, sehingga meningkatkan peluang menemukan solusi global maxima.

```
● ● ●

def createNewSolution(newSolution):
    i1, j1, k1, i2, j2, k2 = random.choices(range(5), k=6)
    newSolution[i1][j1][k1], newSolution[i2][j2][k2] = newSolution[i2][j2][k2], newSolution[i1][j1][k1]
    return newSolution

def simulatedAnnealing(n, initialTemperature, coolingRate, threshold):
    currentSolution = initialize_cube()
    currentScore = fitness(currentSolution)
    temperature = initialTemperature
    sumStuck = 0

    while temperature > 1:
        if currentScore == 0:
            break

        newSolution = np.copy(currentSolution)
        newSolution = createNewSolution(newSolution)
        newScore = fitness(newSolution)

        if newScore < currentScore:
            currentSolution = newSolution
            currentScore = newScore
        else:
            acceptanceProbability = np.exp((newScore - currentScore)*(-1) / temperature)
            temp = random.random()
            if threshold < acceptanceProbability:
                currentSolution = newSolution
                currentScore = newScore
            else:
                sumStuck += 1

        temperature *= coolingRate

    return currentSolution, currentScore, sumStuck
```

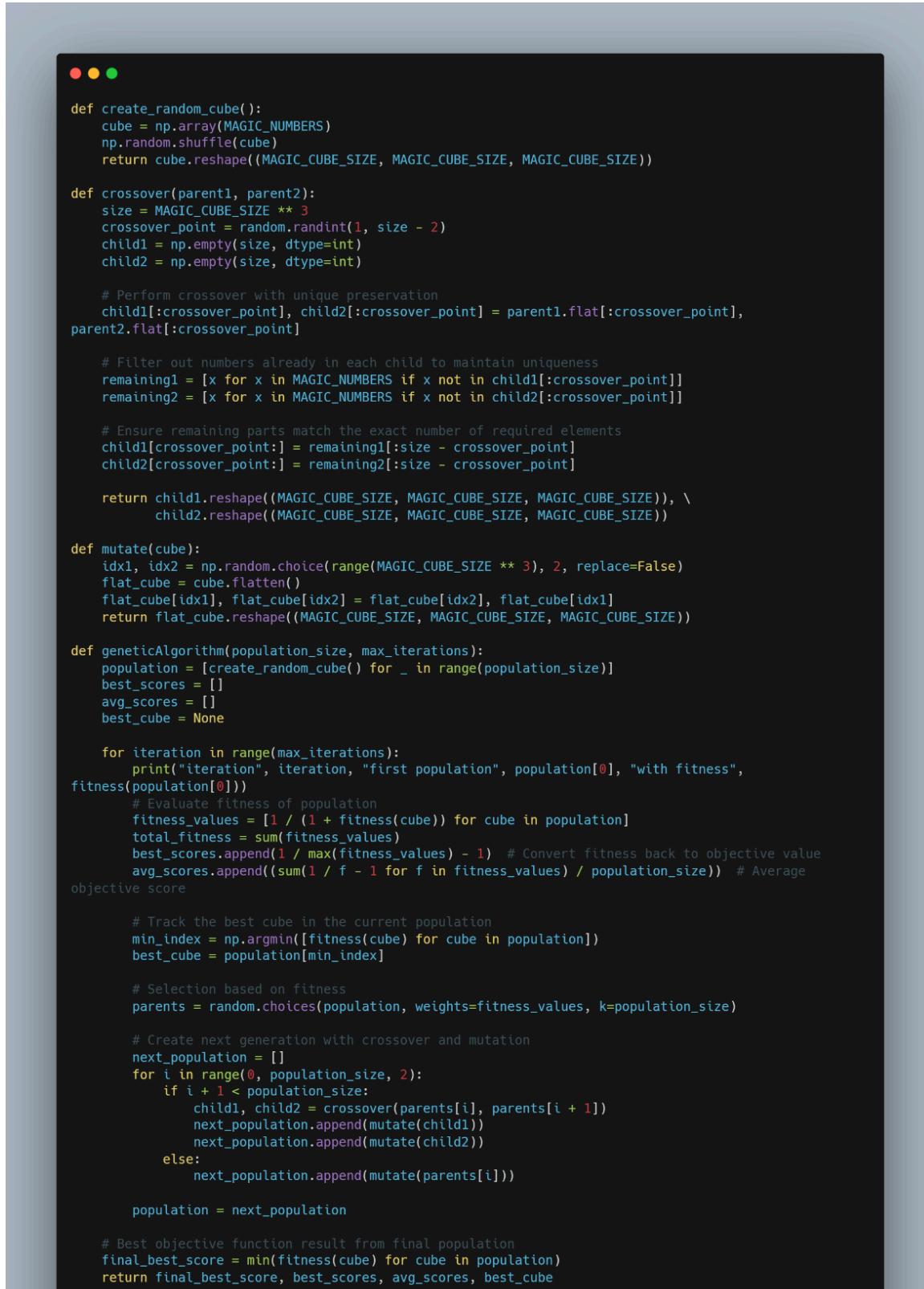
Fungsi *createNewSolution()*, fungsi untuk membuat konfigurasi baru untuk magic cube tersebut dengan menukar dua buah posisi secara acak.

Fungsi *initialize_cube()*, fungsi untuk membuat konfigurasi awal state secara acak yang berisikan angka unik dari 1-125.

Fungsi *fitness()*, fungsi untuk menghitung nilai objective function nya pada suatu state tertentu berdasarkan konfigurasi saat itu.

Fungsi *simulatedAnnealing()*, fungsi utama algoritma dengan menggunakan metode simulated annealing, dimana proses akan selalu melakukan iterasi selama nilai dari fitness terhadap konfigurasi tersebut tidak sama dengan 0 (global maxima). Di setiap iterasinya fungsi ini menggunakan *createNewSolution()* untuk menghasilkan konfigurasi baru dan fungsi *fitness()* untuk menghitung nilai objective function nya. Fungsi ini juga menghitung acceptance probability yang nantinya akan dibandingkan dengan nilai threshold untuk mengecek apakah konfigurasi yang kurang optima tetap dapat dipakai atau tidak.

2.2.3 Genetic Algorithm



```
def create_random_cube():
    cube = np.array(MAGIC_NUMBERS)
    np.random.shuffle(cube)
    return cube.reshape((MAGIC_CUBE_SIZE, MAGIC_CUBE_SIZE, MAGIC_CUBE_SIZE))

def crossover(parent1, parent2):
    size = MAGIC_CUBE_SIZE ** 3
    crossover_point = random.randint(1, size - 2)
    child1 = np.empty(size, dtype=int)
    child2 = np.empty(size, dtype=int)

    # Perform crossover with unique preservation
    child1[:crossover_point], child2[:crossover_point] = parent1.flat[:crossover_point],
    parent2.flat[:crossover_point]

    # Filter out numbers already in each child to maintain uniqueness
    remaining1 = [x for x in MAGIC_NUMBERS if x not in child1[:crossover_point]]
    remaining2 = [x for x in MAGIC_NUMBERS if x not in child2[:crossover_point]]

    # Ensure remaining parts match the exact number of required elements
    child1[crossover_point:] = remaining1[:size - crossover_point]
    child2[crossover_point:] = remaining2[:size - crossover_point]

    return child1.reshape((MAGIC_CUBE_SIZE, MAGIC_CUBE_SIZE, MAGIC_CUBE_SIZE)), \
           child2.reshape((MAGIC_CUBE_SIZE, MAGIC_CUBE_SIZE, MAGIC_CUBE_SIZE))

def mutate(cube):
    idx1, idx2 = np.random.choice(range(MAGIC_CUBE_SIZE ** 3), 2, replace=False)
    flat_cube = cube.flatten()
    flat_cube[idx1], flat_cube[idx2] = flat_cube[idx2], flat_cube[idx1]
    return flat_cube.reshape((MAGIC_CUBE_SIZE, MAGIC_CUBE_SIZE, MAGIC_CUBE_SIZE))

def geneticAlgorithm(population_size, max_iterations):
    population = [create_random_cube() for _ in range(population_size)]
    best_scores = []
    avg_scores = []
    best_cube = None

    for iteration in range(max_iterations):
        print("iteration", iteration, "first population", population[0], "with fitness",
              fitness(population[0]))
        # Evaluate fitness of population
        fitness_values = [1 / (1 + fitness(cube)) for cube in population]
        total_fitness = sum(fitness_values)
        best_scores.append(1 / max(fitness_values) - 1) # Convert fitness back to objective value
        avg_scores.append((sum(1 / f - 1 for f in fitness_values) / population_size)) # Average
        objective score

        # Track the best cube in the current population
        min_index = np.argmin([fitness(cube) for cube in population])
        best_cube = population[min_index]

        # Selection based on fitness
        parents = random.choices(population, weights=fitness_values, k=population_size)

        # Create next generation with crossover and mutation
        next_population = []
        for i in range(0, population_size, 2):
            if i + 1 < population_size:
                child1, child2 = crossover(parents[i], parents[i + 1])
                next_population.append(mutate(child1))
                next_population.append(mutate(child2))
            else:
                next_population.append(mutate(parents[i]))

        population = next_population

    # Best objective function result from final population
    final_best_score = min(fitness(cube) for cube in population)
    return final_best_score, best_scores, avg_scores, best_cube
```

Genetic Algorithm (GA) adalah teknik optimisasi berbasis evolusi yang meniru proses seleksi alam. Dalam kasus *Diagonal Magic Cube*, GA akan mencoba menemukan konfigurasi susunan angka dalam kubus 5x5x5 yang mendekati kriteria *magic cube*. Algoritma ini bekerja melalui beberapa tahapan, yaitu inisialisasi populasi, evaluasi *fitness*, seleksi, crossover, mutasi, dan evaluasi hasil.

Populasi awal terdiri dari sejumlah individu yang mewakili solusi potensial, yang dalam hal ini adalah kubus 5x5x5 dengan angka-angka unik yang tersusun secara acak. Tiap individu (kubus) diinisialisasi dengan memasukkan angka 1 hingga 125 dalam posisi acak, sehingga masing-masing individu memenuhi syarat unik.

Fungsi *fitness* adalah komponen krusial yang mengukur seberapa baik suatu individu memenuhi karakteristik *Diagonal Magic Cube*. Fungsi *fitness* dihitung sebagai selisih antara hasil jumlah diagonal utama dan target yang diharapkan. Semakin kecil selisihnya, semakin tinggi *fitness*-nya, yang berarti kubus tersebut mendekati solusi *magic cube*.

Tahap seleksi digunakan untuk memilih individu-individu terbaik dalam populasi berdasarkan nilai *fitness*-nya. Individu yang memiliki nilai *fitness* yang baik (semakin kecil selisihnya) akan memiliki peluang lebih besar untuk dipilih sebagai *parent* dalam proses *crossover*.

Metode seleksi yang digunakan bisa berupa *roulette wheel selection* atau *fitness proportionate selection*, di mana individu dengan *fitness* lebih tinggi memiliki probabilitas lebih besar untuk terpilih.

Crossover adalah proses penggabungan dua *parent* untuk membentuk dua individu anak (*child*) baru. Dalam kasus ini, *crossover* dilakukan dengan cara memilih titik potong secara acak pada kubus, kemudian:

- Mempertahankan bagian awal dari salah satu *parent* hingga titik potong.
- Memastikan bagian setelah titik potong terdiri dari angka-angka unik yang belum ada di bagian pertama anak.

Hal ini menjaga agar setiap angka tetap unik di kubus hasil *crossover*, karena untuk *Diagonal Magic Cube* ini penting bahwa angka-angka tetap unik tanpa ada duplikasi.

Mutation atau mutasi adalah proses memperkenalkan variasi acak pada individu anak untuk mencegah konvergensi prematur dan menjaga keragaman populasi. Mutasi dilakukan dengan memilih dua posisi acak di kubus dan menukar nilainya.

