

**Laporan Tugas Besar 1
IF3170 Inteligensi Artifisial**
Pencarian Solusi Diagonal Magic Cube dengan Local Search
Semester I Tahun 2024/2025



Disusun Oleh:
Randy Verdian 13522067
Emery Fathan Zwageri 13522079
Sa'ad Abdul Hakim 13522092
Ellijah Darrellshane S. 13522097

**Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2024**

Daftar Isi

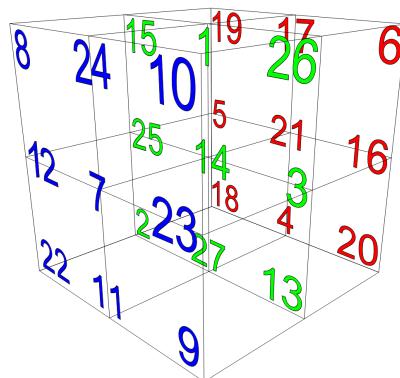
| | |
|---|-----------|
| Daftar Isi | 2 |
| Bab 1 | |
| Deskripsi Persoalan | 3 |
| Bab 2 | |
| Pembahasan | 5 |
| 2.1. Pemilihan Objective Function | 5 |
| 2.2. Implementasi Algoritma Local Search | 6 |
| 2.2.1. Steepest Ascent Hill Climbing | 6 |
| 2.2.2. Stochastic Hill Climbing | 7 |
| 2.2.3. Hill Climbing with Sideways Move | 7 |
| 2.2.4. Random Restart Hill Climbing | 9 |
| 2.2.5. Simulated Annealing | 10 |
| 2.2.6. Genetic Algorithm | 11 |
| 2.3. Hasil Eksperimen dan Analisis | 16 |
| 2.3.1. Steepest Ascent Hill Climbing | 16 |
| 2.3.2. Stochastic Hill Climbing | 21 |
| 2.3.3. Hill Climbing with Sideways Move | 26 |
| 2.3.4. Random Restart Hill Climbing | 31 |
| 2.3.5. Simulated Annealing | 36 |
| 2.3.6. Genetic Algorithm | 43 |
| Pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada Genetic Algorithm adalah sebagai berikut: | 65 |
| 2.4. Perbandingan Seluruh Algoritma Local Search | 66 |
| Bab 3 | |
| Kesimpulan dan Saran | 67 |
| Bab 4 | |
| Pembagian Tugas | 68 |
| Bab 5 | |
| Referensi | 69 |
| Bab 6 | |
| Lampiran | 69 |

Bab 1

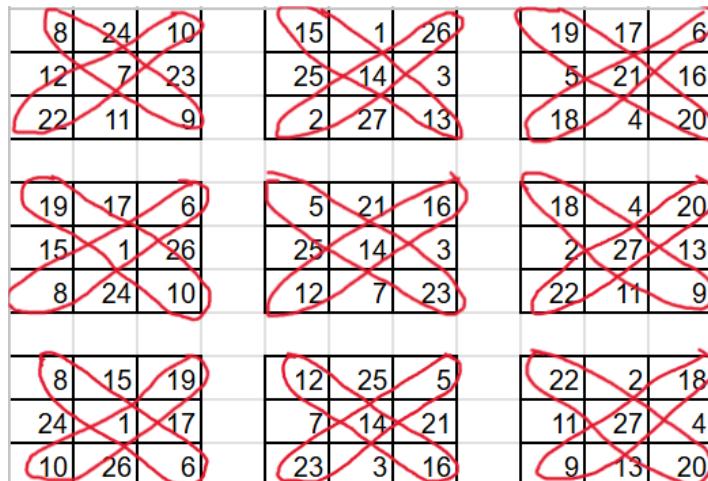
Deskripsi Persoalan

Diagonal magic cube merupakan kubus yang tersusun dari angka 1 hingga n^3 tanpa pengulangan dengan n adalah panjang sisi pada kubus tersebut. Angka-angka pada tersusun sedemikian rupa sehingga properti-properti berikut terpenuhi:

- Terdapat satu angka yang merupakan magic number dari kubus tersebut (Magic number tidak harus termasuk dalam rentang 1 hingga n^3 , magic number juga bukan termasuk ke dalam angka yang harus dimasukkan ke dalam kubus)
- Jumlah angka-angka untuk setiap baris sama dengan magic number
- Jumlah angka-angka untuk setiap kolom sama dengan magic number
- Jumlah angka-angka untuk setiap tiang sama dengan magic number
- Jumlah angka-angka untuk seluruh diagonal ruang pada kubus sama dengan magic number
- Jumlah angka-angka untuk seluruh diagonal pada suatu potongan bidang dari kubus sama dengan magic number
 - Berikut ilustrasi dari potongan bidang yang ada pada suatu kubus berukuran 3:



- Terdapat 9 potongan bidang, yaitu:



- Diagonal yang dimaksud adalah yang dilingkari warna merah saja

- Ilustrasi dan penjelasan lebih detail bisa anda lihat di link berikut: [Features of the magic cube - Magisch vierkant](#)

Pada tugas ini, kami akan menyelesaikan permasalahan Diagonal Magic Cube berukuran 5x5x5. Initial state dari suatu kubus adalah susunan angka 1 hingga 5^3 secara acak. Kemudian, tiap iterasi pada algoritma local search, langkah yang boleh dilakukan adalah menukar posisi dari 2 angka pada kubus tersebut (2 angka yang ditukar tidak harus bersebelahan). Khusus untuk genetic algorithm, boleh dilakukan penukaran posisi lebih dari 2 angka sekaligus dalam satu iterasi (tetapi hanya menukar posisi 2 angka saja juga diperbolehkan).

Bab 2

Pembahasan

2.1. Pemilihan Objective Function

Dengan menggunakan rumus *magic number*, nilai *magic number* dalam sebuah kubus berukuran $5 \times 5 \times 5$ adalah $0.5(5^4 + 5)$, yang setara dengan 315. Karena nilai *magic number* ini sudah diketahui, kami menetapkan fungsi objektif *count_of_match(n)* yang mengukur seberapa dekat kondisi saat ini dengan tujuan atau *goal*. Fungsi ini disebut sebagai *state value*, di mana semakin tinggi nilai *count_of_match(n)*, semakin dekat kondisi tersebut dengan tujuan.

Goal yang ingin dicapai adalah agar jumlah angka pada setiap baris, kolom, tiang, diagonal ruang, dan diagonal pada bidang tertentu dari kubus sama dengan *magic number*. Nilai maksimum dari *count_of_match(n)* adalah total dari seluruh baris, kolom, tiang, diagonal ruang, dan diagonal bidang di dalam kubus, yaitu $25 + 25 + 25 + 4 + 30 = 109$. Dengan demikian, rumus *count_of_match(n)* dapat dinyatakan sebagai:

$$count_of_match(n) = b(n) + k(n) + t(n) + r(n) + d(n)$$

dengan rincian sebagai berikut:

- $b(n)$: jumlah baris yang memiliki total angka sama dengan *magic number* (315),
- $k(n)$: jumlah kolom yang memiliki total angka sama dengan *magic number* (315),
- $t(n)$: jumlah tiang yang memiliki total angka sama dengan *magic number* (315),
- $r(n)$: jumlah diagonal ruang yang memiliki total angka sama dengan *magic number* (315),
- $d(n)$: jumlah diagonal pada bidang tertentu di dalam kubus yang memiliki total angka sama dengan *magic number* (315).

Dengan demikian, tujuan utama adalah untuk mencapai atau mendekati nilai *count_of_match(n) = 109*.

2.2. Implementasi Algoritma Local Search

2.2.1. Steepest Ascent Hill Climbing

Algoritma *Steepest Ascent Hill Climbing* ini dimulai dengan kondisi awal kubus dan berusaha memaksimalkan nilai state melalui serangkaian perbaikan iteratif. Pada setiap iterasi, algoritma menghasilkan semua kemungkinan tetangga dari keadaan saat ini ($125C2 = 7750$ kemungkinan), lalu menghitung *state value* setiap tetangga dengan menggunakan fungsi `count_of_match`. Algoritma kemudian memilih tetangga dengan nilai tertinggi dan memperbarui kubus jika nilai tetangga tersebut lebih baik daripada nilai kubus saat ini. Proses ini berulang hingga tidak ada tetangga yang memiliki nilai lebih tinggi dari kubus saat ini, menandakan bahwa algoritma telah mencapai puncak lokal, lalu algoritma berhenti.

```
def steepest_ascent(current_cube):
    count_iteration = 0
    current_score = count_of_match(current_cube)
    scores = [current_score]
    start_time = time.time()

    while True:
        count_iteration += 1
        neighbors = generate_all_neighbors(current_cube)
        best_neighbor = None
        best_score = current_score

        for neighbor in neighbors:
            score = count_of_match(neighbor)
            if score > best_score:
                best_neighbor = neighbor
                best_score = score

        if best_neighbor is not None and best_score > current_score:
            current_cube = best_neighbor
            current_score = best_score
            scores.append(current_score)
        else:
            break

    end_time = time.time()
    duration = end_time - start_time

    return count_iteration, current_cube, current_score, scores,
```

```
duration
```

2.2.2. Stochastic Hill Climbing

Algoritma *Stochastic Hill Climbing* ini memulai dengan kondisi awal kubus dan bertujuan untuk memperbaiki *state* secara iteratif hingga mencapai nilai terbaik yang ditemukan dalam jumlah iterasi tertentu. Pada setiap iterasi, algoritma secara acak memilih satu tetangga dari kubus saat ini menggunakan fungsi *generate_random_neighbor*, lalu menghitung *state value* dari tetangga ini dengan *count_of_match*. Jika nilai dari tetangga acak tersebut lebih tinggi daripada kubus saat ini, maka algoritma memperbarui kubus ke tetangga tersebut. Proses ini berlangsung hingga iterasi selesai, setelah itu algoritma mengembalikan kubus terbaik yang ditemukan.

```
def stochastic_hill_climbing(current_cube, iterations):
    count_iteration = 0
    current_score = count_of_match(current_cube)
    scores = [current_score]
    start_time = time.time()

    for i in range(iterations):
        count_iteration += 1
        neighbor = generate_random_neighbor(current_cube)
        new_score = count_of_match(neighbor)

        if new_score > current_score:
            current_cube, current_score = neighbor, new_score
            scores.append(current_score)

    end_time = time.time()
    duration = end_time - start_time

    return count_iteration, current_cube, current_score, scores,
           duration
```

2.2.3. Hill Climbing with Sideways Move

Algoritma *Sideways Move* ini dimulai dari kondisi awal kubus dan berusaha memperbaiki skornya secara iteratif dengan mempertimbangkan langkah-langkah *sideways*. Pada setiap iterasi, algoritma menghasilkan semua tetangga dari kubus saat ini dan mencari tetangga dengan skor terbaik menggunakan fungsi *count_of_match*. Jika tetangga terbaik memiliki skor lebih tinggi, kubus diperbarui dengan tetangga

tersebut, dan penghitung *sideways moves* diatur ulang ke nol. Namun, jika skor tetangga sama dengan kubus saat ini, algoritma masih bisa bergerak ke tetangga tersebut tetapi menghitungnya sebagai *sideways move*. Jika *sideways moves* mencapai batas yang ditentukan, algoritma berhenti untuk menghindari kebuntuan, lalu mengembalikan kubus terbaik yang ditemukan.

```
def sideways_move(current_cube, max_sideways_moves):
    count_iteration = 0
    current_score = count_of_match(current_cube)
    scores = [current_score]
    sideways_moves = 0
    start_time = time.time()

    while True:
        count_iteration += 1
        neighbors = generate_all_neighbors(current_cube)
        best_neighbor = None
        best_score = current_score

        for neighbor in neighbors:
            score = count_of_match(neighbor)
            if score > best_score:
                best_neighbor = neighbor
                best_score = score

        if best_neighbor is not None and best_score > current_score:
            current_cube = best_neighbor
            current_score = best_score
            scores.append(current_score)
            sideways_moves = 0
        elif best_neighbor is not None and best_score == current_score:
            sideways_moves += 1
            if sideways_moves >= max_sideways_moves:
                break
        else:
            break

    end_time = time.time()
    duration = end_time - start_time

    return count_iteration, current_cube, current_score, scores,
duration
```

2.2.4. Random Restart Hill Climbing

Algoritma *Random Restart Hill Climbing* ini menggabungkan *hill climbing* dengan beberapa *restart* untuk menghindari *local optima*. Pertama, algoritma menghitung skor awal kubus dan memulai proses perbaikan melalui sejumlah *restart* yang ditentukan. Pada setiap *restart*, algoritma mulai dari kondisi kubus terbaik yang ditemukan sejauh ini dan menjalankan iterasi *hill climbing* sebanyak iterations kali. Pada setiap iterasi, algoritma menghasilkan tetangga acak dari kubus saat ini dan menghitung count_of_match atau skornya. Jika skor tetangga lebih tinggi dari skor saat ini, algoritma memperbarui kubus dengan tetangga tersebut. Setelah semua iterasi selesai, jika skor yang dihasilkan lebih tinggi dari skor terbaik sebelumnya, algoritma memperbarui kubus terbaik yang ditemukan. Proses ini berakhir setelah semua restart selesai atau sudah menemukan global optimum, lalu algoritma mengembalikan kubus terbaik yang ditemukan.

```
def random_restart_hill_climbing(current_cube, iterations,
max_restarts):
    count_restart = 0
    best_cube = current_cube
    best_score = count_of_match(current_cube)
    best_scores = [best_score]
    start_time = time.time()

    for i in range(max_restarts):
        if best_score == 109:
            break
        count_restart += 1
        current_cube = best_cube
        current_score = best_score
        scores = [current_score]

        for j in range(iterations):
            neighbor = generate_random_neighbor(current_cube)
            new_score = count_of_match(neighbor)

            if new_score > current_score:
                current_cube, current_score = neighbor, new_score
                scores.append(current_score)

        if current_score > best_score:
            best_cube, best_score = current_cube, current_score
            best_scores = scores

    end_time = time.time()
```

```

duration = end_time - start_time

return count_restart, best_cube, best_score, best_scores,
duration