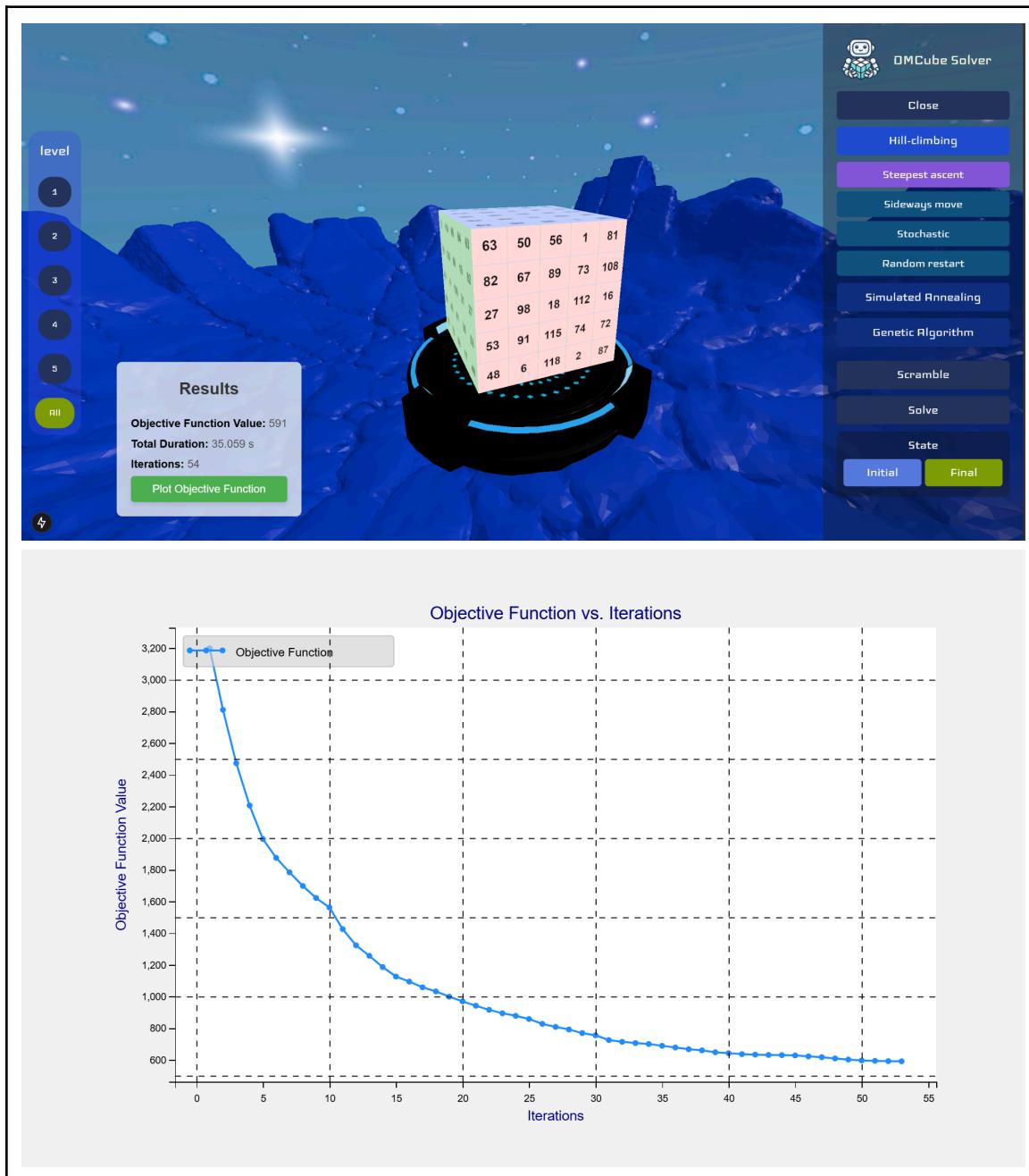
Mutasi ini hanya mengubah dua angka di dalam kubus sehingga menjaga perubahan yang kecil namun signifikan, yang diharapkan dapat membantu proses eksplorasi solusi lebih jauh.

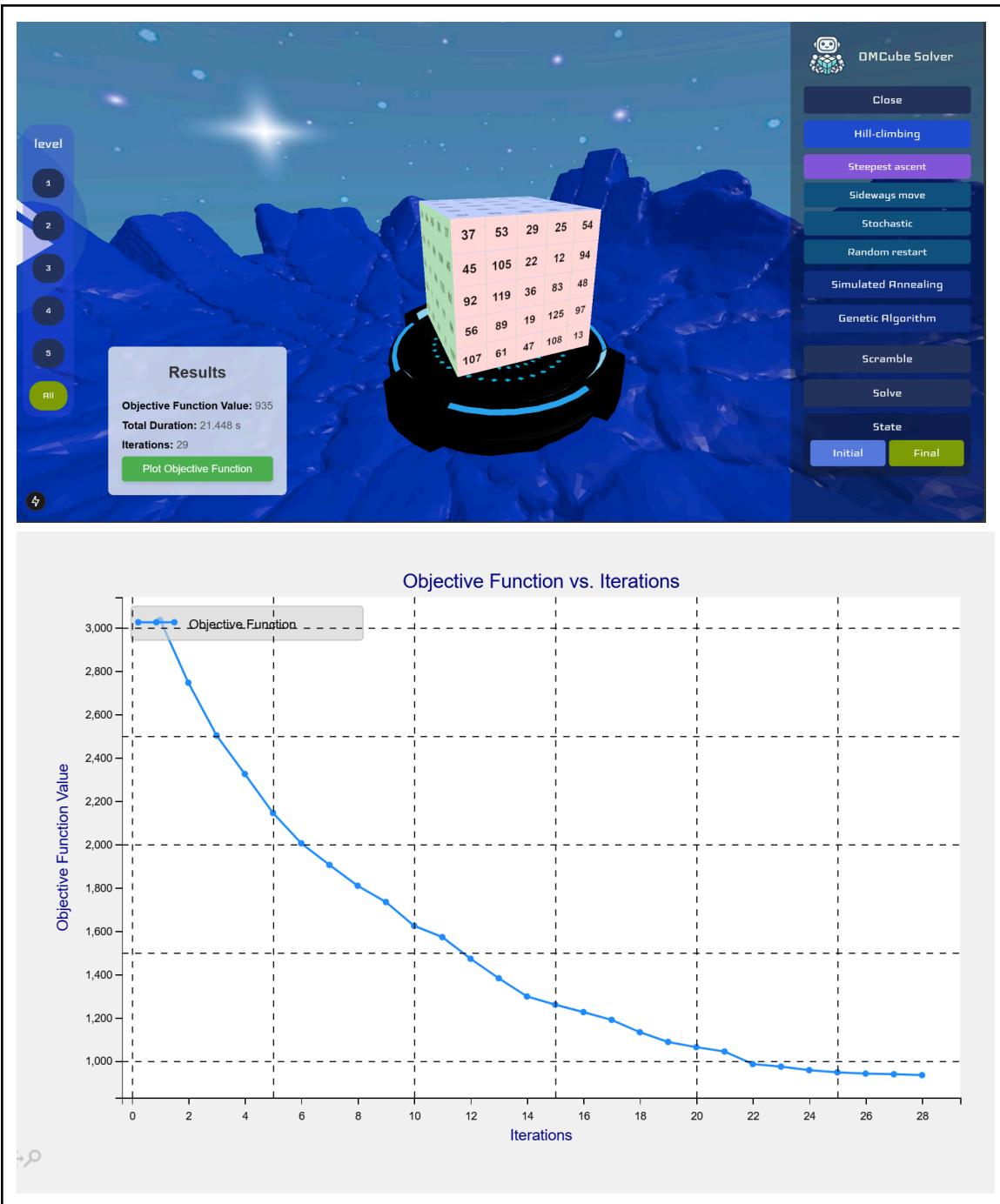
Setelah sejumlah iterasi tertentu, solusi terbaik yang ditemukan dalam populasi dievaluasi. Jika solusi ini memenuhi syarat nilai *fitness* yang diharapkan (yakni mendekati nilai target atau *magic constant* untuk semua diagonal), maka solusi tersebut dianggap sebagai solusi terbaik untuk masalah *Diagonal Magic Cube*.