```

2.2.5. Simulated Annealing

Algoritma *Simulated Annealing* ini menggabungkan stochastic *hill climbing* dan *random walk* untuk menghindari *local optima* dengan memperbolehkan perpindahan ke *worse neighbor* yang memenuhi syarat *probability* tertentu. Pada setiap iterasi, algoritma ini akan secara acak memilih satu tetangga dari kubus saat ini menggunakan fungsi *generate_random_neighbor*, lalu menghitung *state value* dari tetangga ini dengan *count_of_match*. Jika nilai dari tetangga acak tersebut lebih tinggi daripada kubus saat ini, maka algoritma memperbarui kubus ke tetangga tersebut. Di sisi lain, jika nilai dari tetangga acak tersebut lebih rendah daripada kubus saat ini, algoritma akan menghitung terlebih dahulu nilai probabilitas perpindahan dengan rumus $e^{\frac{\Delta E}{T}}$, dimana ΔE adalah selisih nilai *state value* tetangga dan kubus saat ini sedangkan T adalah nilai suhu yang akan terus turun seiring bertambahnya iterasi. Jika nilai probabilitas perpindahan tersebut lebih dari 0.5, maka algoritma akan memperbarui kubus ke tetangga tersebut. Proses ini berlangsung hingga iterasi selesai, setelah itu algoritma mengembalikan kubus terbaik yang ditemukan.

```

def simulated_annealing(current_cube, initial_temp, cooling_rate,
max_iter):
    current_score = count_of_match(current_cube)
    temperature = initial_temp
    scores = [current_score]
    probabilities = []
    count_stuck = 0
    start = time.time()

    for i in range(max_iter):
        neighbor = generate_random_neighbor(current_cube)
        neighbor_score = count_of_match(neighbor)

        delta_score = neighbor_score - current_score

        if delta_score > 0:
            current_cube = neighbor
            current_score = neighbor_score
            scores.append(current_score)
        else:
            probability = math.exp(-delta_score / temperature)
            if random.random() < probability:
                current_cube = neighbor
                current_score = neighbor_score
                scores.append(current_score)
            else:
                count_stuck += 1
                if count_stuck == max_iter:
                    break

        temperature *= cooling_rate

```

```

        probabilities.append(1)
    else:
        probability = math.exp(delta_score / temperature)
        probabilities.append(probability)
        count_stuck += 1

    if probability > 0.5:
        current_cube = neighbor
        current_score = neighbor_score
        scores.append(current_score)

    temperature *= cooling_rate

end = time.time()
duration = end - start

return max_iter, current_cube, current_score, scores,
probabilities, count_stuck, duration

```

2.2.6. Genetic Algorithm

Genetic Algorithm (GA) adalah salah satu metode optimasi dan pencarian berdasarkan prinsip seleksi alam dari teori evolusi biologis. Dalam Genetic Algorithm, solusi optimal untuk suatu masalah dicari melalui proses yang menyerupai evolusi makhluk hidup, yaitu seleksi, crossover, dan mutasi. Algoritma ini sangat berguna dalam mencari solusi optimal untuk masalah yang memiliki ruang solusi yang sangat luas dan kompleks.

Genetic Algorithm bekerja dengan cara membangkitkan *K state* acak di awal. Lalu mengkombinasikan dua *parent state* untuk mendapatkan *successor*. Setiap *state* dinilai berdasarkan fungsi objective yang biasa disebut *fitness function*. Pada Genetic Algorithm *feasible* atau tidaknya algoritma ini tergantung dengan fungsi *crossover* dan jumlah populasi dan iterasi yang digunakan.

1. Kelas GeneticAlgorithm

```

class GeneticAlgorithm:
    def __init__(self, crossover_func="randomized",
population_size=100, iterations=1000, shuffle=True):
        """
        Inisialisasi solver GA.

    Parameters:
        crossover_func (str): Jenis crossover function yang akan

```

```
digunakan ("randomized" atau "probabilistic").
    population_size (int): Jumlah individu dalam populasi.
    iterations (int): Jumlah iterasi algoritma genetika.
    shuffle (bool): Apakah populasi baru akan diacak setelah
setiap iterasi.
"""
self.n = 5
self.magic_number = 315
self.crossover_func = crossover_func
self.shuffle = shuffle
self.population_size = population_size
self.iterations = iterations
self.initial_population =
self.generate_initial_population(population_size)

def generate_initial_population(self, size=100) -> list:
"""
Membuat populasi awal dengan elemen unik.

Parameters:
    size (int): Jumlah individu dalam populasi.

Returns:
    list: Populasi awal yang terdiri dari individu-individu.
"""
population = []
for _ in range(size):
    individual = np.random.permutation(self.n ** 3) + 1
    population.append(individual)
return population

def find_best_individual(self, population):
"""
Menemukan individu terbaik dalam populasi berdasarkan skor.

Parameters:
    population (list): Daftar individu dalam populasi.

Returns:
    np.array: Individu dengan skor tertinggi dalam populasi.
"""
scores = self.count_score(population)
best_index = np.argmax(scores)
return population[best_index]

def count_score(self, population) -> list:
"""
Menghitung skor untuk setiap individu dalam populasi.
```

```

Parameters:
    population (list): Daftar individu dalam populasi.

Returns:
    list: Daftar skor untuk setiap individu.
"""
scores = []
for individual in population:
    scores.append(count_of_match(individual))
return scores

def selection(self, population, scores, k=1) -> list:
"""
Memilih individu dari populasi dengan weighted selection.

Parameters:
    population (list): Daftar individu dalam populasi.
    scores (list): Daftar skor untuk setiap individu.
    k (int): Jumlah individu yang dipilih.

Returns:
    list: Individu yang terpilih dari populasi.
"""
weights = scores if np.sum(scores) > 0 else None
selected_population = random.choices(population,
weights=weights, k=k)
return selected_population[0]

def mutate(self, individual):
"""
Melakukan mutasi pada individu dengan menukar dua elemen acak.

Parameters:
    individual (np.array): Individu yang akan dimutasi.

Returns:
    np.array: Individu yang telah dimutasi.
"""
idx1, idx2 = random.sample(range(self.n ** 3), 2)
individual[idx1], individual[idx2] = individual[idx2],
individual[idx1]
return individual

def genetic_algorithm(self):
"""
Menjalankan algoritma genetika untuk menemukan solusi terbaik.

```

```

    Returns:
        dict: Informasi konfigurasi akhir, indeks terbaik, dan
skor iterasi.
    """
best_score = 0
population = self.initial_population
iteration_scores = []
for iter in range(self.iterations):
    scores = self.count_score(population)
    individual_with_best_score = np.argmax(scores)
    best_score = scores[individual_with_best_score]
    iteration_scores.append((best_score,
individual_with_best_score))
    # if (iter + 1) % 10 == 0:
    #     print(f"Best score in {iter + 1}-iterations:
{best_score} state ke: {individual_with_best_score}")

new_population = []
with ThreadPoolExecutor() as executor:
    futures = [
        executor.submit(self.create_children, population,
scores, self.crossover_func, iter, self.iterations)
        for _ in range(self.population_size // 2)
    ]
    for future in futures:
        child1, child2 = future.result()
        new_population.extend([child1, child2])

if self.shuffle:
    random.shuffle(new_population)
population = new_population

final_solution_index = np.argmax(scores)
final_solution = population[final_solution_index]

solution = {
    "config": final_solution,
    "index": final_solution_index,
    "iteration_scores": iteration_scores,
    "best_score" : best_score
}
return solution

def create_children(self, population, scores, crossover_func,
iter, total_iteration):
"""
Membuat anak-anak dari dua induk menggunakan crossover dan
mutasi.

```

```

Parameters:
    population (list): Daftar individu dalam populasi.
    scores (list): Daftar skor untuk setiap individu.
    crossover_func (str): Jenis crossover yang akan digunakan.
    iter (int): Iterasi saat ini.
    total_iteration (int): Total iterasi yang akan dijalankan.

Returns:
    tuple: Dua anak yang dihasilkan dari crossover dan mutasi.
"""
parent1 = self.selection(population, scores)
parent2 = self.selection(population, scores)

if crossover_func == "randomized":
    child1, child2 =
custom_segment_preserving_crossover(parent1, parent2)
else:
    child1, child2 =
custom_probabilistic_segment_preserving_crossover(parent1, parent2,
iterations=total_iteration, current_iteration=iter)

if random.random() < 0.1:
    child1 = self.mutate(child1)
if random.random() < 0.1:
    child2 = self.mutate(child2)

return child1, child2

```

2. Crossover Function

Crossover function yang kami gunakan merupakan *custom crossover*. *Crossover function* ini bekerja dengan cara mempertahankan segmen yang *match* dengan 315 lalu sisanya didapat dari merandom dari parent yang lainnya. Misal parent1 dan parent2, child1 akan mempertahankan segmen yang *match* dengan 315 dari parent1 lalu sisanya akan diisi dengan urutan parent2 yang tidak ada di parent1. Sebenarnya *crossover function* ini bisa ditingkatkan lagi jadi mempertahankan segmen yang *match* di parent1 dan parent2 di kedua child dan mengisinya dengan sisa di *parent* yang nomornya berbeda dengan *child*. Kelebihan dari metode ini adalah proses konvergensi yang cepat, tetapi kelemahannya adalah sering kali konvergen di sekitar 40 an *match*.

```
def custom_segment_preserving_crossover(parent1, parent2):
```

```

child1 = [-1] * len(parent1)
child2 = [-1] * len(parent2)

protected_indices_p1 = find_protected_indices(parent1)
protected_indices_p2 = find_protected_indices(parent2)

for i in protected_indices_p1:
    child1[i] = parent1[i]
for i in protected_indices_p2:
    child2[i] = parent2[i]

unused_elements_child1 = [parent2[i] for i in
range(len(parent2)) if parent2[i] not in child1]
unused_elements_child2 = [parent1[i] for i in
range(len(parent1)) if parent1[i] not in child2]

fill_randomly(child1, unused_elements_child1)
fill_randomly(child2, unused_elements_child2)

return child1, child2

def fill_randomly(child, unused_elements):
    """Mengisi elemen yang kosong di anak dengan elemen tersisa."""
    # random.shuffle(unused_elements)
    missing_elements = iter(unused_elements)

    for i in range(len(child)):
        if child[i] == -1:
            child[i] = next(missing_elements)

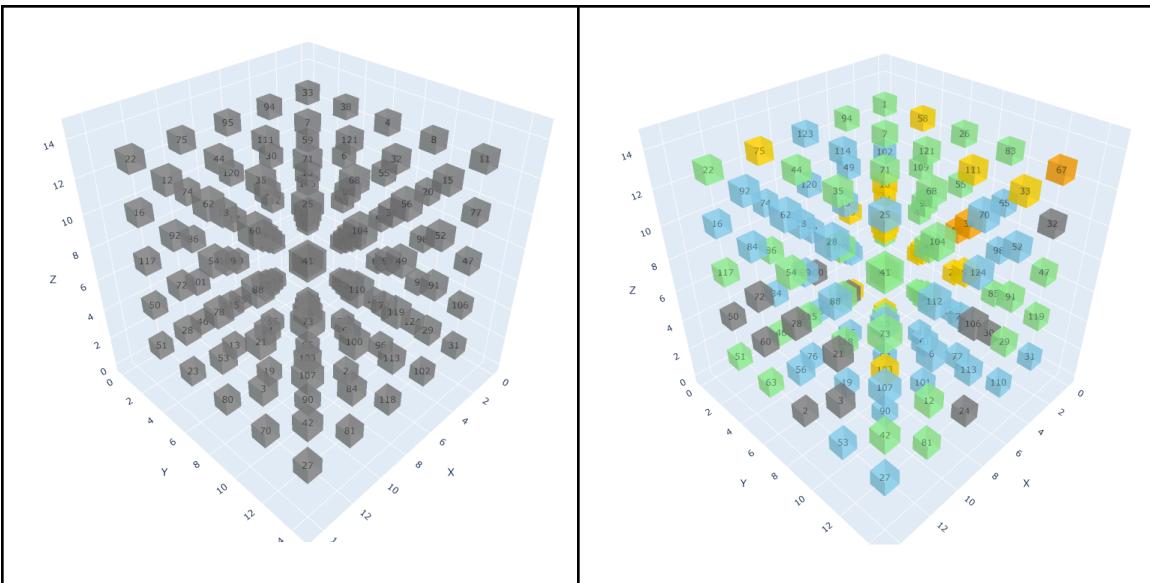
```

2.3. Hasil Eksperimen dan Analisis

2.3.1. Steepest Ascent Hill Climbing

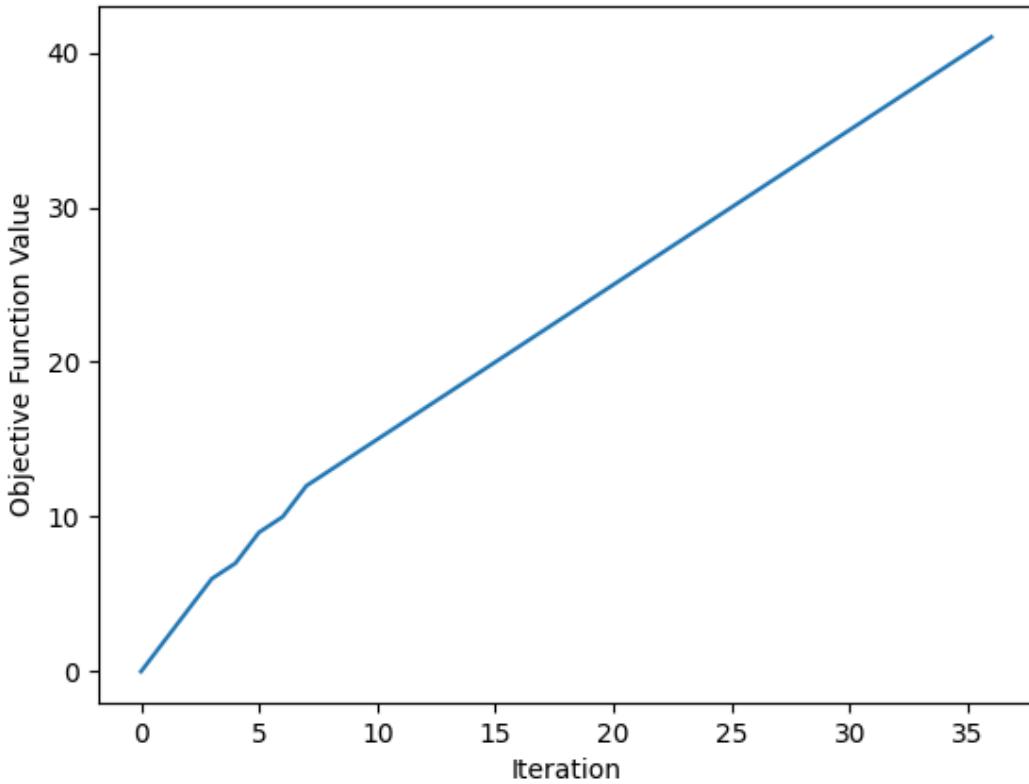
- Iterasi 1

| | |
|------------|-------------|
| State Awal | State Akhir |
|------------|-------------|



Plot Nilai Objective Function

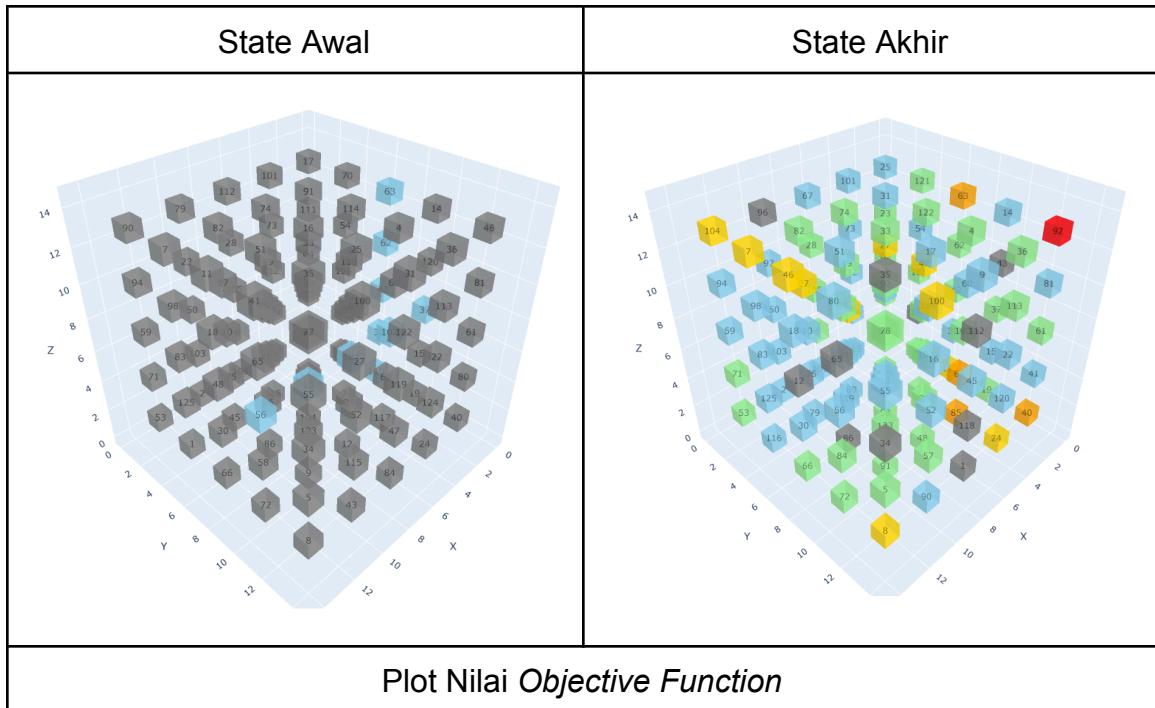
Steepest Ascent Progress

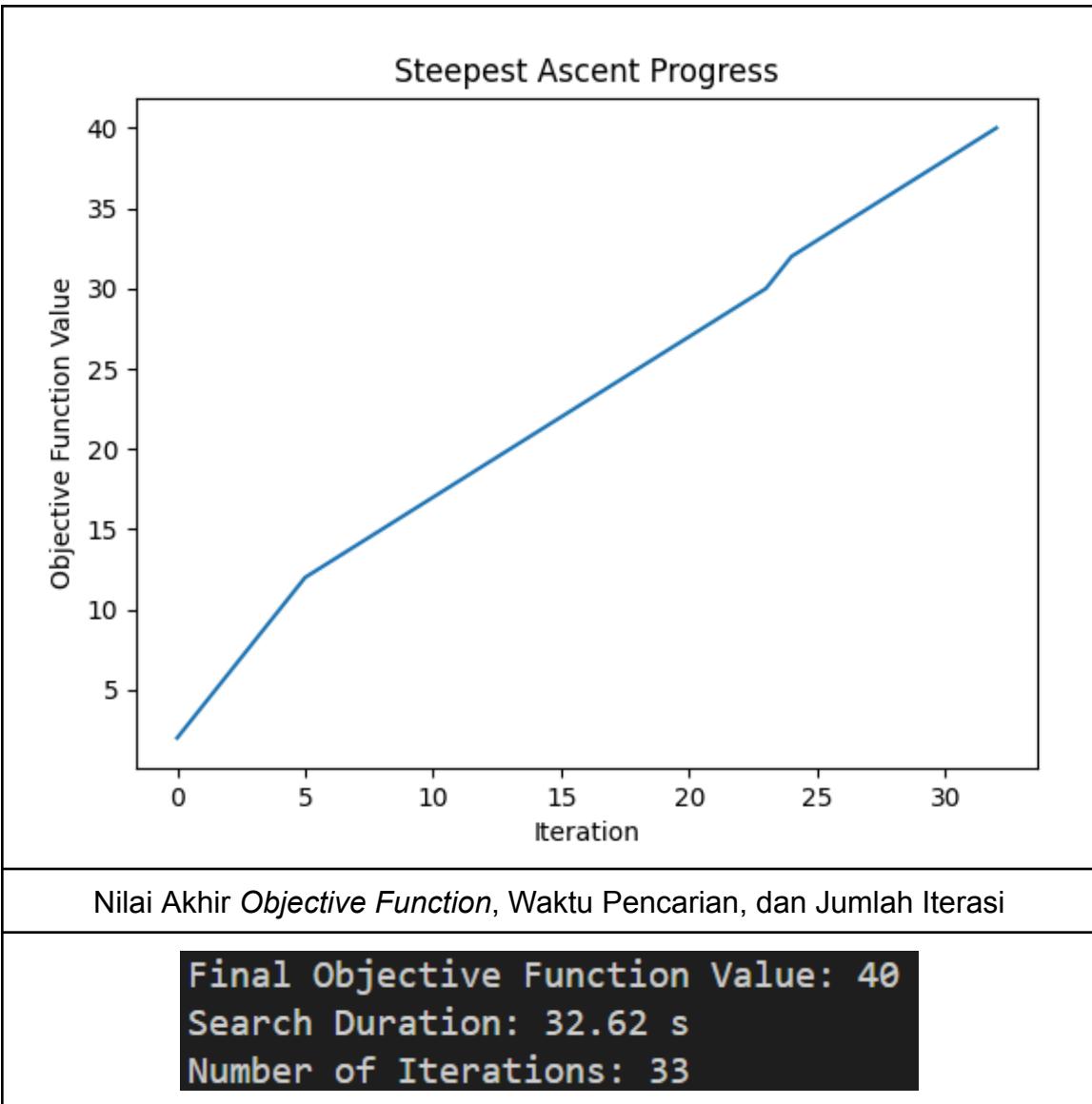


Nilai Akhir Objective Function, Waktu Pencarian, dan Jumlah Iterasi

Final Objective Function Value: 41
Search Duration: 26.63 s
Number of Iterations: 37

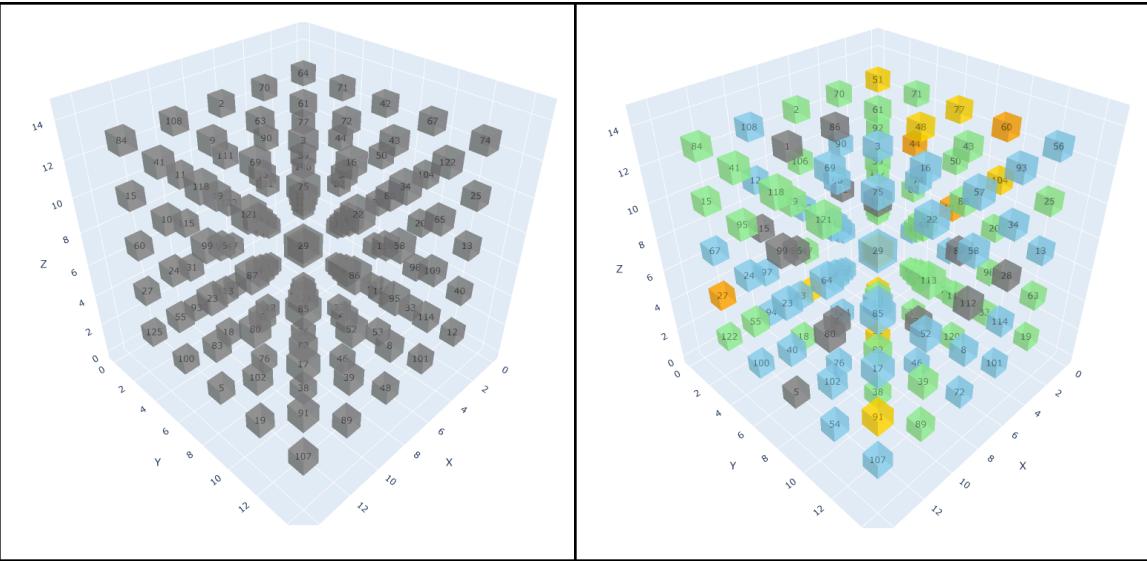
- Iterasi 2





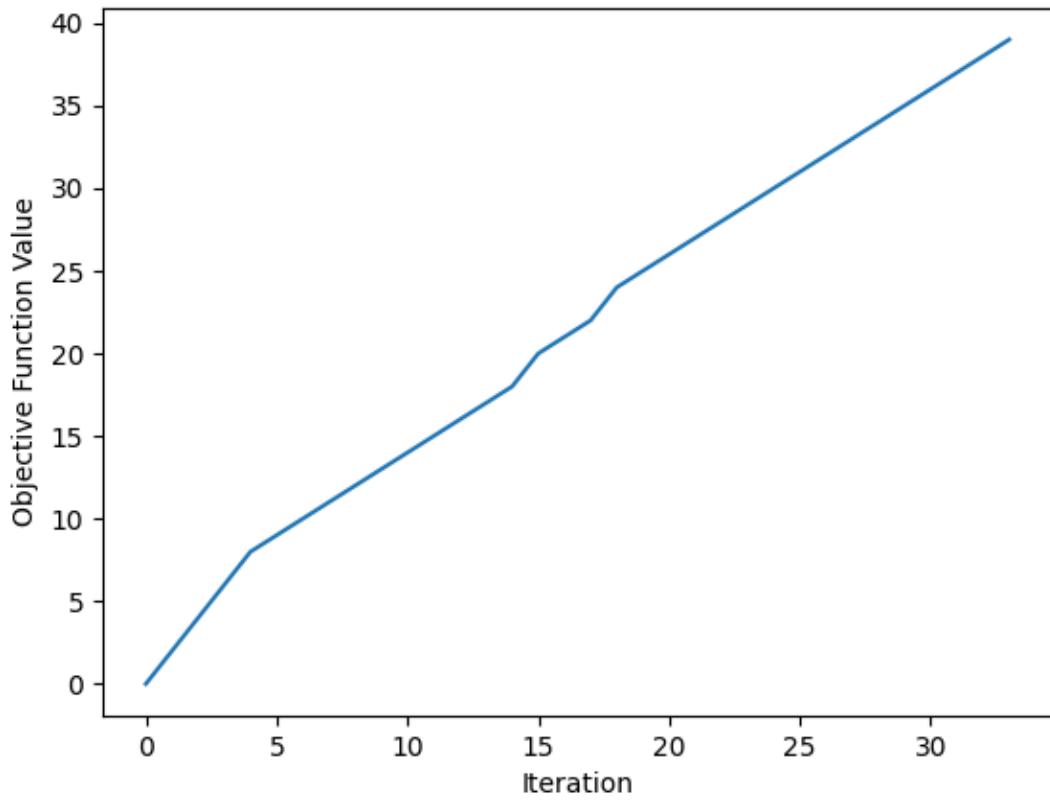
- Iterasi 3

| | |
|------------|-------------|
| State Awal | State Akhir |
|------------|-------------|



Plot Nilai Objective Function

Steepest Ascent Progress



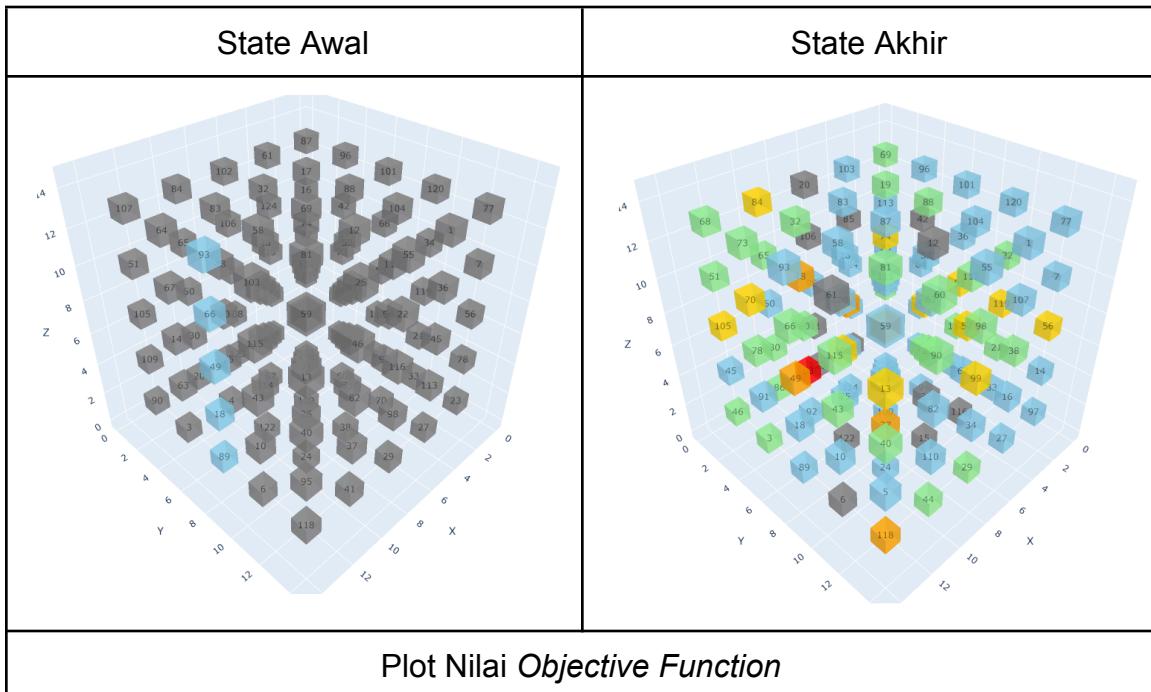
Nilai Akhir Objective Function, Waktu Pencarian, dan Jumlah Iterasi