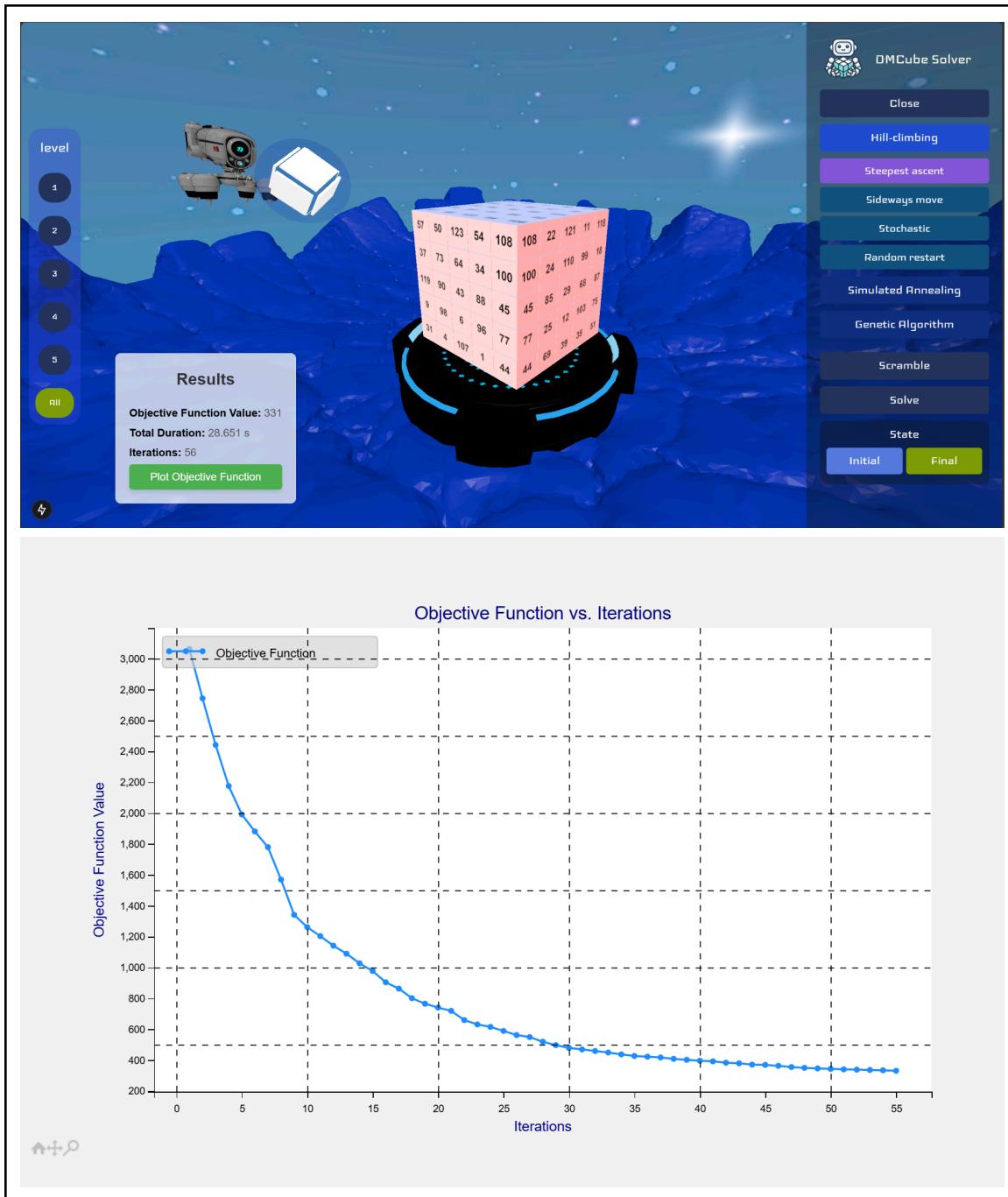
2.3 Hasil Eksperimen dan Analisis

2.3.1 Hill Climb

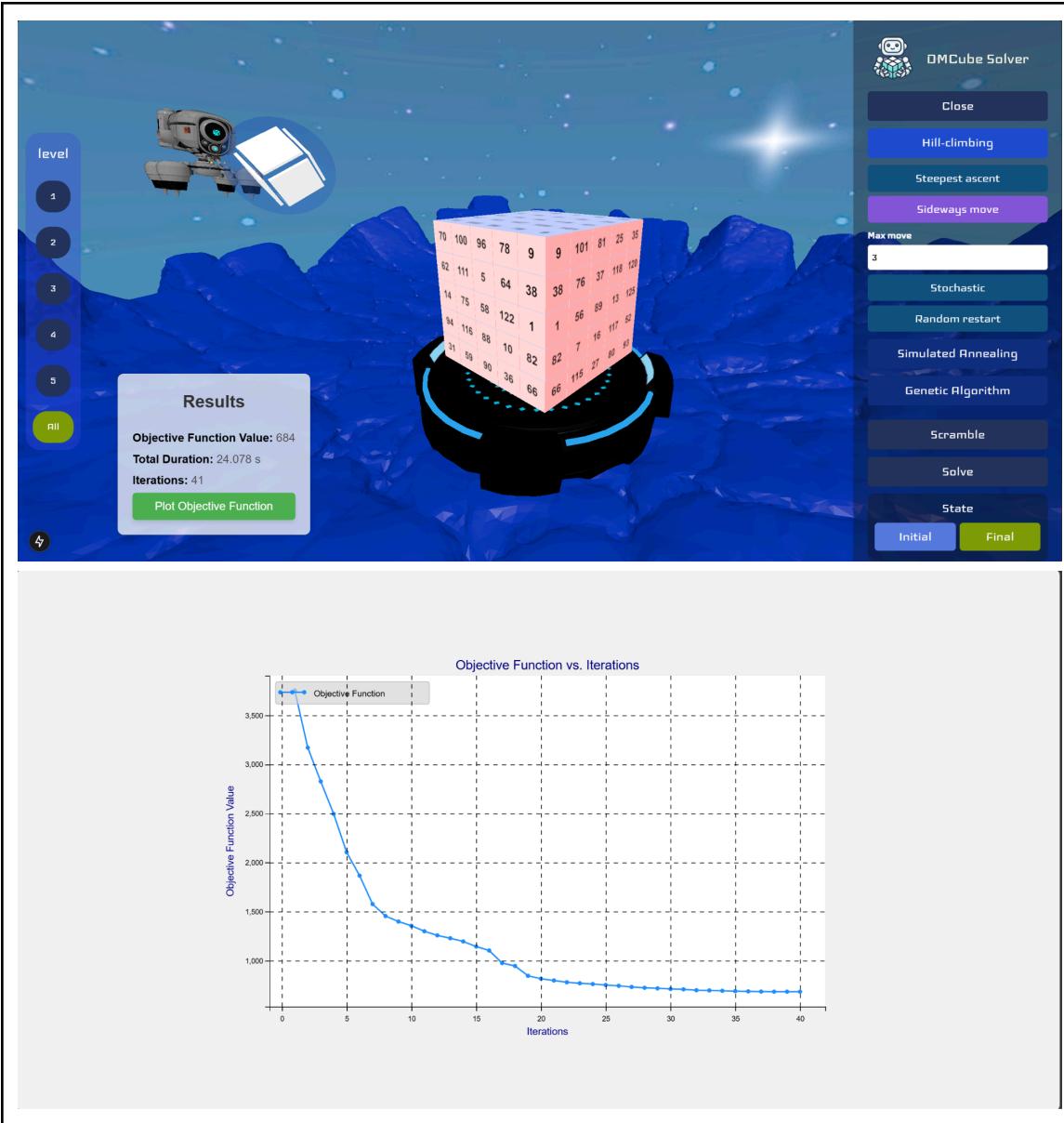
2.3.1.1 Steepest Ascent

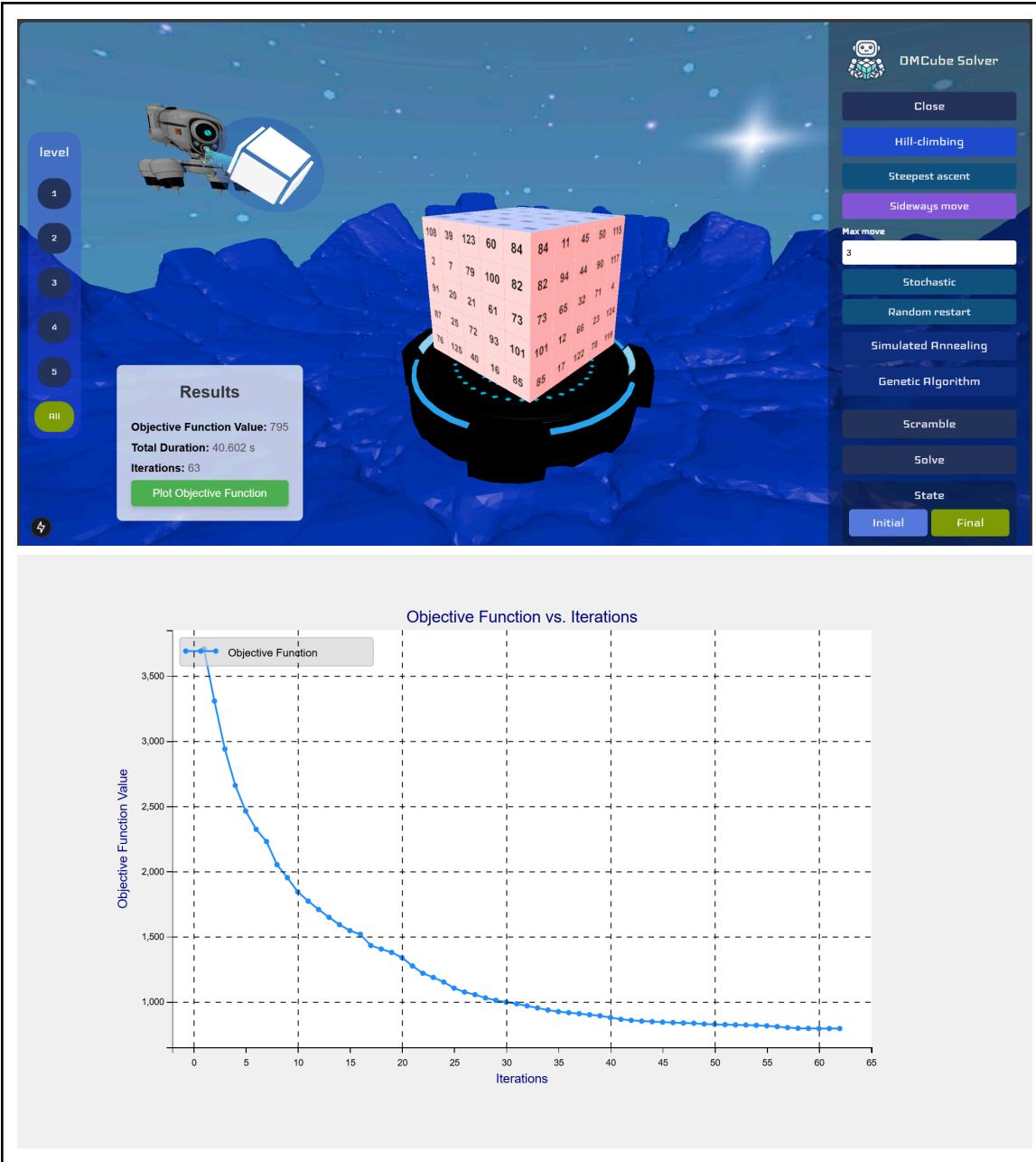


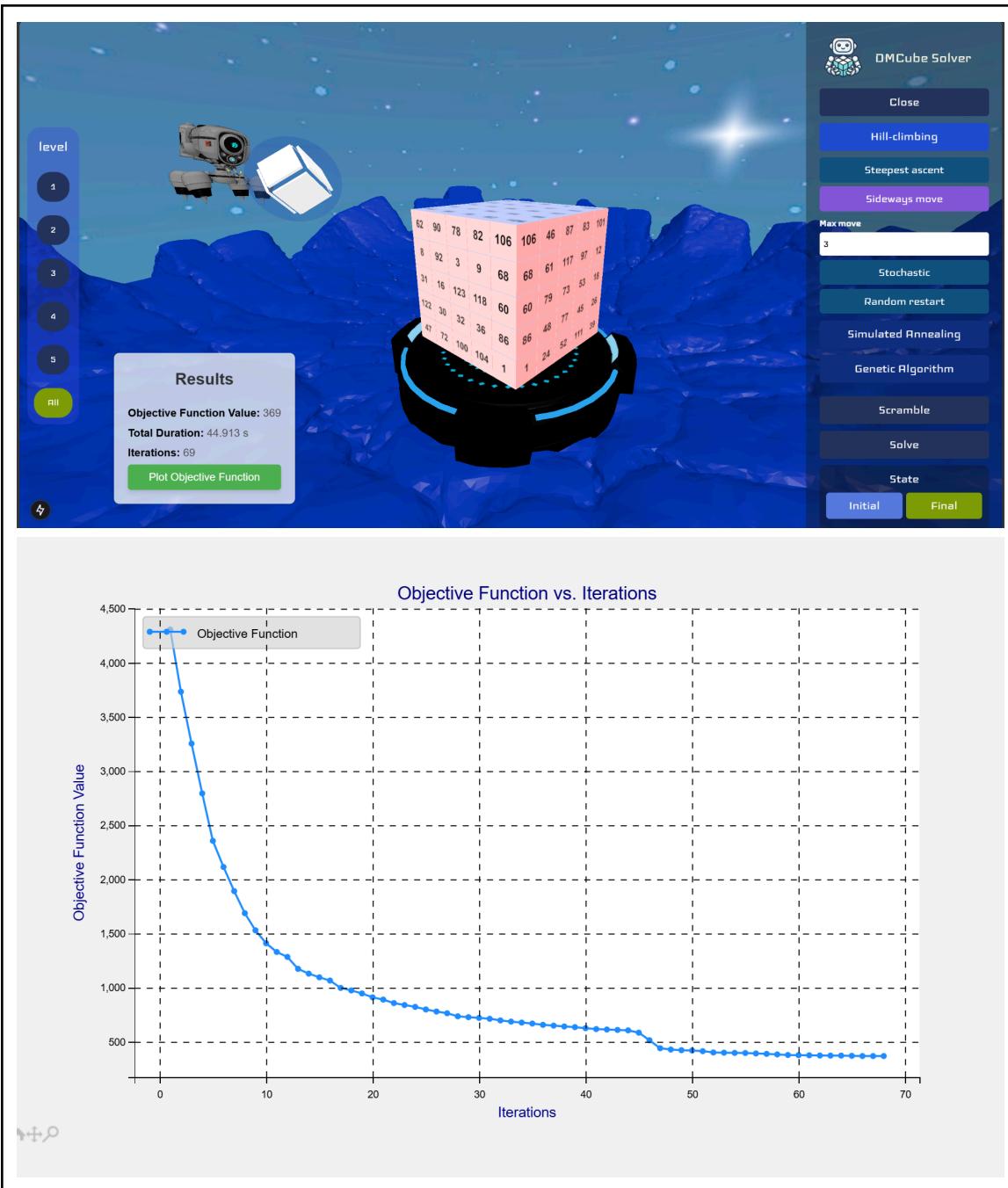




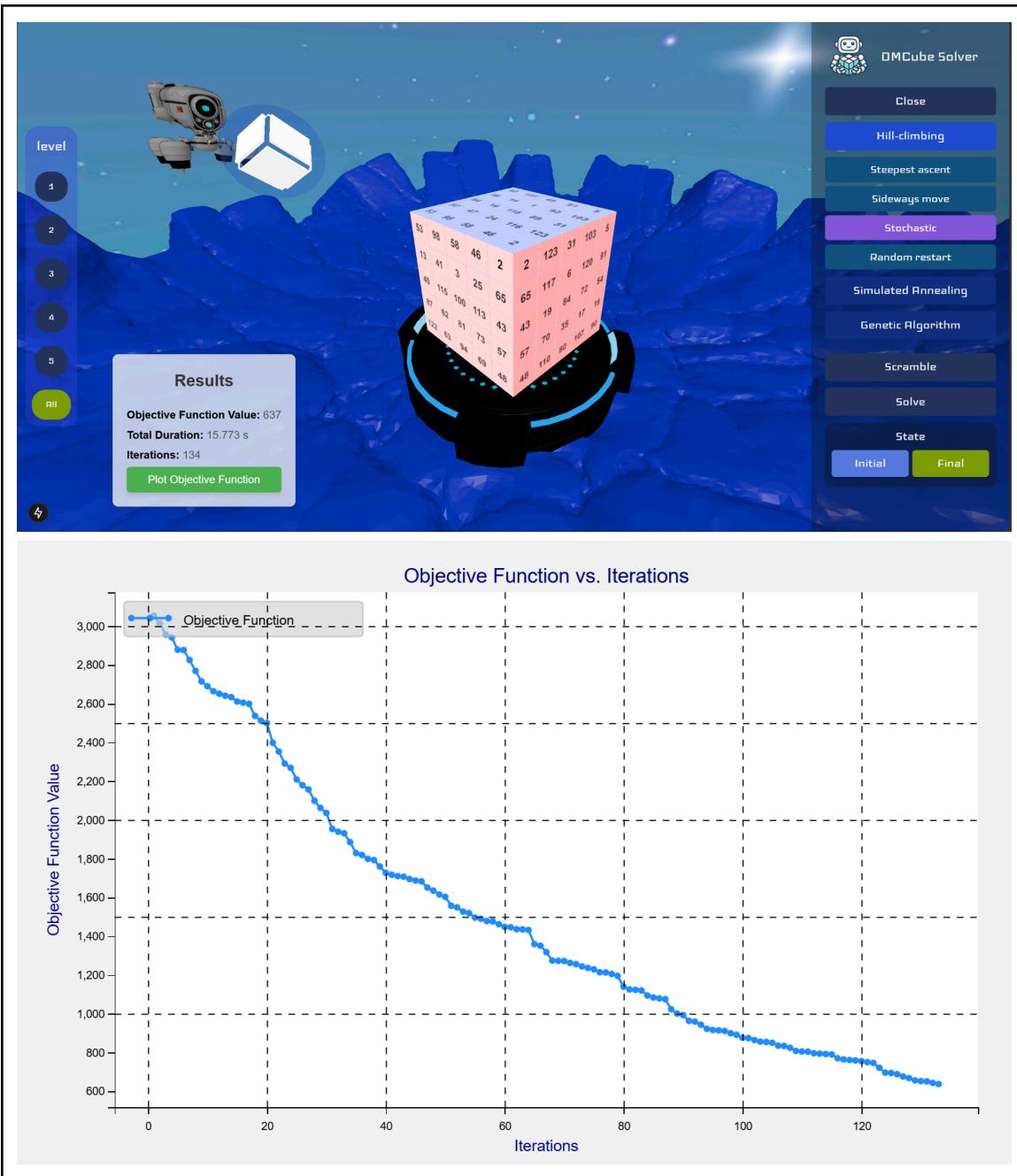
2.3.1.2 Sideways Move

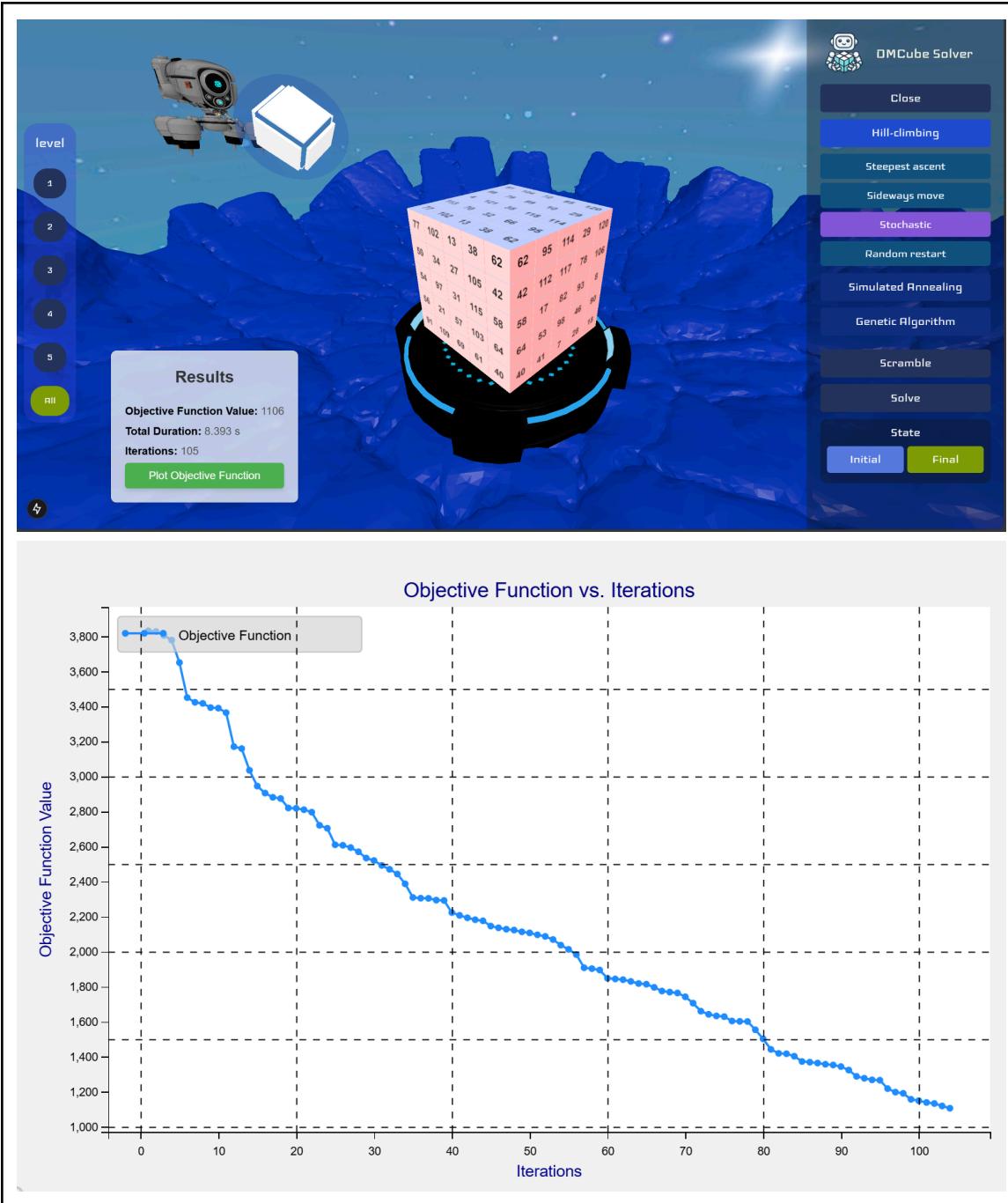