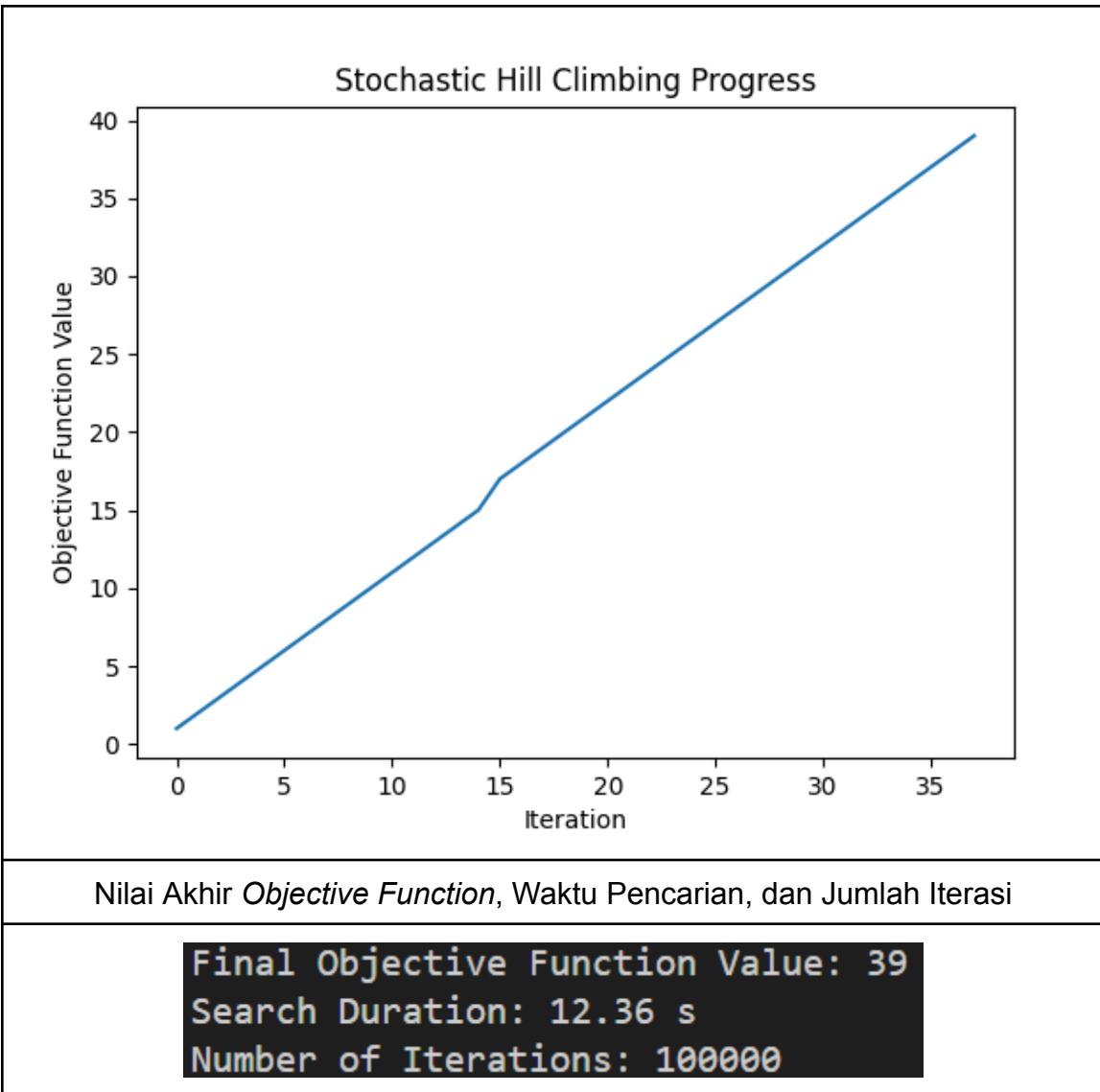
| |
|------------------------------------|
| Final Objective Function Value: 39 |
| Search Duration: 34.01 s |
| Number of Iterations: 34 |

Dapat dilihat bahwa nilai akhir *Objective Function Value* dari ketiga percobaan cenderung sama, yaitu 41, 40, dan 39 yang mana cukup konsisten dan menunjukkan bahwa *steepest ascent* memberikan hasil yang relatif stabil di setiap percobaan. Variasi ini terjadi karena algoritma mengikuti jalur pendakian terdekat. Kemudian, waktu yang didapat juga relatif cepat sekitar 26 hingga 34 detik. Hal ini menunjukkan bahwa algoritma ini dapat mendekati solusi optimal (109), namun tidak mencapai global optima. Karena *steepest ascent* cenderung terjebak pada local optima dan kurang mampu menjelajahi solusi yang lebih jauh untuk mencari global optima.

2.3.2. Stochastic Hill Climbing

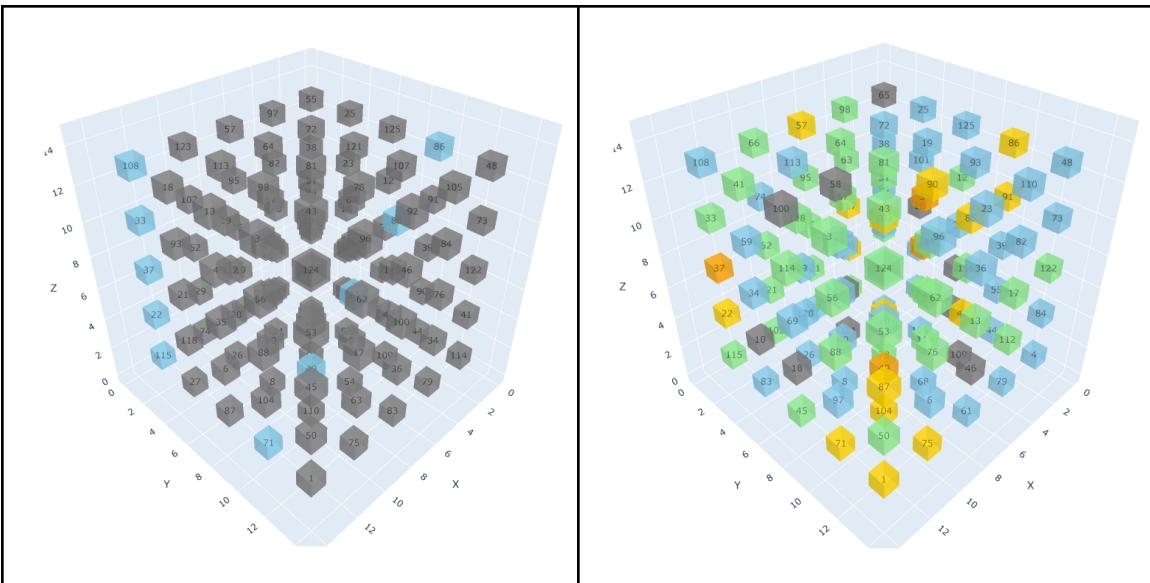
- Iterasi 1





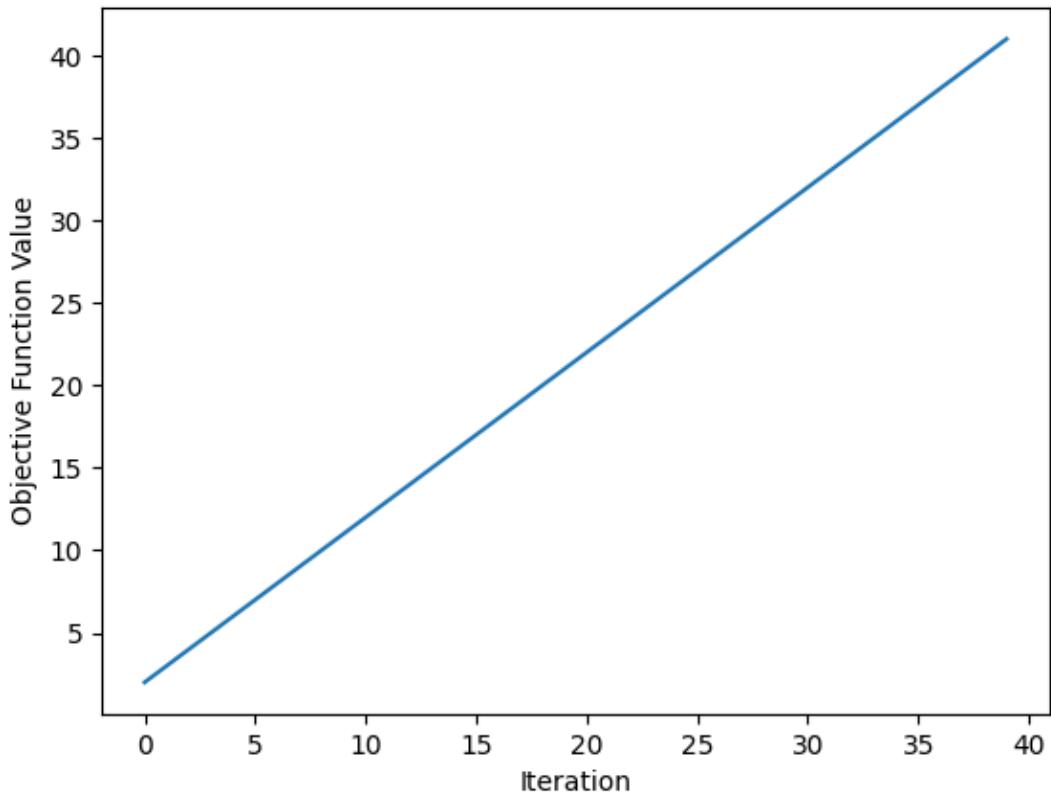
- Iterasi 2

| | |
|------------|-------------|
| State Awal | State Akhir |
|------------|-------------|



Plot Nilai Objective Function

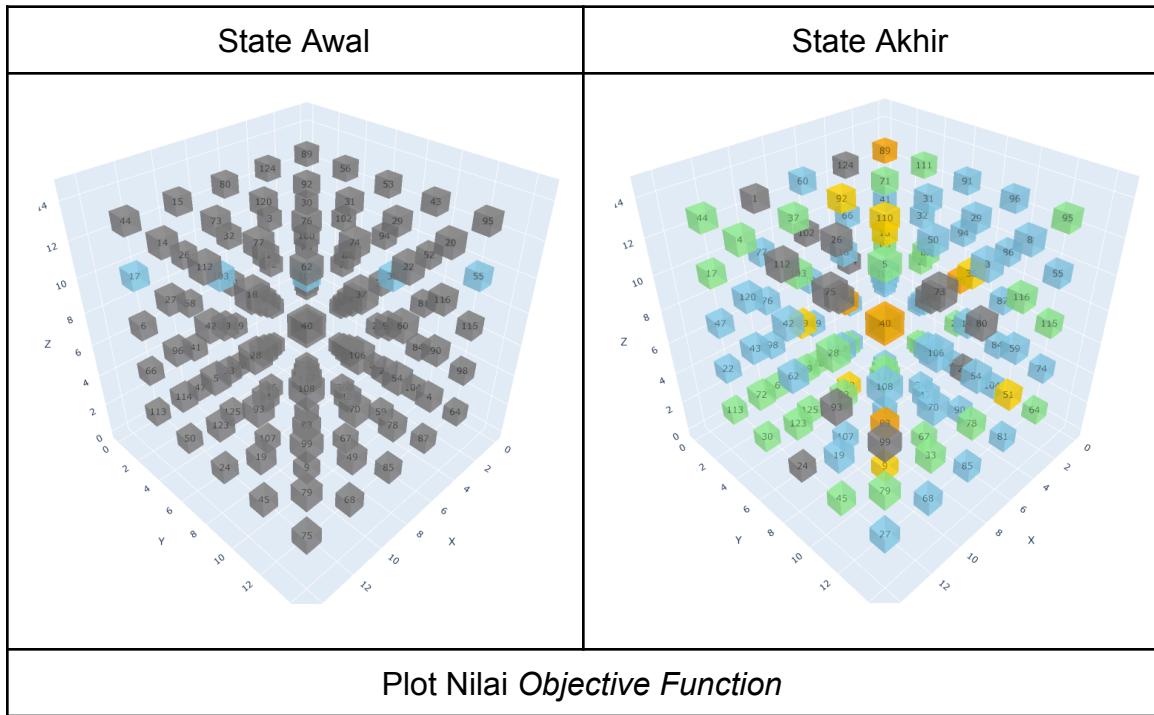
Stochastic Hill Climbing Progress

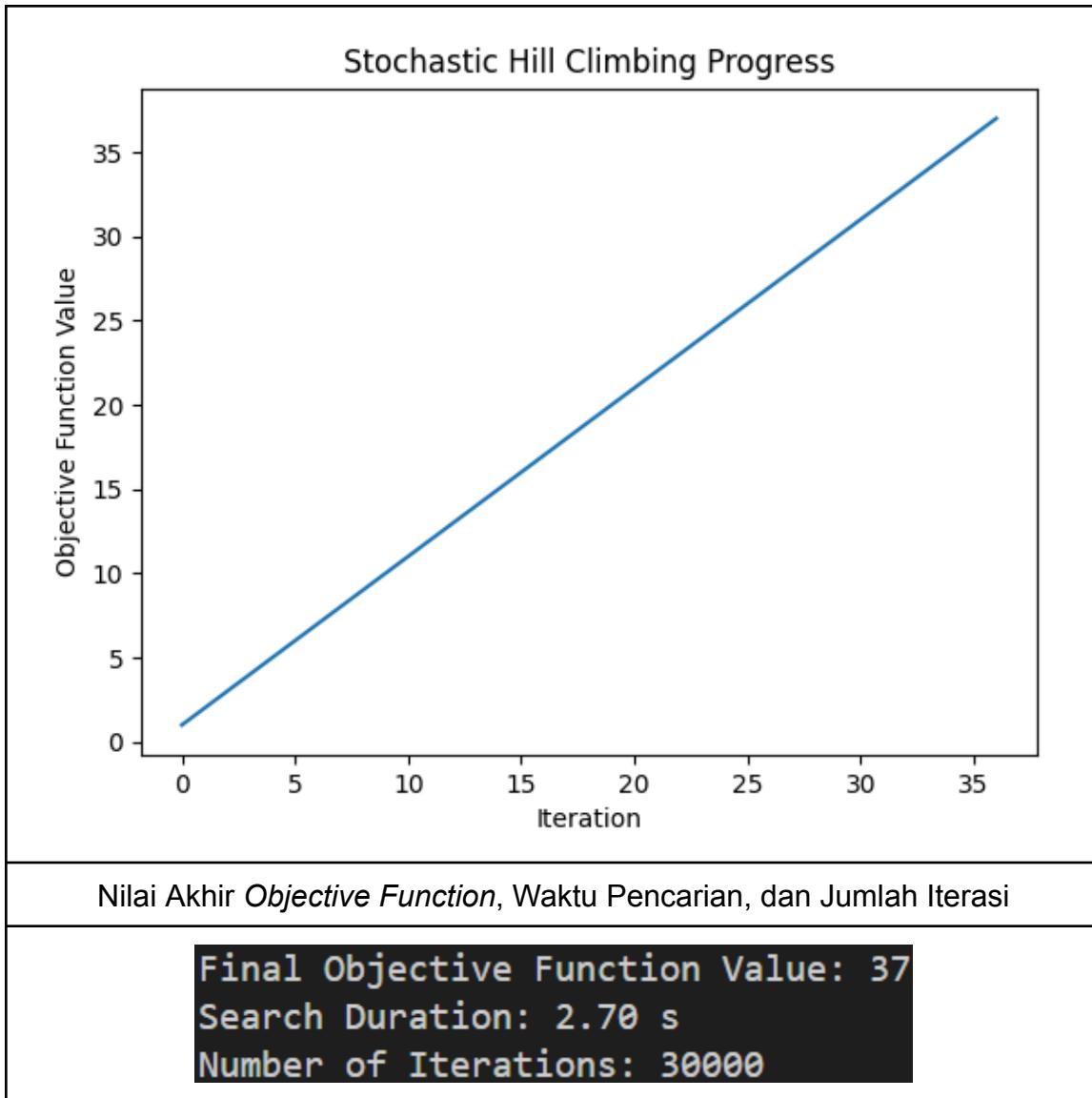


Nilai Akhir Objective Function, Waktu Pencarian, dan Jumlah Iterasi

Final Objective Function Value: 41
Search Duration: 4.37 s
Number of Iterations: 50000

- Iterasi 3

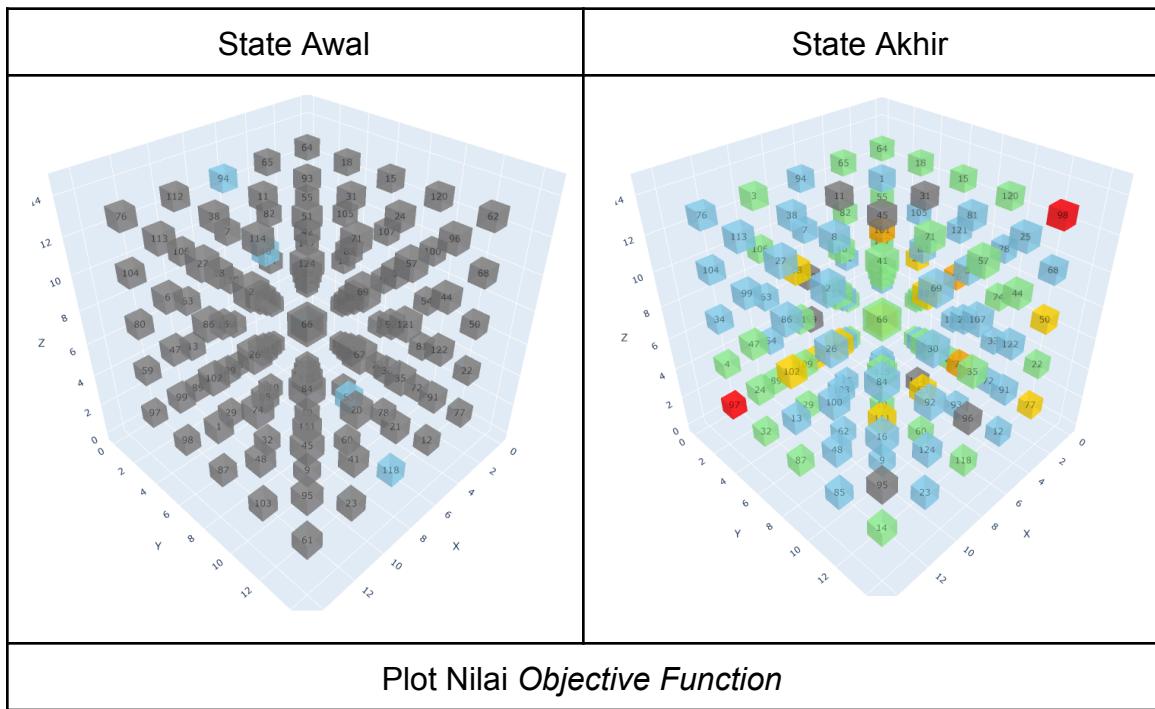


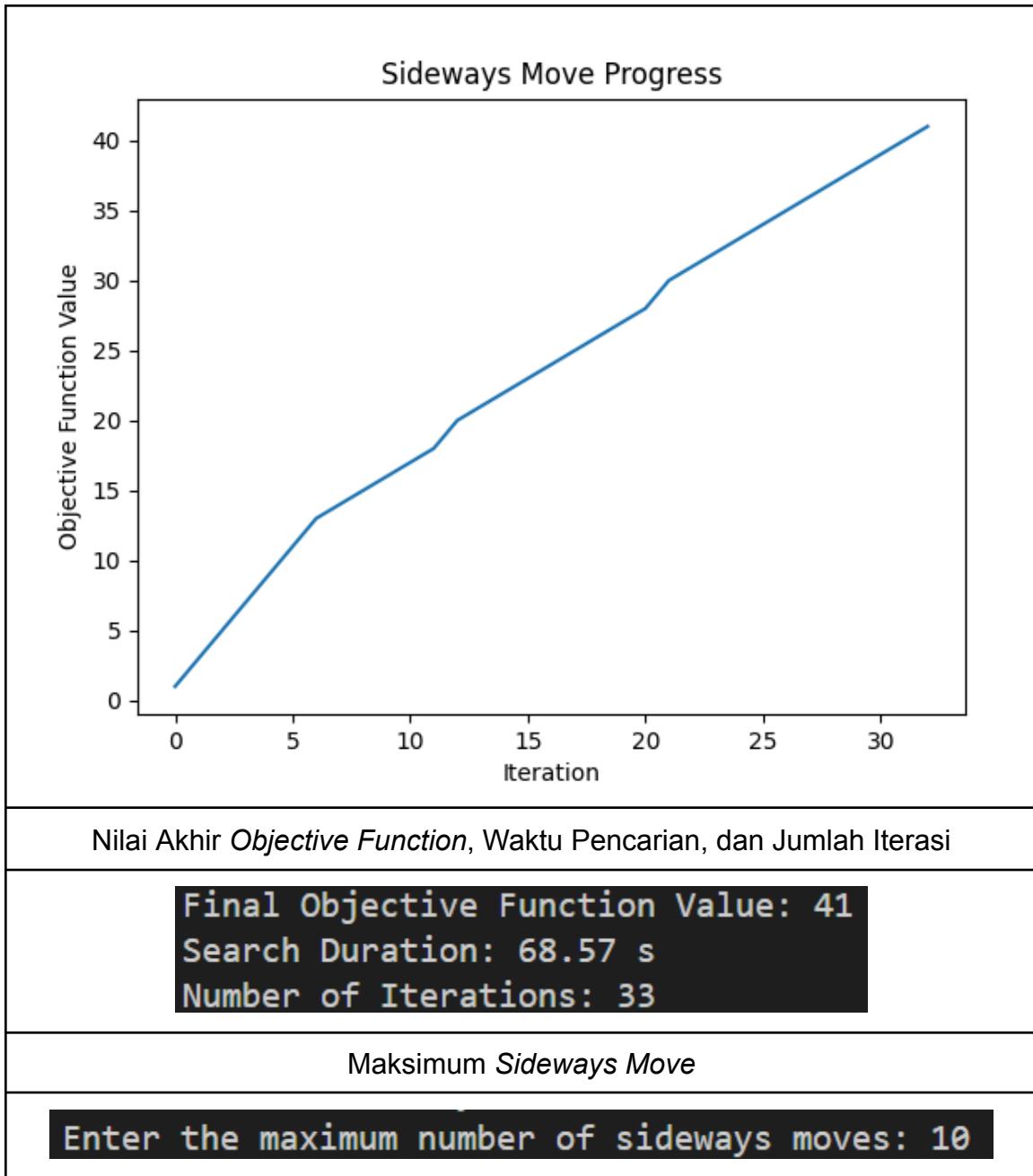


Algoritma *stochastic hill climbing* mampu mencapai nilai *objective function* hingga 39, 41, dan 37. Dari ketiga eksperimen yang memberikan perbedaan di jumlah iterasi, yaitu 30000, 50000, dan 100000, bahwa hasil yang didapat atau nilai *objective function* cenderung konsisten dan berhenti di local optima. Namun, jumlah iterasi juga berpengaruh terhadap nilai *objective function* akhir yang diperoleh. Karena jika terlalu kecil jumlah iterasi yang digunakan, akan menyebabkan nilai *objective function* nya juga kecil. Kemudian, algoritma *stochastic* memiliki waktu pencarian yang sangat cepat yaitu sekitar 2 hingga 12 detik. Karena *stochastic* sifatnya yang tidak secara sistematis mengevaluasi semua kemungkinan tetangga di setiap langkah, namun hanya memilih satu tetangga secara acak dan bergerak ke arah tersebut jika nilai tetangga lebih baik dari posisi saat ini.

2.3.3. Hill Climbing with Sideways Move

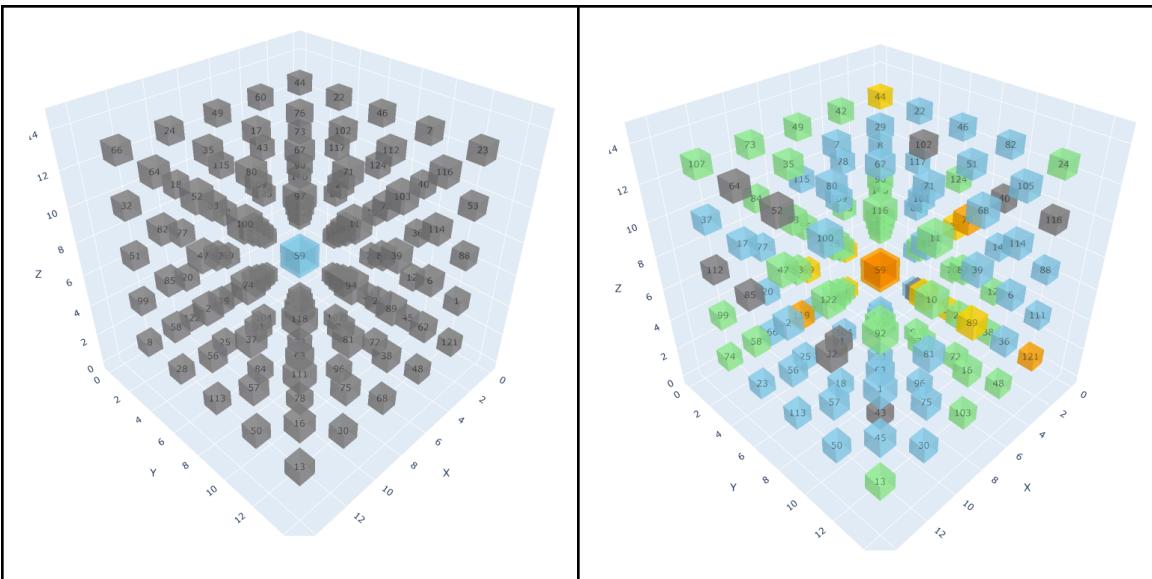
- Iterasi 1





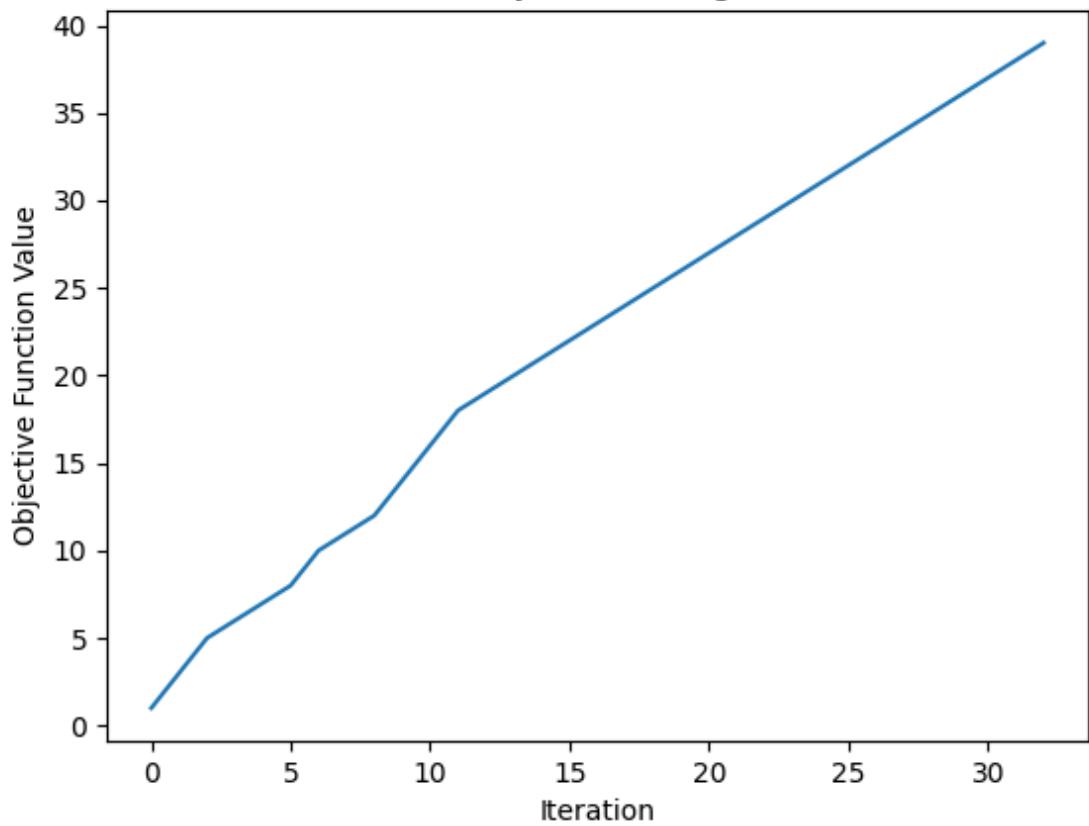
- Iterasi 2

| | |
|------------|-------------|
| State Awal | State Akhir |
|------------|-------------|



Plot Nilai *Objective Function*

Sideways Move Progress



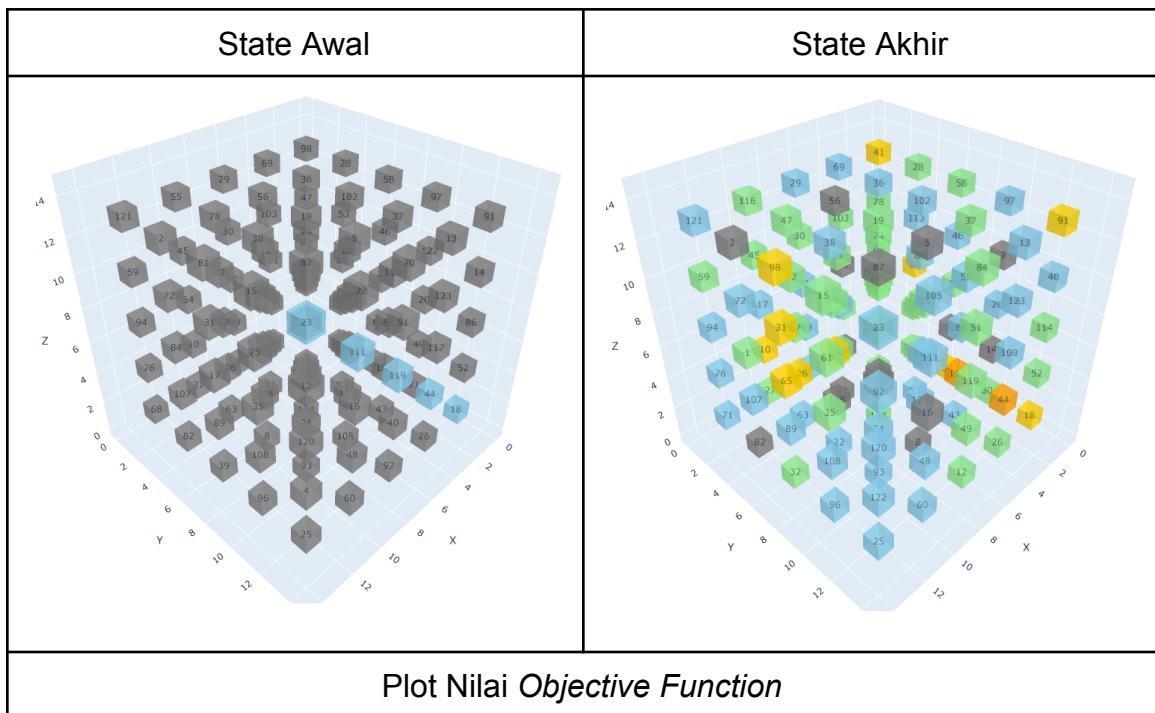
Nilai Akhir *Objective Function*, Waktu Pencarian, dan Jumlah Iterasi