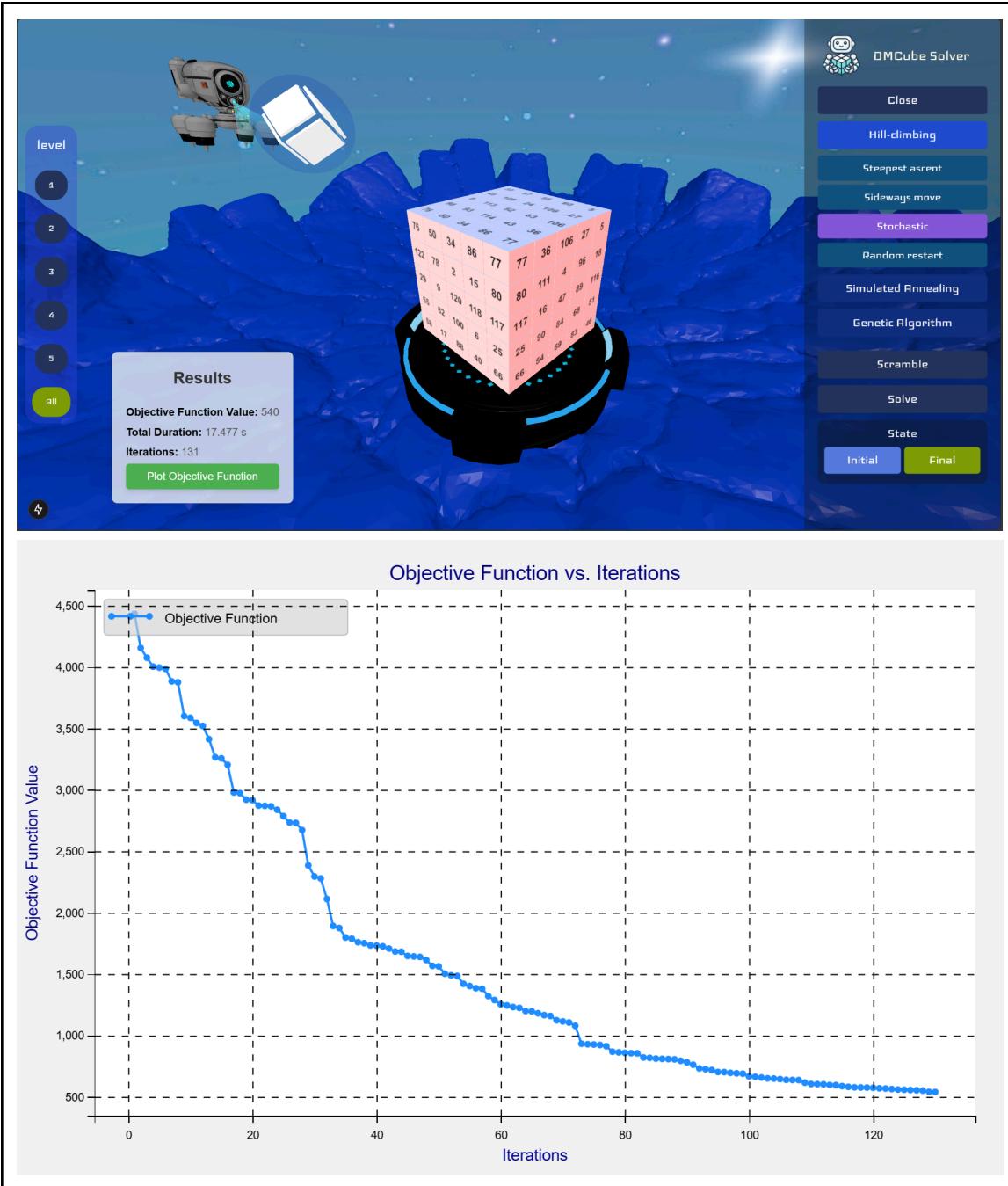




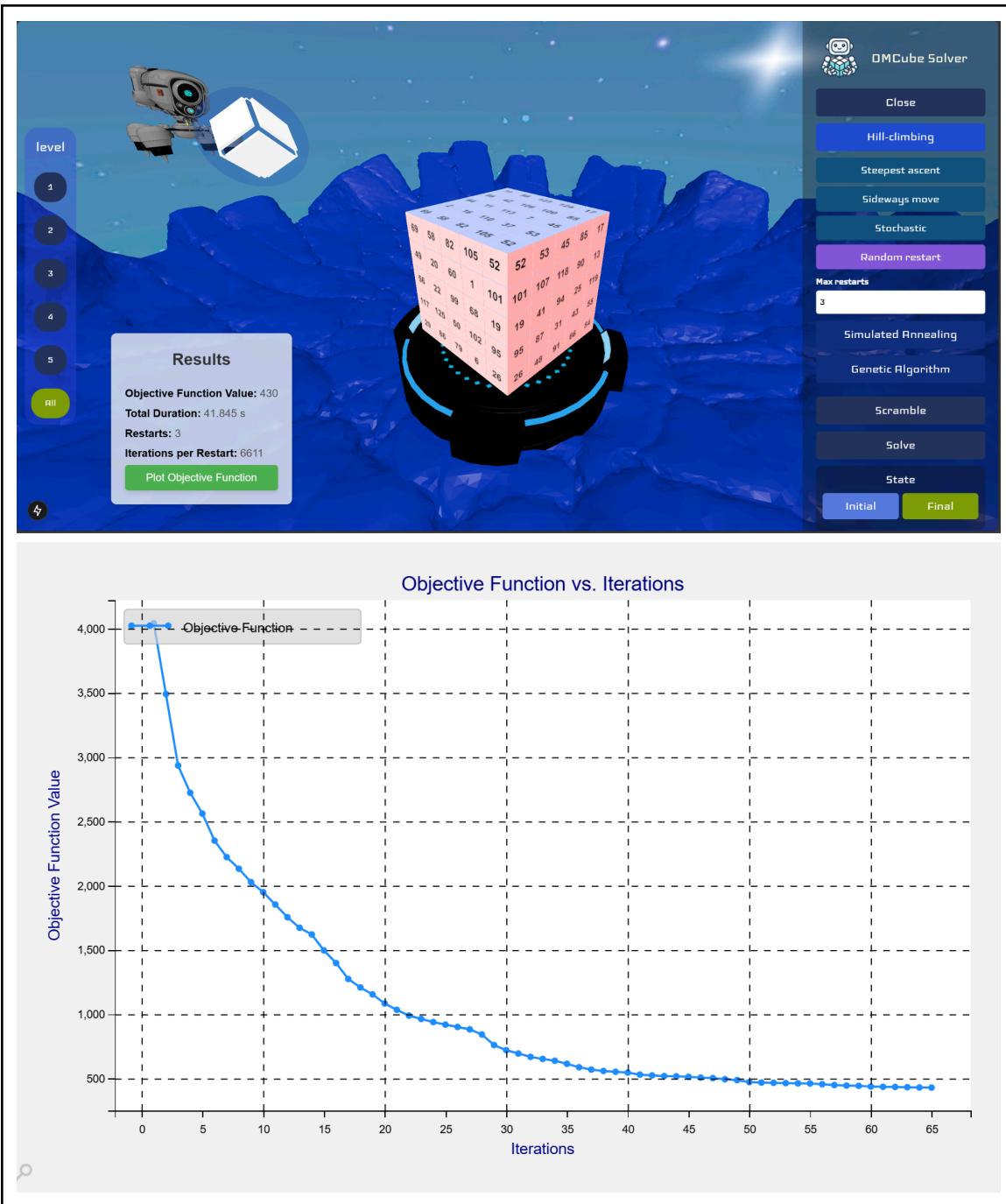
2.3.1.3 Stochastic

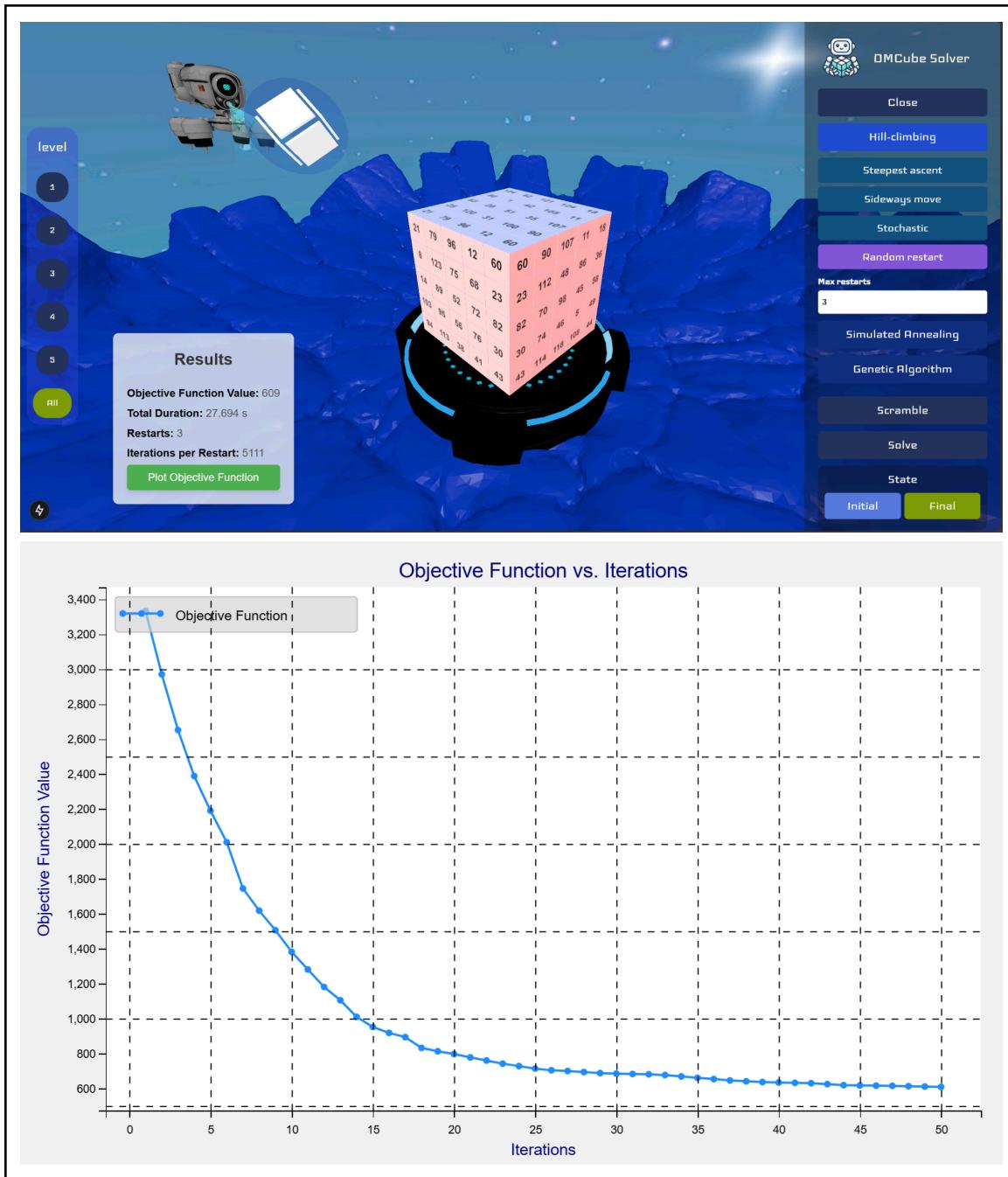


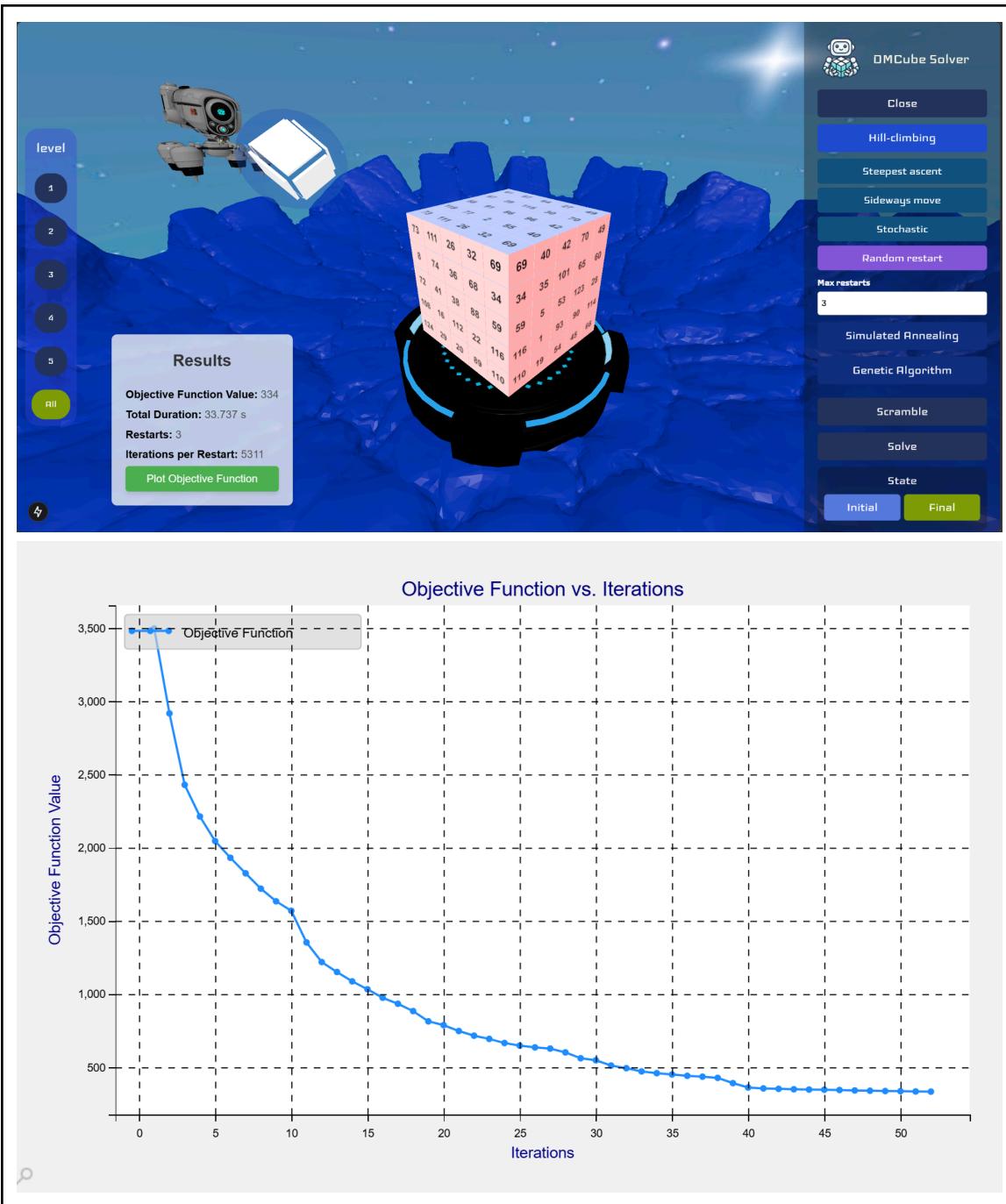




2.3.1.4 Random Restart





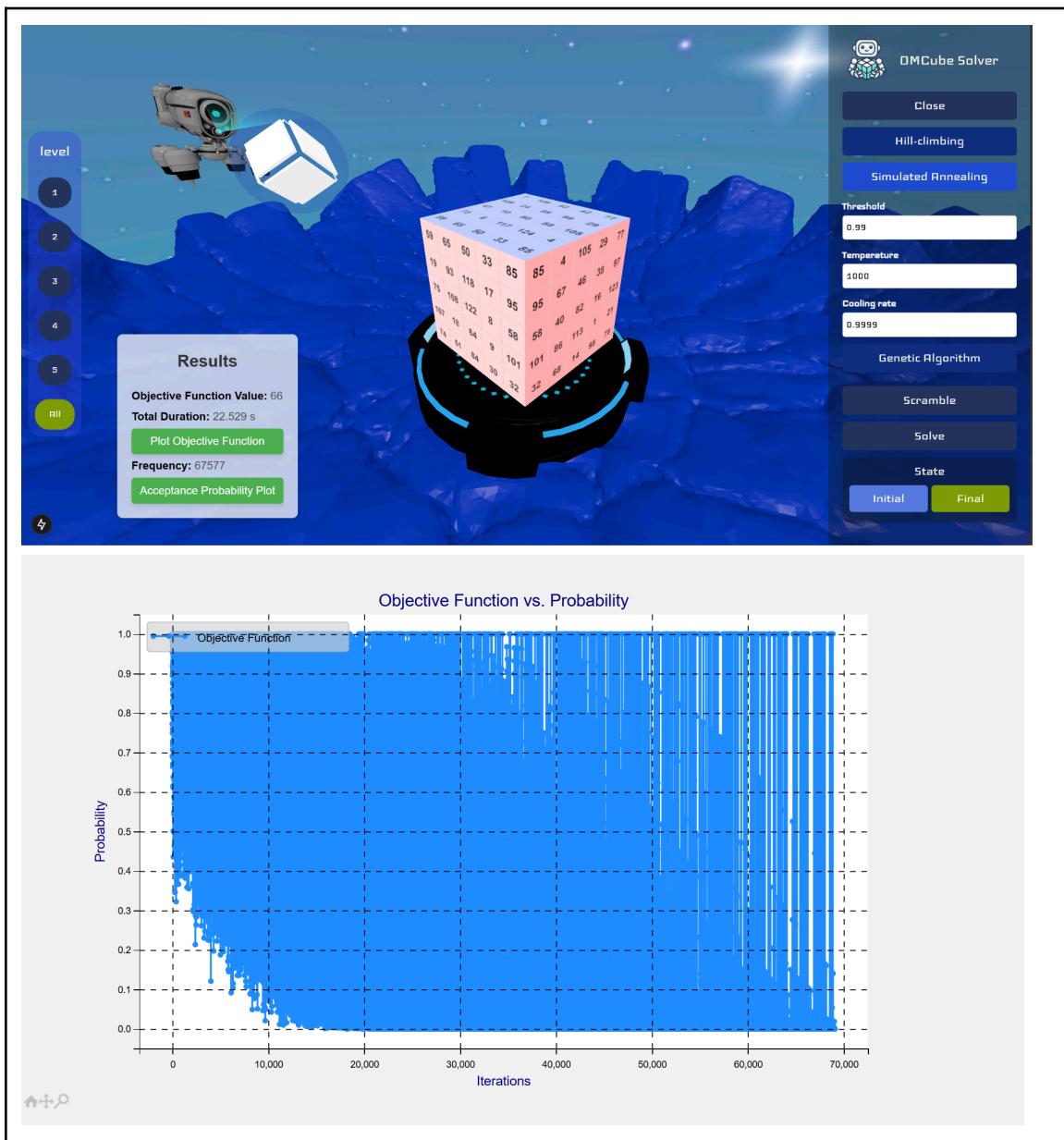


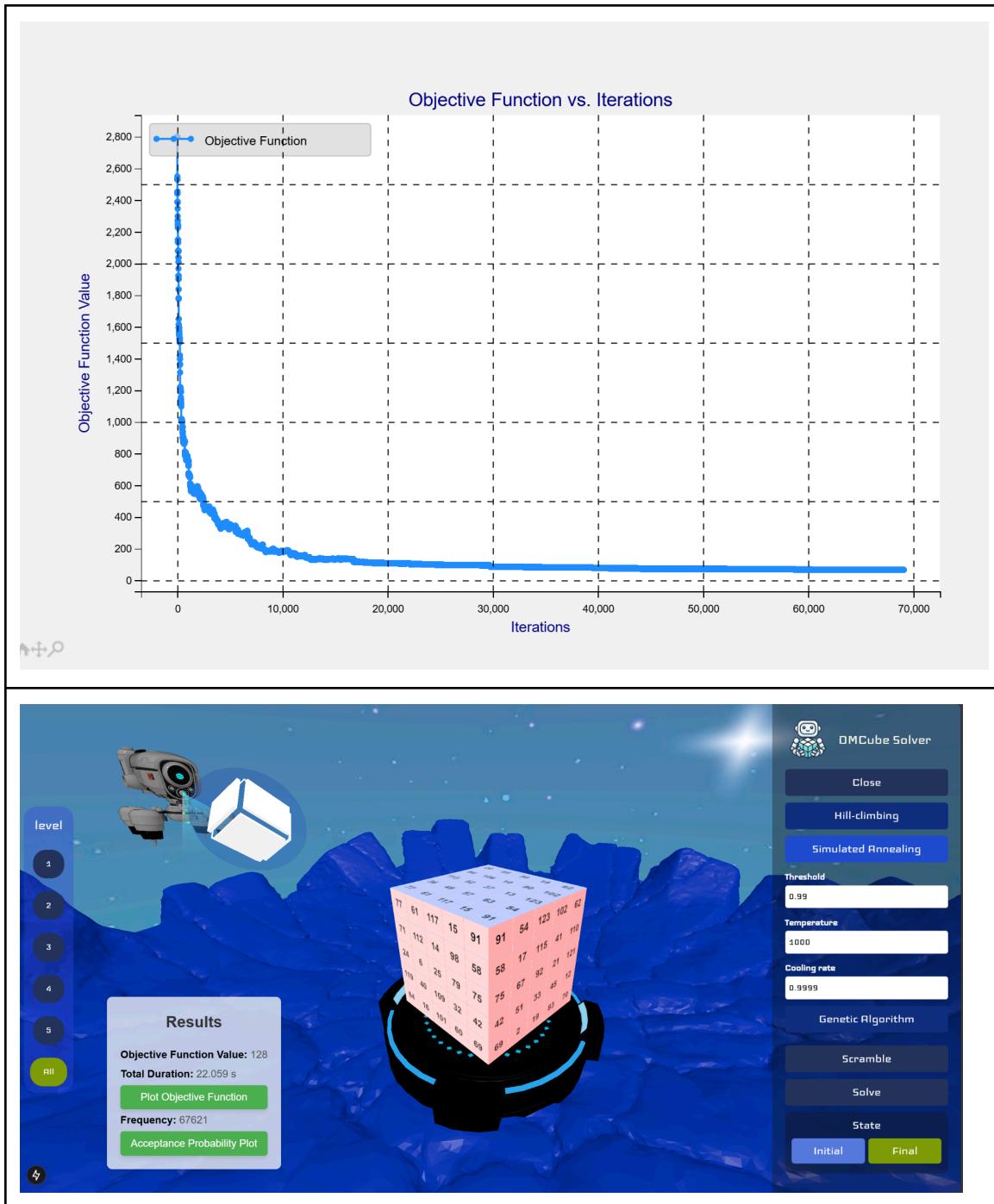
Analisis :

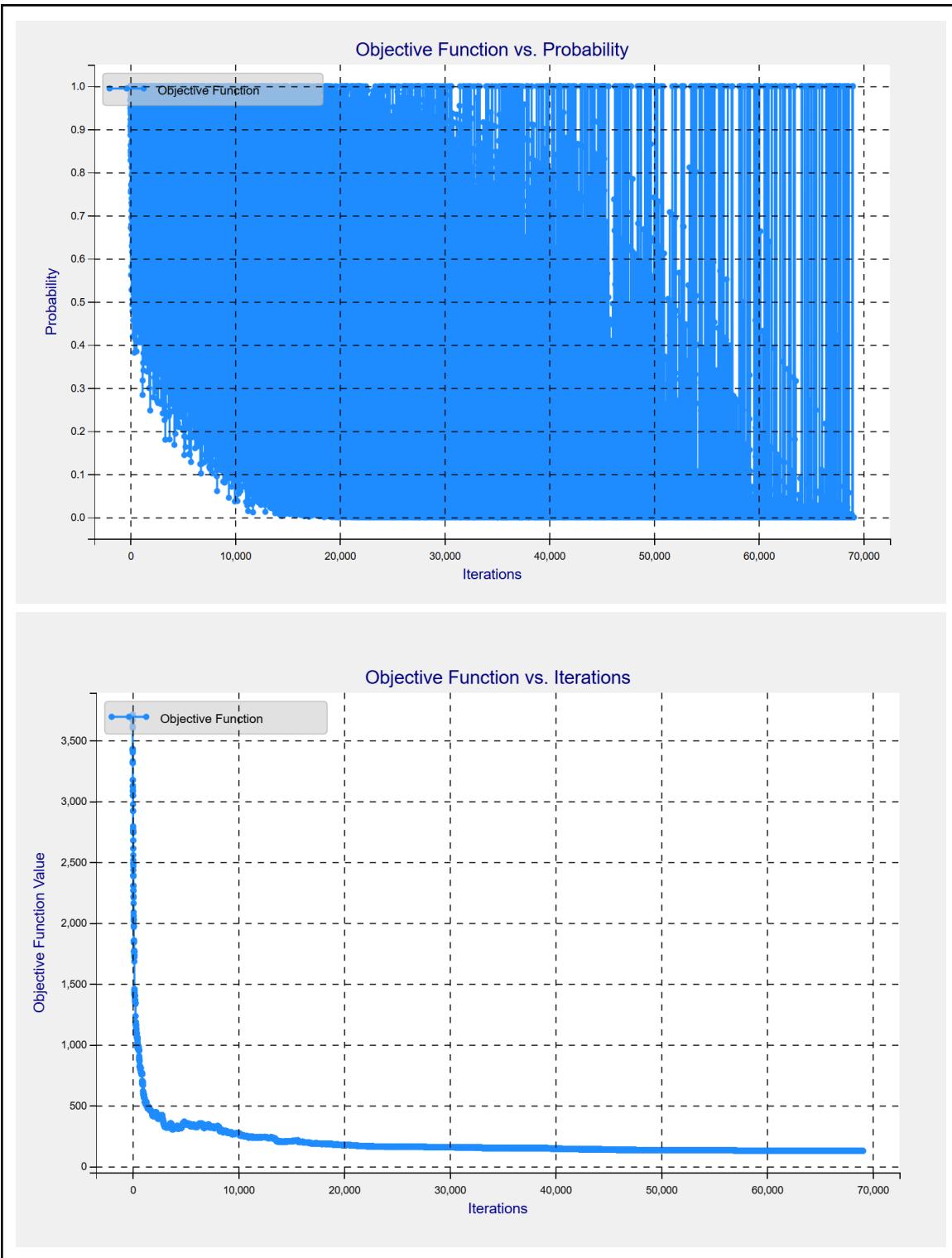
Hasil analisis menunjukkan bahwa nilai objective value yang diperoleh cenderung menuju ke arah global maximum, meskipun belum benar - benar mendekatinya. Hal ini dapat terjadi karena hill-climbing memiliki mekanisme penerimaan neighbor yang lebih baik atau sama dengan current state. Kemampuan ini memungkinkan algoritma untuk mencari langkah yang paling optimal dan efektif di setiap iterasi. Sehingga jika dibandingkan dengan algoritma lain, hill climbing memiliki iterasi paling sedikit by default. Sayangnya dikarenakan algoritma ini diharuskan mencari neighbor state yang

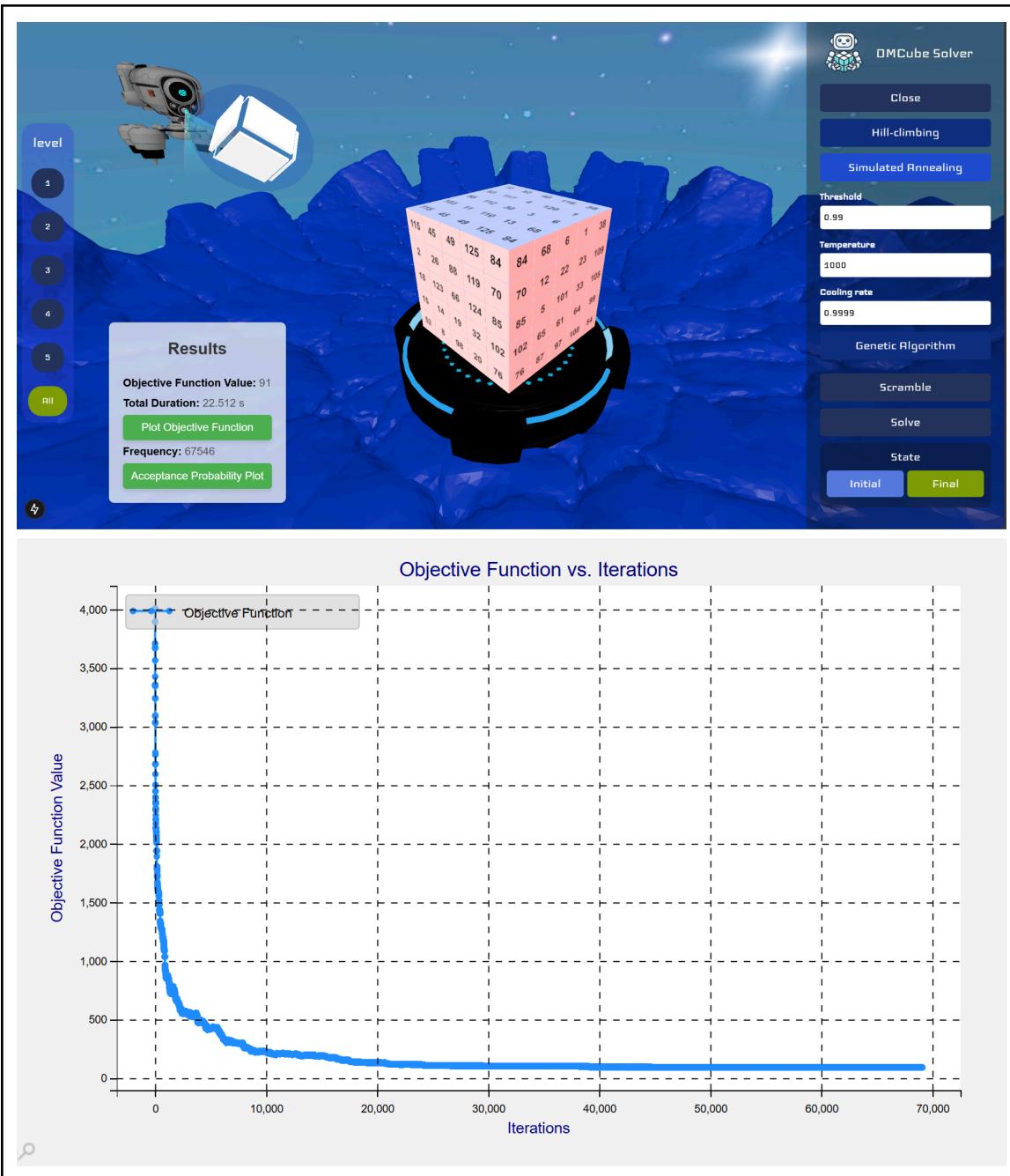
sangat banyak, maka waktu yang dipakai pun cukup banyak rata - rata bisa lebih dari 20 detik.