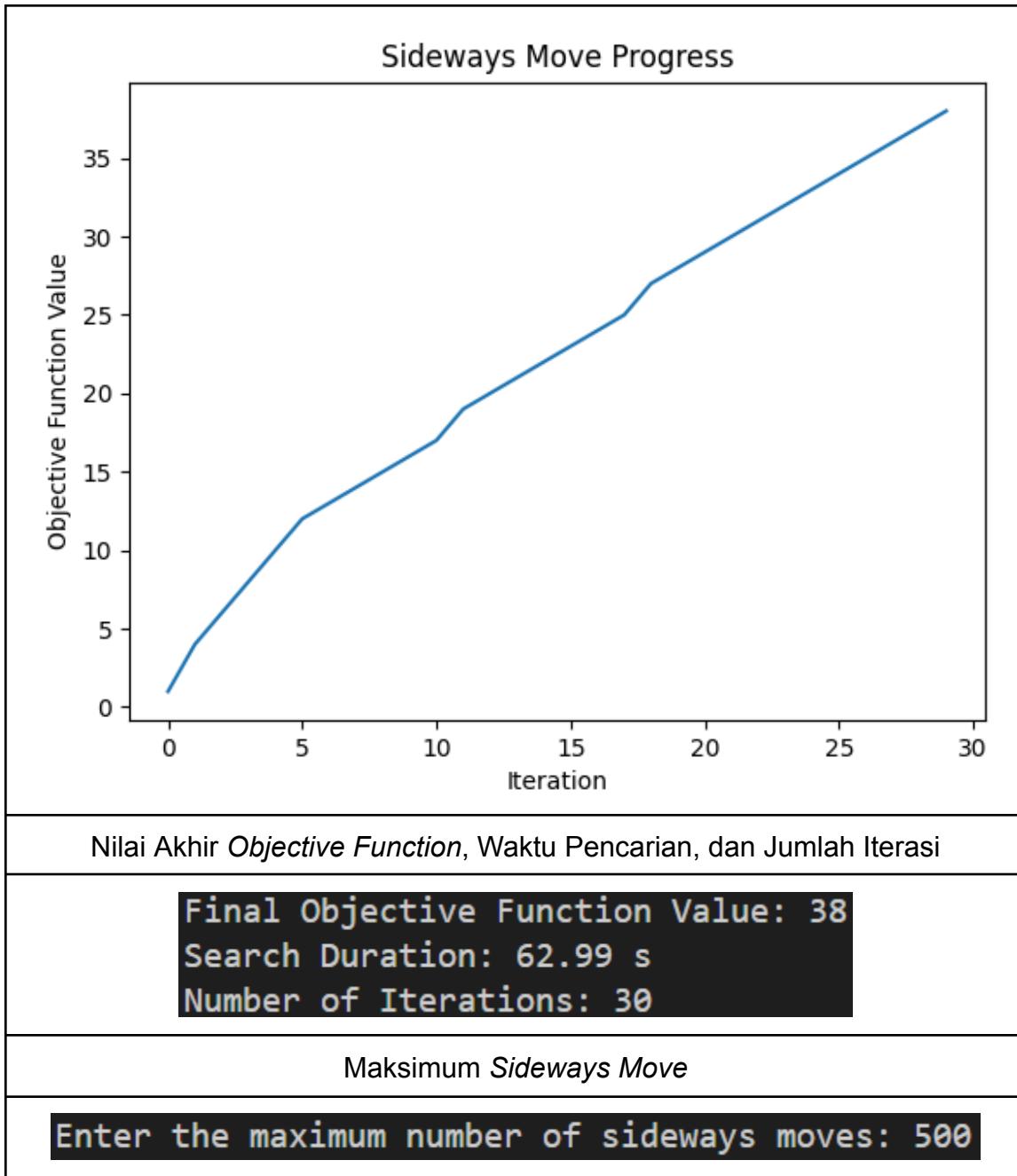
Final Objective Function Value: 39
Search Duration: 68.62 s
Number of Iterations: 33

Maksimum Sideways Move

Enter the maximum number of sideways moves: 50

- Iterasi 3



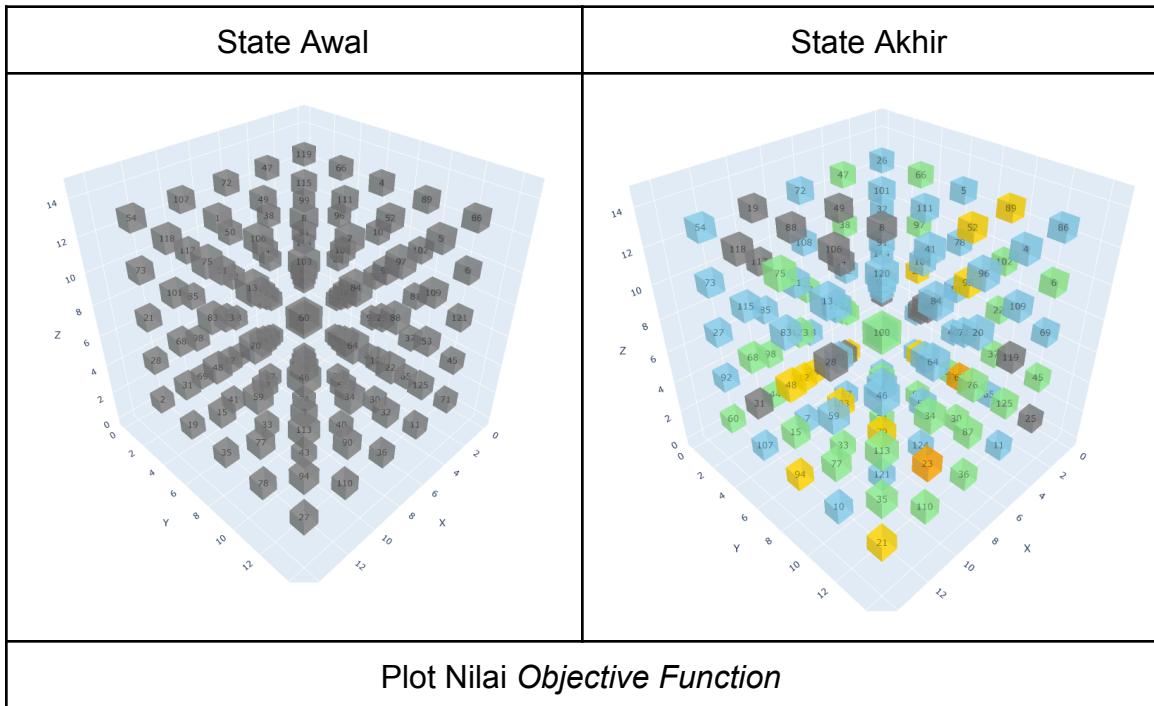


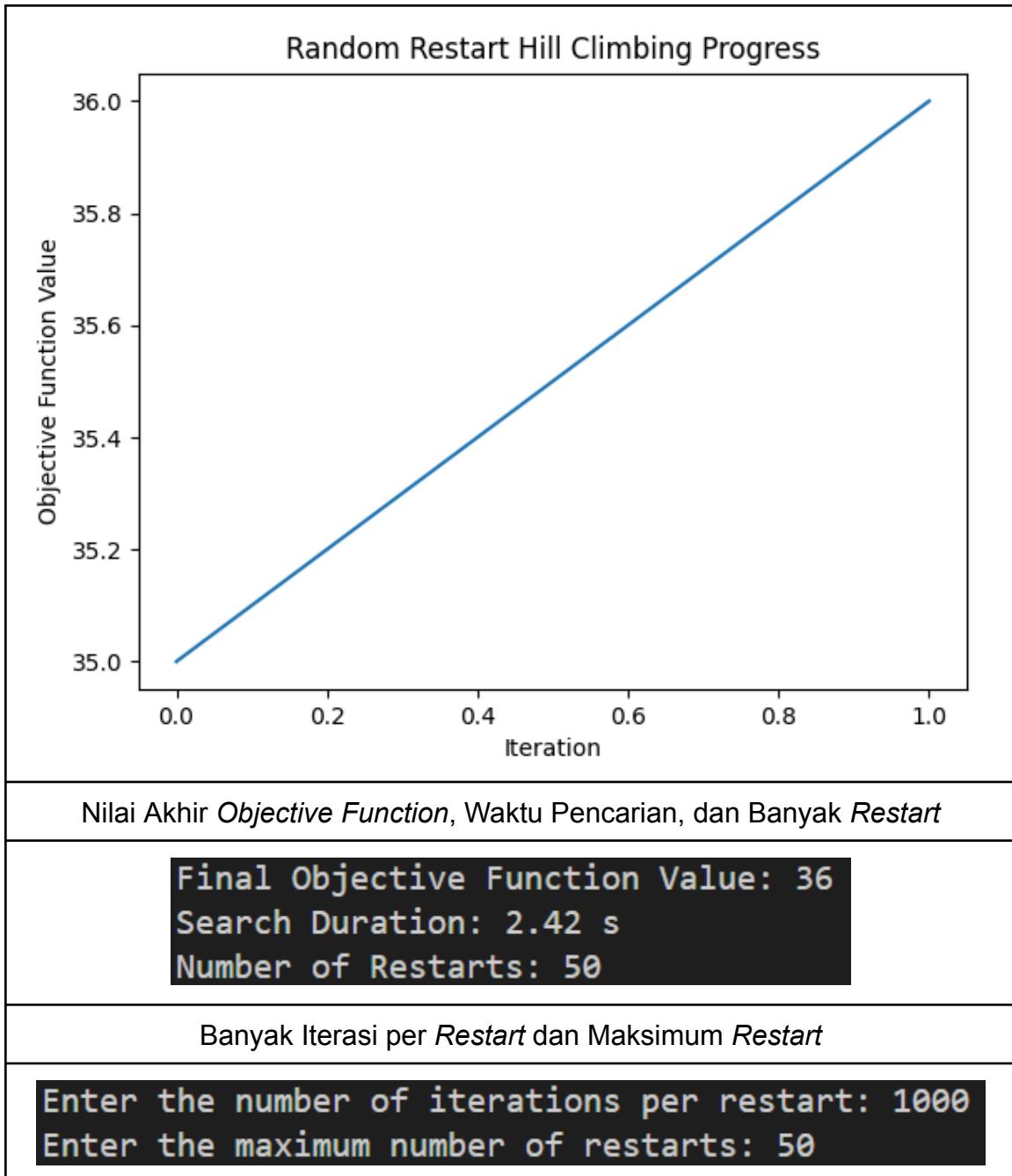
Pada eksperimen ini, dibuat variasi jumlah *sideways moves* yaitu 10, 50, dan 500. Namun, walaupun jumlah *sideways moves* nya sangat bervariasi dari yang kecil hingga sangat besar, nilai final *objective function* nya juga hampir sama, yaitu sekitar 41, 39, 38. Sehingga jumlah *sideways move* tidak terlalu berpengaruh terhadap hasil yang didapatkan. Hasil yang didapatkan hampir mirip dengan *steepest ascent*, karena algoritma *sideways moves* memperbolehkan pergerakan jika nilai tetangga dan nilai *current* memiliki skor yang sama atau kondisi *plateau*. Waktu pencarian yang dibutuhkan dalam algoritma ini sangat lama, yaitu sekitar 1 menit, sehingga kurang efisien dalam melakukan pencarian, terlebih lagi memiliki skor

yang sama dengan algoritma lain seperti *steepest* dan *stochastic* yang mana mereka memiliki waktu yang lebih cepat.

2.3.4. Random Restart Hill Climbing

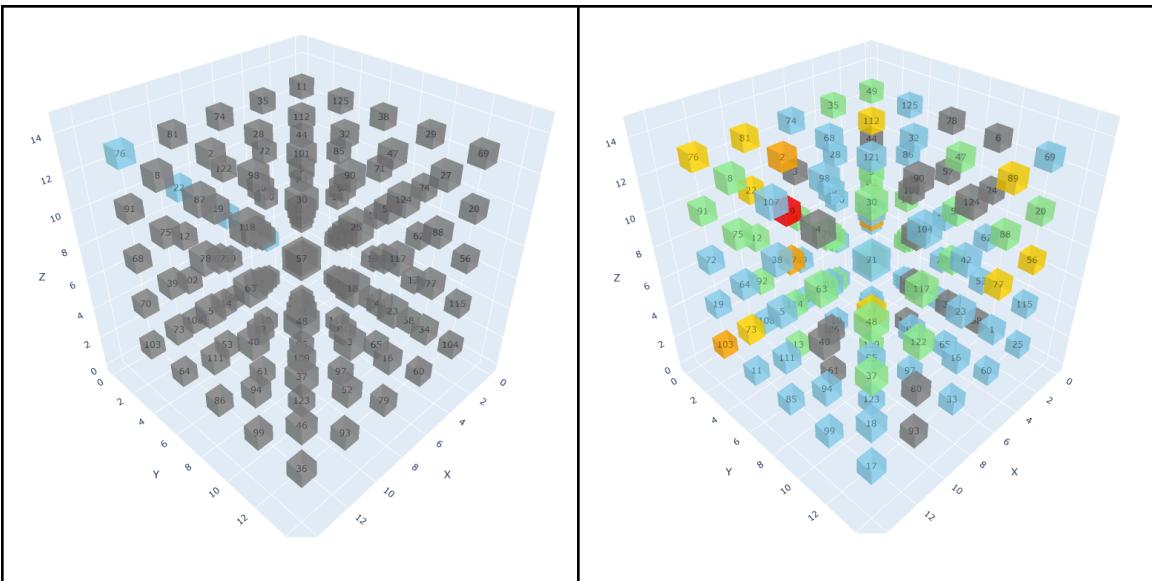
- Iterasi 1





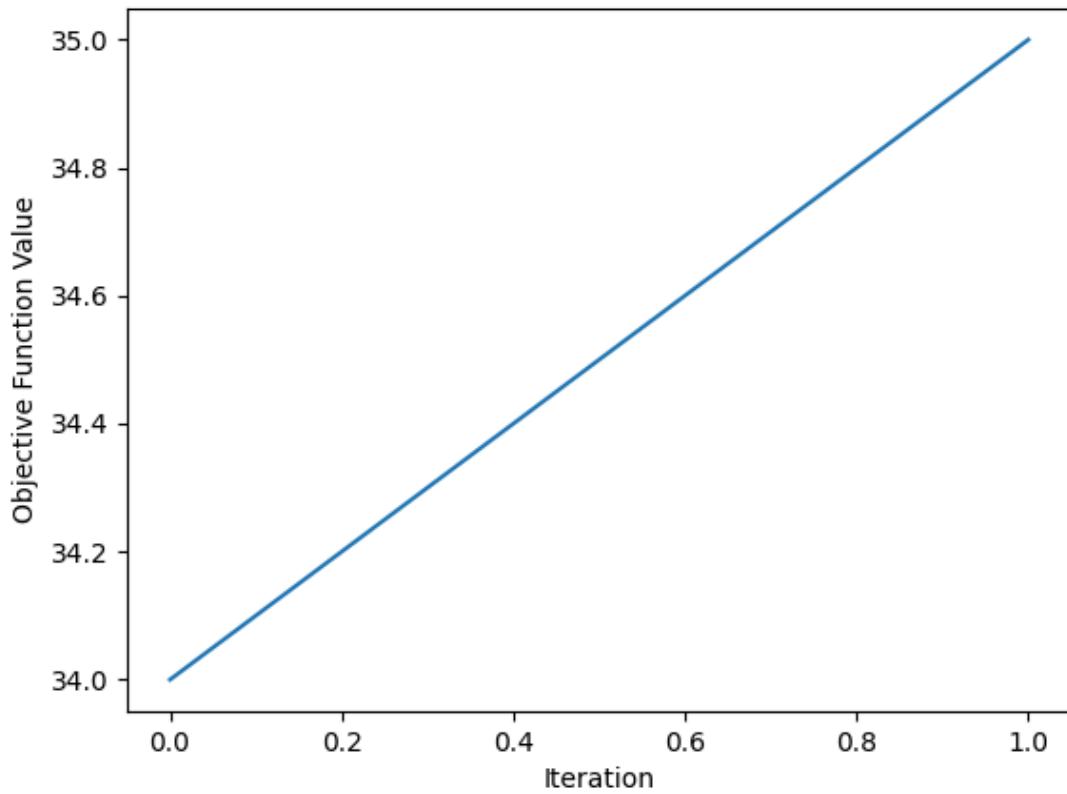
- Iterasi 2

| | |
|------------|-------------|
| State Awal | State Akhir |
|------------|-------------|



Plot Nilai *Objective Function*

Random Restart Hill Climbing Progress



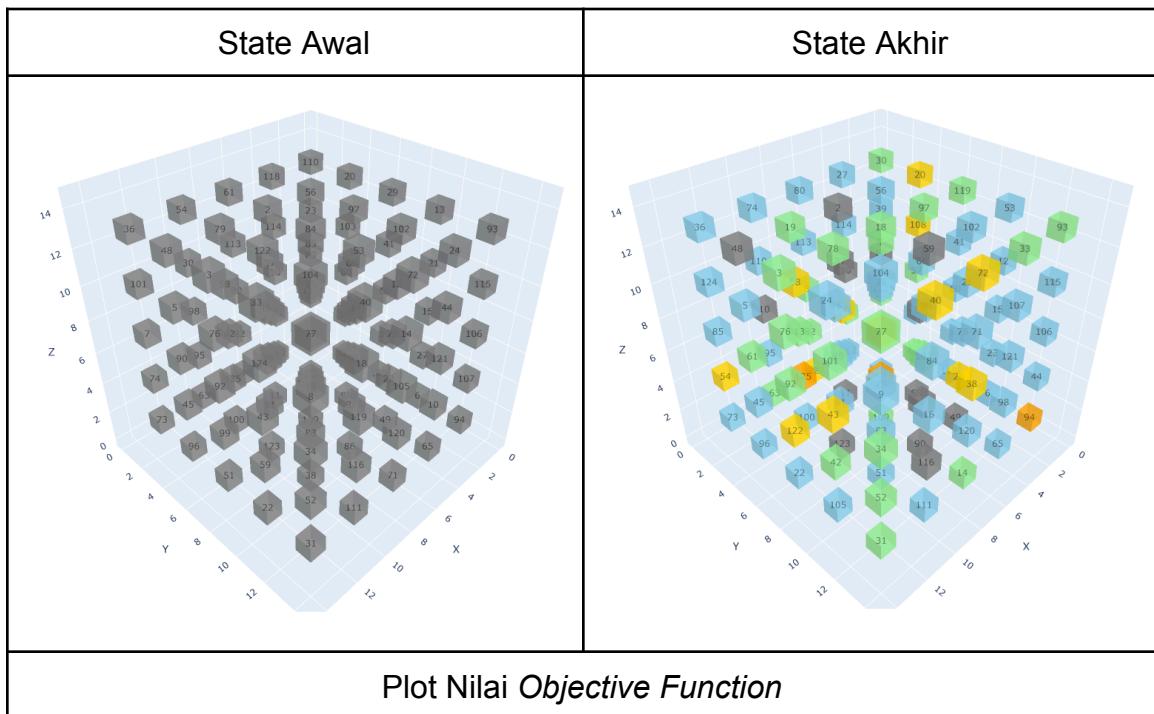
Nilai Akhir *Objective Function*, Waktu Pencarian, dan Banyak *Restart*

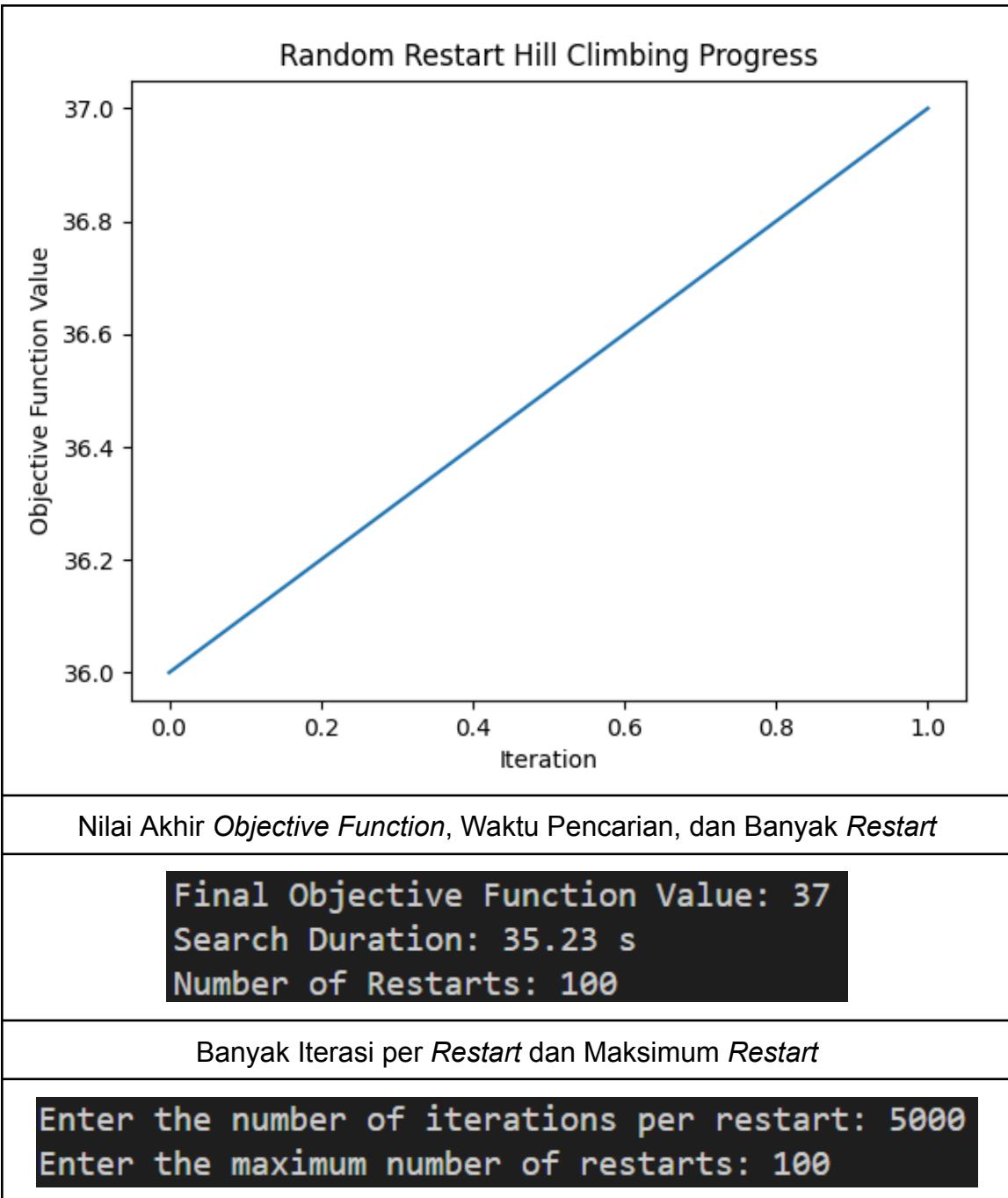
**Final Objective Function Value: 35
Search Duration: 25.90 s
Number of Restarts: 500**

Banyak Iterasi per *Restart* dan Maksimum *Restart*

**Enter the number of iterations per restart: 1000
Enter the maximum number of restarts: 500**

- Iterasi 3





Dari eksperimen di atas, dapat dilihat bahwa dengan memvariasikan banyaknya iterasi per restart dan maksimum restart, memperoleh hasil yang hampir seimbang, yaitu 36, 35, dan 37. Jumlah restart memungkinkan pencarian untuk menjelajahi lebih banyak titik awal, sehingga meningkatkan peluang menemukan solusi yang lebih optimal atau mendekati global optimum. Terdapat variasi durasi pencarian yang signifikan, dimana semakin banyak restart yang dilakukan, semakin lama waktu yang dibutuhkan. Pada eksperimen dengan 50 restart, waktu pencarian relatif singkat (2.42 detik), sedangkan

dengan 500 restart waktu meningkat menjadi 25.90 detik. Hasil dari random restart cukup bervariasi meskipun jumlah restart ditingkatkan. Pada eksperimen dengan 100 restart misalnya, nilai objektif yang dicapai adalah 37, lebih tinggi dari eksperimen 50 restart (36) dan 500 restart (35). Hal ini menunjukkan bahwa meskipun random restart meningkatkan peluang mendapatkan solusi lebih baik, hasilnya tidak selalu konsisten karena tergantung pada keberuntungan *state* yang dipilih secara acak.