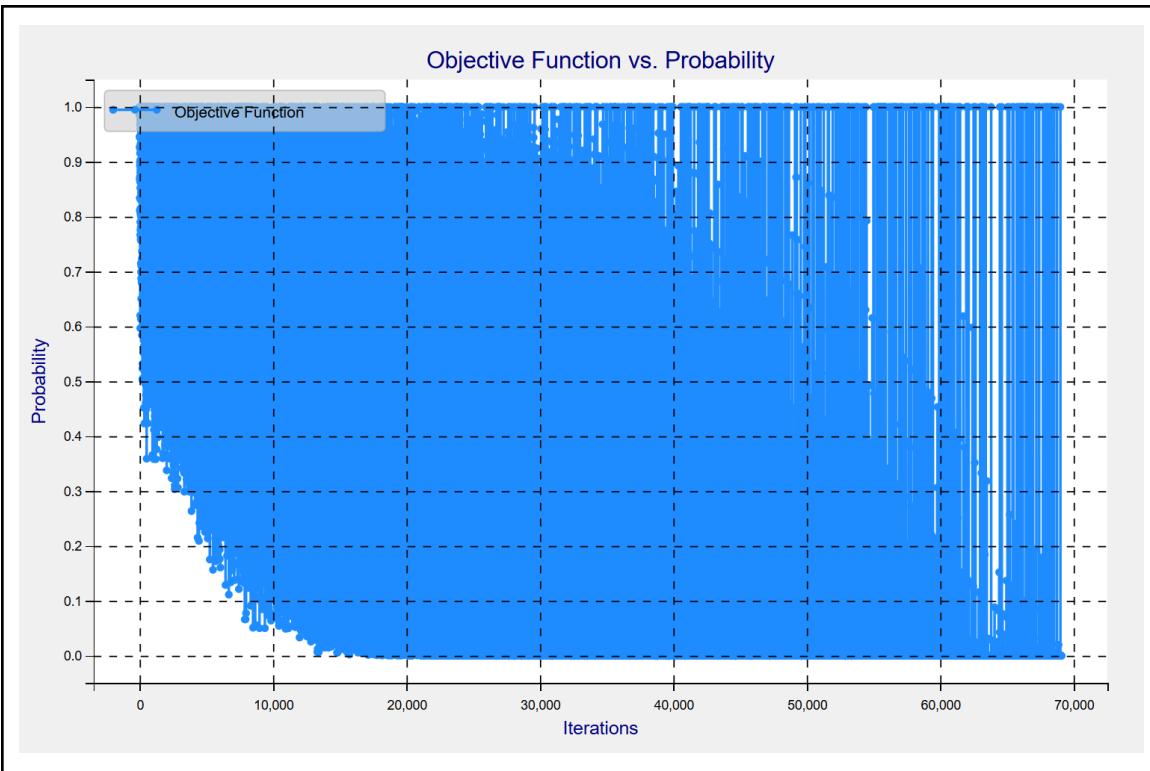
2.3.2 Simulated Annealing









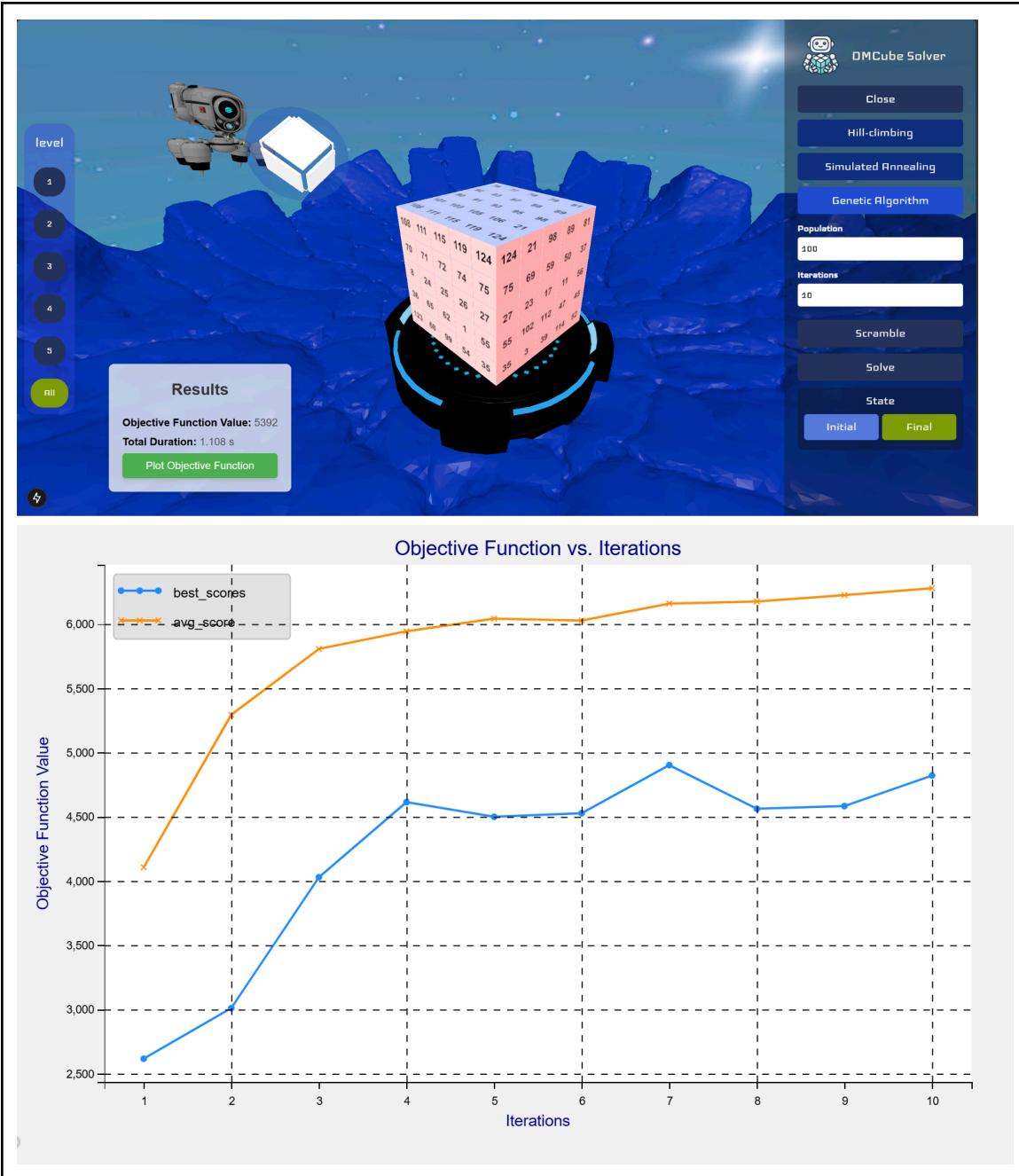


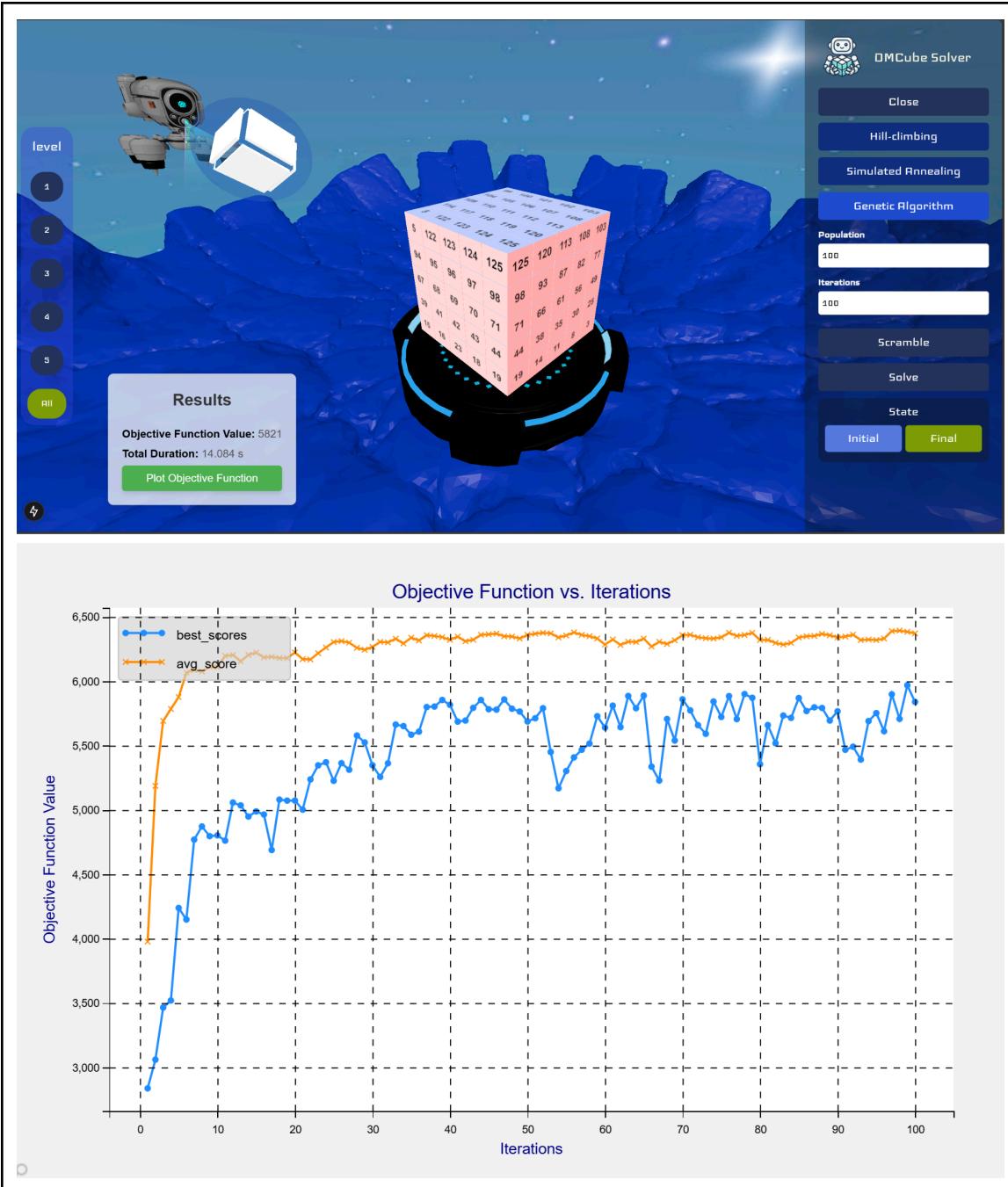
Analisis :

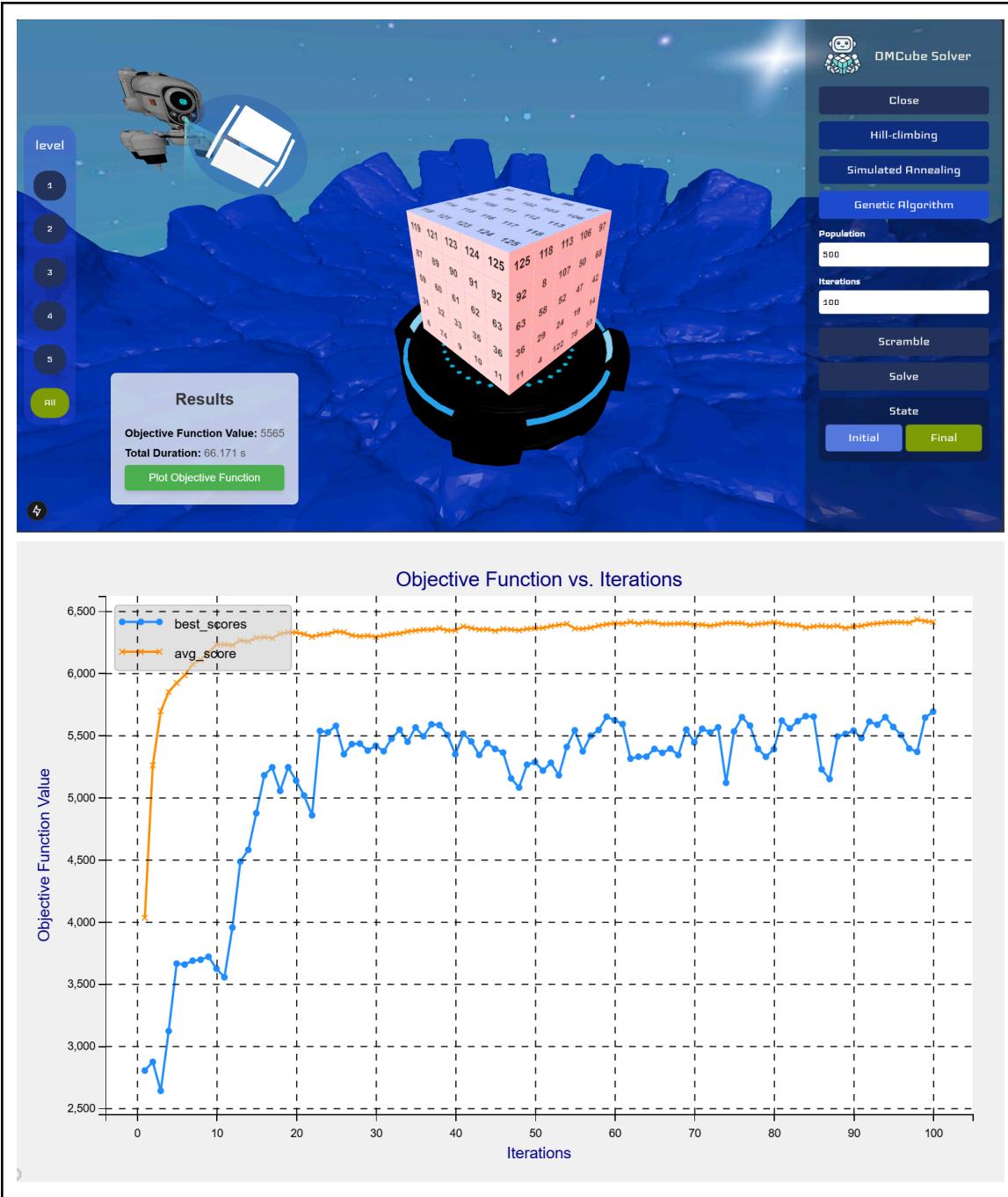
Hasil analisis menunjukkan bahwa nilai objective value yang diperoleh cenderung mendekati global maxima. Hal ini dapat terjadi karena simulated annealing memiliki mekanisme penerimaan neighbor dengan nilai lebih buruk dengan menggunakan fungsi $e^{\Delta E/T}$, dimana jika acceptance probability lebih besar dari pada threshold, kemampuan ini memungkinkan algoritma untuk keluar dari jebakan local maxima, yang sering menghambat algoritma pencarian lokal lainnya. Dengan temperatur dan cooling rate yang tepat, jumlah iterasi dapat ditingkatkan, memungkinkan proses pencarian yang lebih menyeluruh. Semakin banyak iterasi, semakin besar kemungkinan mencapai nilai mendekati global maxima, namun semakin lama durasi pencarinya.

Jika dibandingkan dengan algoritma local search lainnya, simulated annealing mampu memberikan hasil yang lebih optimal. Dengan waktu yang lebih singkat, algoritma ini berhasil menghasilkan nilai objective value yang mendekati global maxima. Berdasarkan keseimbangan antara waktu dan kualitas solusi yang diperoleh, simulated annealing dapat dianggap sebagai pilihan algoritma yang sangat efektif untuk menemukan solusi yang optimal.

2.3.3 Genetic Algorithm







Nilai objective function terbaik dan rata-rata dengan algoritma ini cenderung stagnan dan tidak membaik. Hal ini dikarenakan

- Seberapa dekat tiap-tiap algoritma bisa mendekati global optima dan mengapa hasilnya demikian?
Hill-Climbing → sudah mendekati
Karena algoritma ini memiliki cara mencari solusi yang paling efektif dibandingkan yang lain, algoritma ini mampu menawarkan hasil yang mumpuni

dengan iterasi yang minimal, sayangnya dikarenakan algoritma ini yang tidak bisa menjelajah solusi lebih luas, maka ada kemungkinan algoritma ini terperangkap ke dalam local optimum.

Simulated annealing → paling dekat

Karena algoritma ini memiliki penerimaan solusi yang fleksibel (dapat menerima solusi lebih buruk), pengaturan temperatur dan cooling rate yang menyesuaikan tingkat eksplorasi sepanjang iterasi, dan elemen acak dalam pemilihan neighbor. Sehingga hal ini memungkinkan SA untuk menjelajahi ruang solusi secara luas pada awalnya, kemudian mengarahkan pencarian ke solusi yang semakin optimal seiring penurunan temperatur, sehingga meningkatkan kemungkinan mencapai atau mendekati global maxima. Namun hasil dari simulated annealing sangat bergantung pada parameter temperatur awal, cooling rate, dan threshold, yang akan menentukan jumlah iterasi yang akan dilakukan. Semakin banyak iterasi yang dilakukan maka hasil dari simulated annealing akan semakin baik.

Genetic algorithm → paling jauh.

Karena algoritma ini yang paling tidak bisa ditebak dan acak. Pengubahan konfigurasi kubus sedikit saja dapat meningkatkan nilai objective function secara drastis. Sehingga, 2 parent yang sudah memiliki objective function kecil dapat menjadi rusak (nilai objective meningkat) saat dilakukan crossover dan mutation.

- Bagaimana perbandingan hasil pencarian, durasi, dan konsistensi dari tiap-tiap algoritma dengan algoritma local search yang lain?

Parameter	Hill-climb	Simulated Annealing	Genetic Algorithm
Hasil	Local optima, bukan global optima	Paling mendekati global optima	Jauh dari global optima
Waktu	Cepat, tetapi sering terjebak pada solusi lokal dan tidak optimal	Sedang, membutuhkan waktu lebih lama untuk eksplorasi solusi suboptimal namun memiliki peluang menghindari solusi lokal	Lambat, membutuhkan waktu yang lebih lama karena populasi besar dan operasi genetik
Konsistensi	Tidak konsisten, hasil sangat bergantung pada	Lebih konsisten dibanding Hill Climbing, tetapi	Konsisten, hasil cenderung serupa karena variasi

	titik awal dan sering menghasilkan solusi yang berbeda pada setiap eksekusi	hasil masih bervariasi karena adanya elemen probabilitas	populasi yang besar dan kemampuan berevolusi menuju solusi optimal
--	---	--	--

- Bagaimana pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada Genetic Algorithm?

Seharusnya semakin banyak iterasi dan populasi, hasil dari algoritma ini akan semakin baik. Namun, pada permasalahan Diagonal Magic Cube, dikarenakan banyaknya faktor acak, genetic algorithm menjadi sangat tidak efektif.

BAB III

KESIMPULAN DAN SARAN

3.1 Kesimpulan

Dalam penelitian dan implementasi algoritma *local search* untuk menyelesaikan permasalahan *magic cube*, ditemukan bahwa algoritma terbaik untuk mencapai solusi optimal adalah **simulated annealing**. Algoritma ini berhasil menyeimbangkan eksplorasi dan eksploitasi, memungkinkan solusi keluar dari *local optimum* dan mengeksplorasi ruang pencarian yang lebih luas, sehingga meningkatkan peluang menemukan solusi global.

Berikut rangkuman hasil analisis:

- **Simulated Annealing:** Menunjukkan kinerja terbaik karena mampu melakukan *controlled randomness* untuk menghindari *local maxima* dan menemukan solusi optimal dengan efisiensi waktu yang relatif baik.
- **Hill Climbing:** Meskipun lebih cepat dalam eksekusi dibandingkan *simulated annealing*, algoritma ini rentan terjebak di *local optimum*. Namun, dalam beberapa percobaan, algoritma ini tetap mampu menghasilkan hasil yang mendekati optimal, terutama jika dilengkapi dengan modifikasi seperti *sideways move* atau *random restart*.
- **Genetic Algorithm:** Menunjukkan ketidakefektifan untuk permasalahan Diagonal Magic Cube karena terlalu banyaknya faktor acak yang menyebabkan perginya ke arah yang tidak lebih baik.

Secara keseluruhan, *simulated annealing* terbukti unggul dalam menyelesaikan permasalahan dengan efisiensi tinggi, sementara *hill climbing* dapat menjadi alternatif cepat dalam kondisi tertentu. *Genetic algorithm*, meskipun memiliki potensi yang besar dalam diversifikasi solusi, memerlukan optimasi tambahan untuk meningkatkan kinerjanya pada kasus ini.

3.2 Saran

Berdasarkan hasil penelitian ini, terdapat beberapa saran yang dapat dipertimbangkan untuk pengembangan lebih lanjut:

1. **Penggabungan Algoritma:** Mengkombinasikan algoritma *simulated annealing* dengan strategi *genetic algorithm* seperti *crossover* yang adaptif dapat meningkatkan performa pencarian solusi.
2. **Optimasi Parameter:** Penelitian lebih lanjut sebaiknya melibatkan optimasi parameter untuk algoritma *simulated annealing* dan *genetic algorithm*, seperti suhu awal dan tingkat penurunan suhu (cooling schedule) serta ukuran populasi dan probabilitas mutasi.

3. **Penggunaan Teknik Hybrid:** Menggabungkan *hill climbing* dengan *simulated annealing* dalam tahap eksplorasi awal untuk mempercepat proses pencarian awal dan kemudian meneruskan dengan *simulated annealing* dapat menghasilkan hasil yang lebih efisien.
4. **Implementasi Paralel:** Memanfaatkan pendekatan paralel untuk *genetic algorithm* dan *hill climbing* dapat membantu mengurangi waktu komputasi secara signifikan.
5. **Evaluasi Skala Lebih Besar:** Uji coba pada skala masalah yang lebih besar dan beragam, termasuk kompleksitas tambahan dalam *magic cube*, dapat memberikan wawasan lebih dalam tentang performa masing-masing algoritma.

BAB IV

PEMBAGIAN TUGAS

NIM	Tugas
13522131	Genetic Algorithm
13522141	Frontend
13522143	Simulated Annealing
13522155	Hill Climb Algorithm

BAB V

REFERENSI

1. Ahmed, M. M. (2004). *Algebraic Combinatorics of Magic Squares*. arXiv: Combinatorics.
2. Johnson, A. M., Hale, M., Haynes, G. C., & Koditschek, D. E. (2011). *Autonomous legged hill and stairwell ascent*. IEEE, pp. 134-142.
3. Hoffmann, J. (2000). *A Heuristic for Domain Independent Planning and Its Use in an Enforced Hill-Climbing Algorithm*. In Springer, Berlin, Heidelberg, pp. 216-227.
4. Tovey, C. A. (1985). *Hill Climbing with Multiple Local Optima*. Siam Journal on Algebraic and Discrete Mathematics, 6(3), 384-393.
5. Aarts, E. H. L., & Korst, J. (1989). *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Eindhoven University of Technology, Philips.
6. Reeves, C. R. (1993). *Genetic Algorithms*. pp. 151-196.
7. Focacci, F., Laburthe, F., & Lodi, A. (2003). *Local Search and Constraint Programming*. In Springer, Boston, MA, pp. 369-403.
8. Paquete, L., Schiavinotto, T., & Stützle, T. (2007). *On local optima in multiobjective combinatorial optimization problems*. Annals of Operations Research, 156(1), 83-97.
9. Fang, H., & Ruml, W. (2004). *Complete local search for propositional satisfiability*. In AAAI Press, pp. 161-166.
10. Myers, R. H., & Khuri, A. I. (1979). *A new procedure for steepest ascent*. Communications in Statistics-theory and Methods, 8(14), 1359-1376.
11. Hoffmann, J. (2000). *A Heuristic for Domain Independent Planning and Its Use in an Enforced Hill-Climbing Algorithm*. In Springer, Berlin, Heidelberg, pp. 216-227.
12. *Perfect Magic Cube*. MathWorld--A Wolfram Web Resource. Tersedia di: <https://mathworld.wolfram.com/PerfectMagicCube.html>. Diakses pada 29 Oktober 2024.