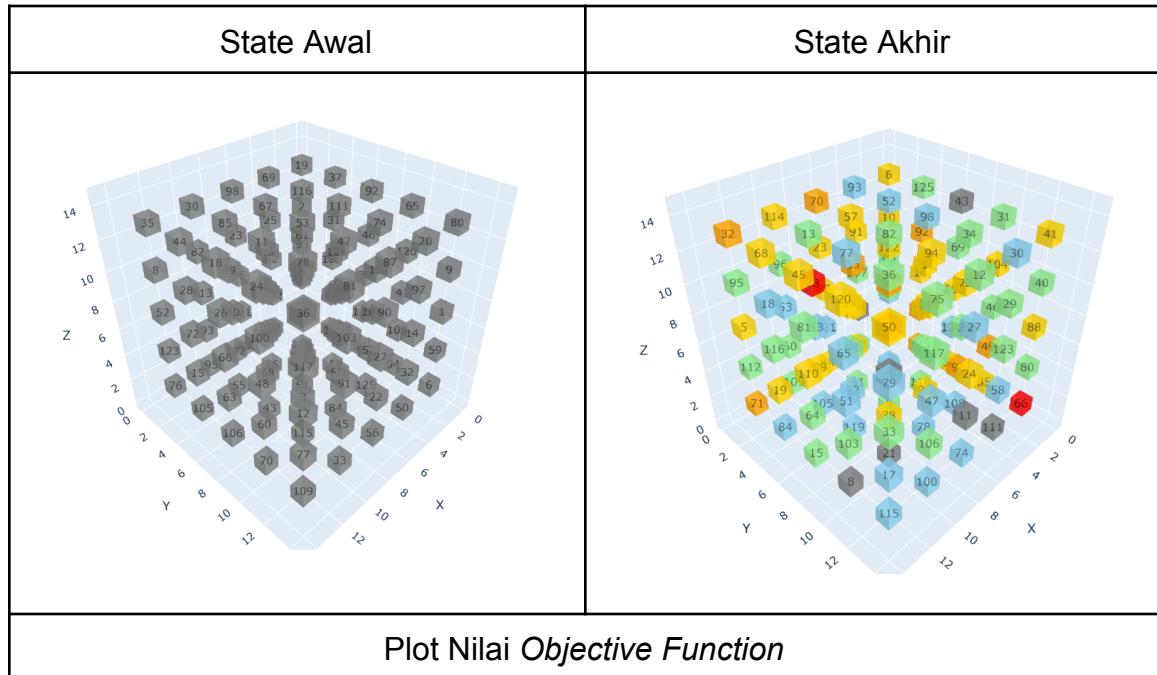
2.3.5. Simulated Annealing

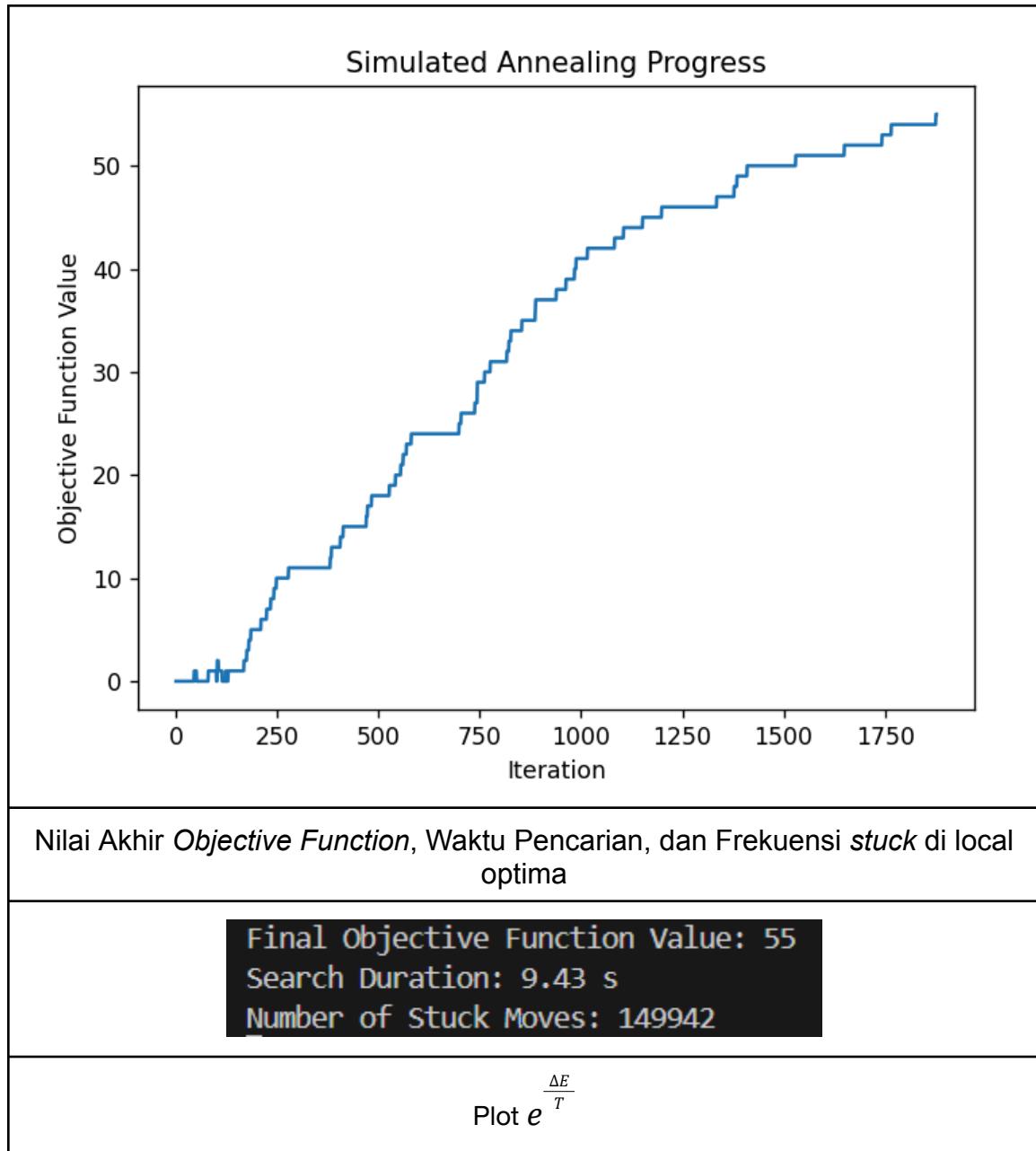
- Iterasi 1

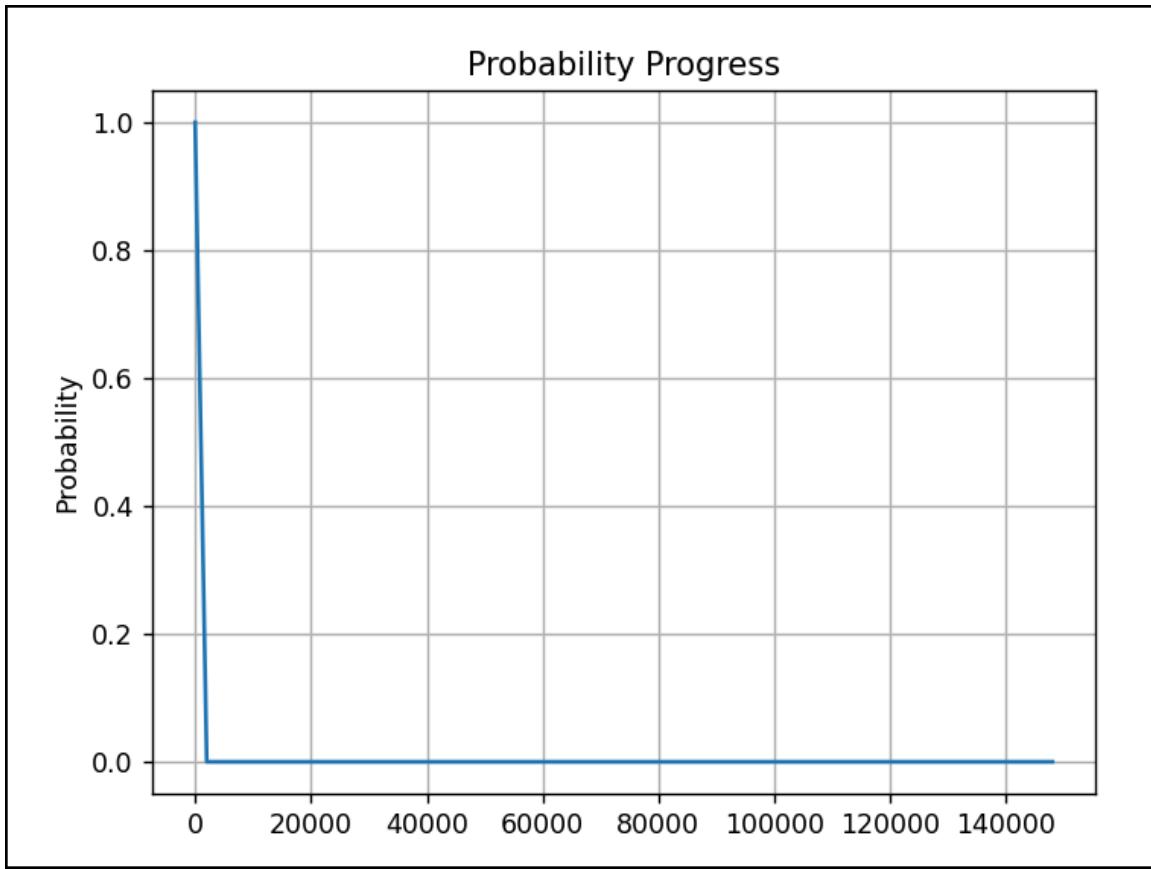
Initial Temperature : 1000

Cooling Rate : 0.95

Maximum Iteration : 150000

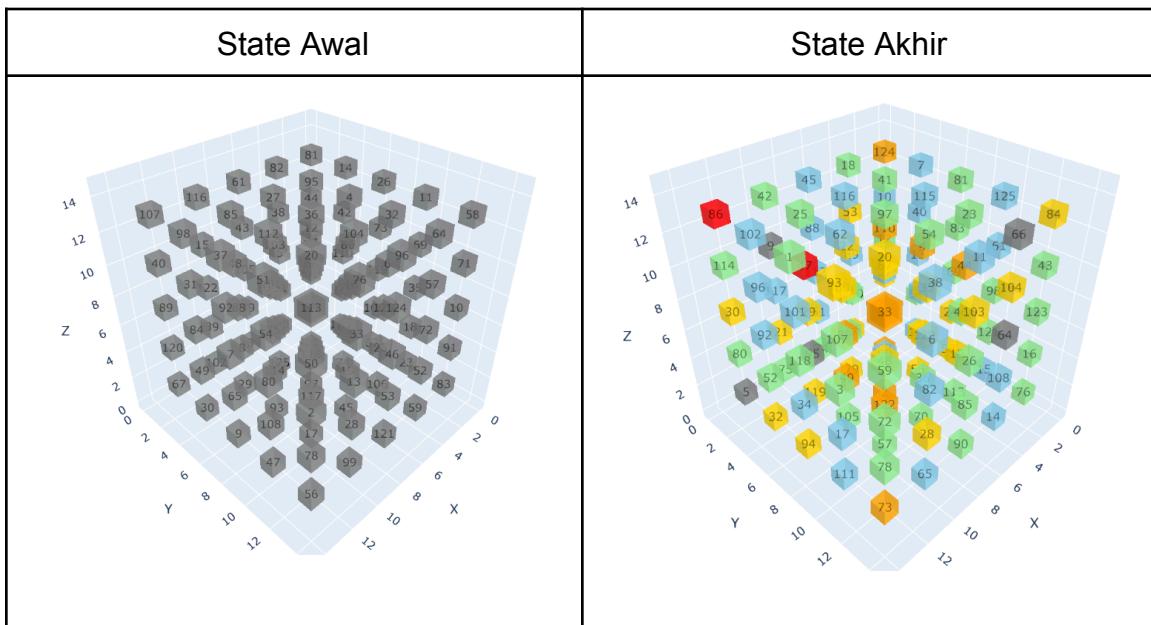






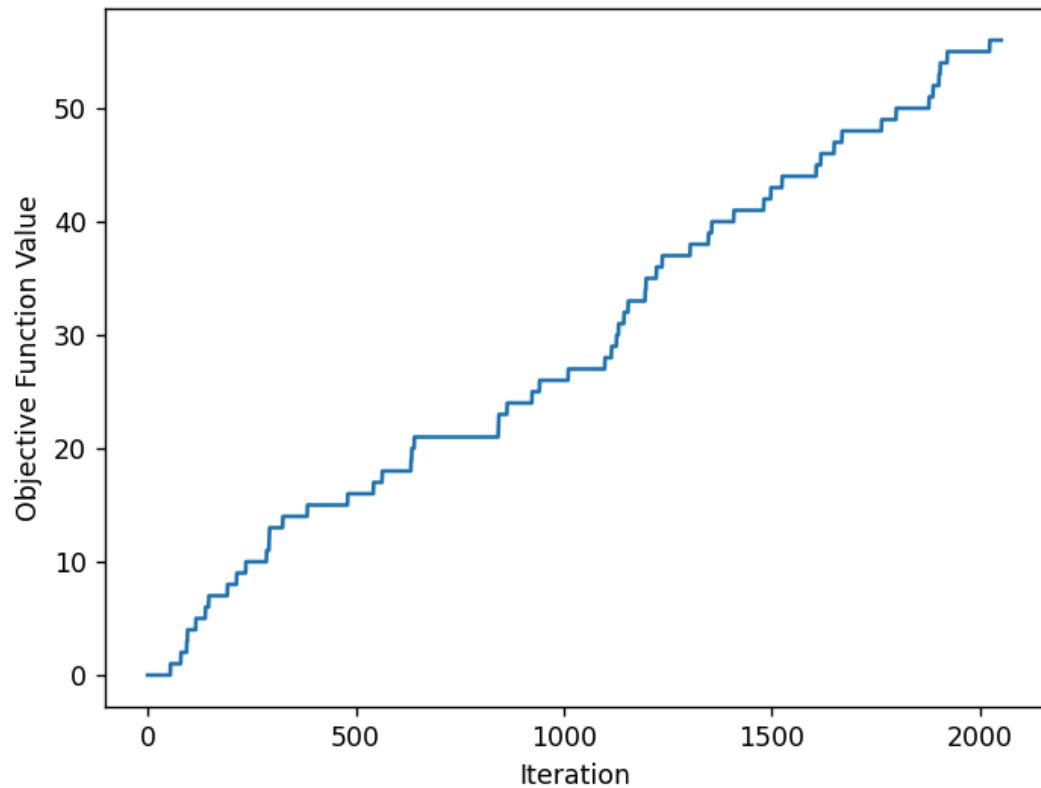
- Iterasi 2

Initial Temperature : 1000
 Cooling Rate : 0.8
 Maximum Iteration : 130000



Plot Nilai *Objective Function*

Simulated Annealing Progress



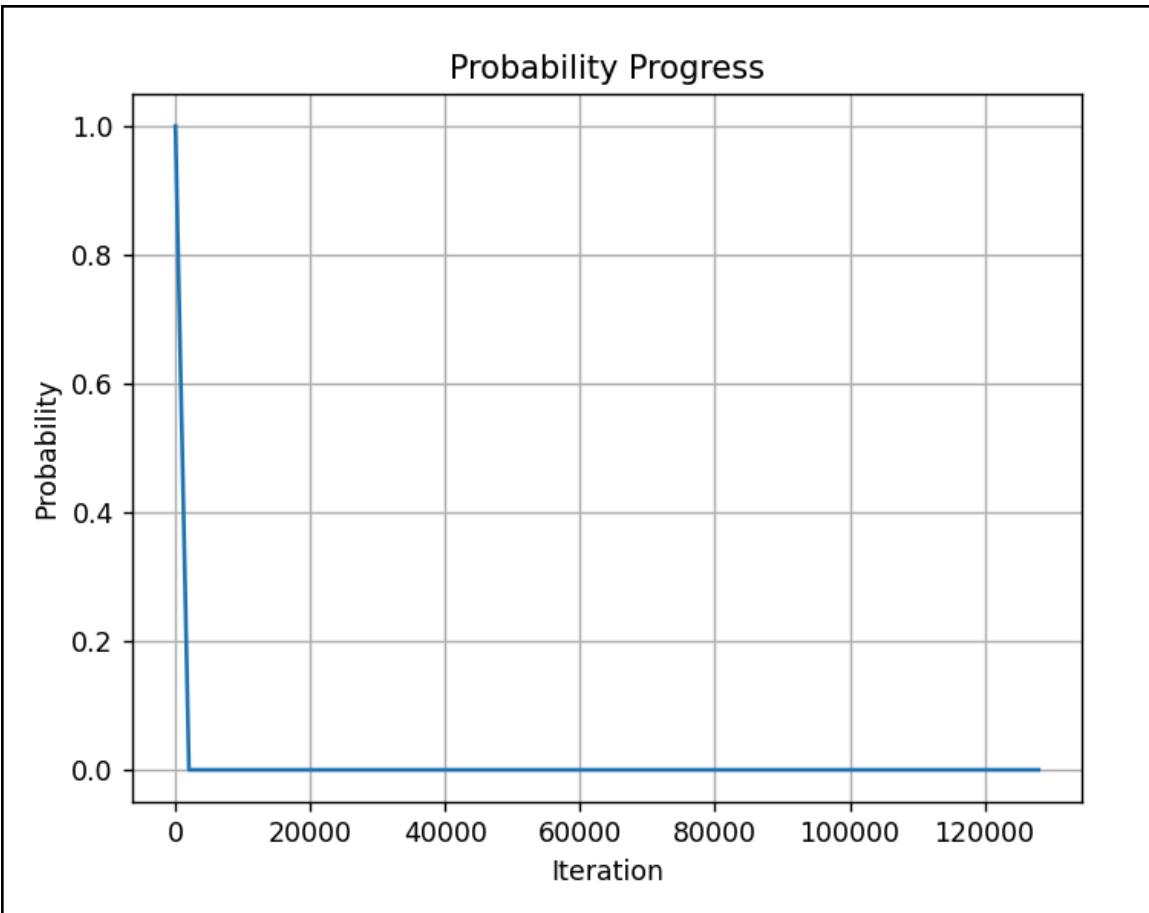
Nilai Akhir *Objective Function*, Waktu Pencarian, dan Frekuensi *stuck* di local optima

Final Objective Function Value: 56

Search Duration: 8.12 s

Number of Stuck Moves: 129944

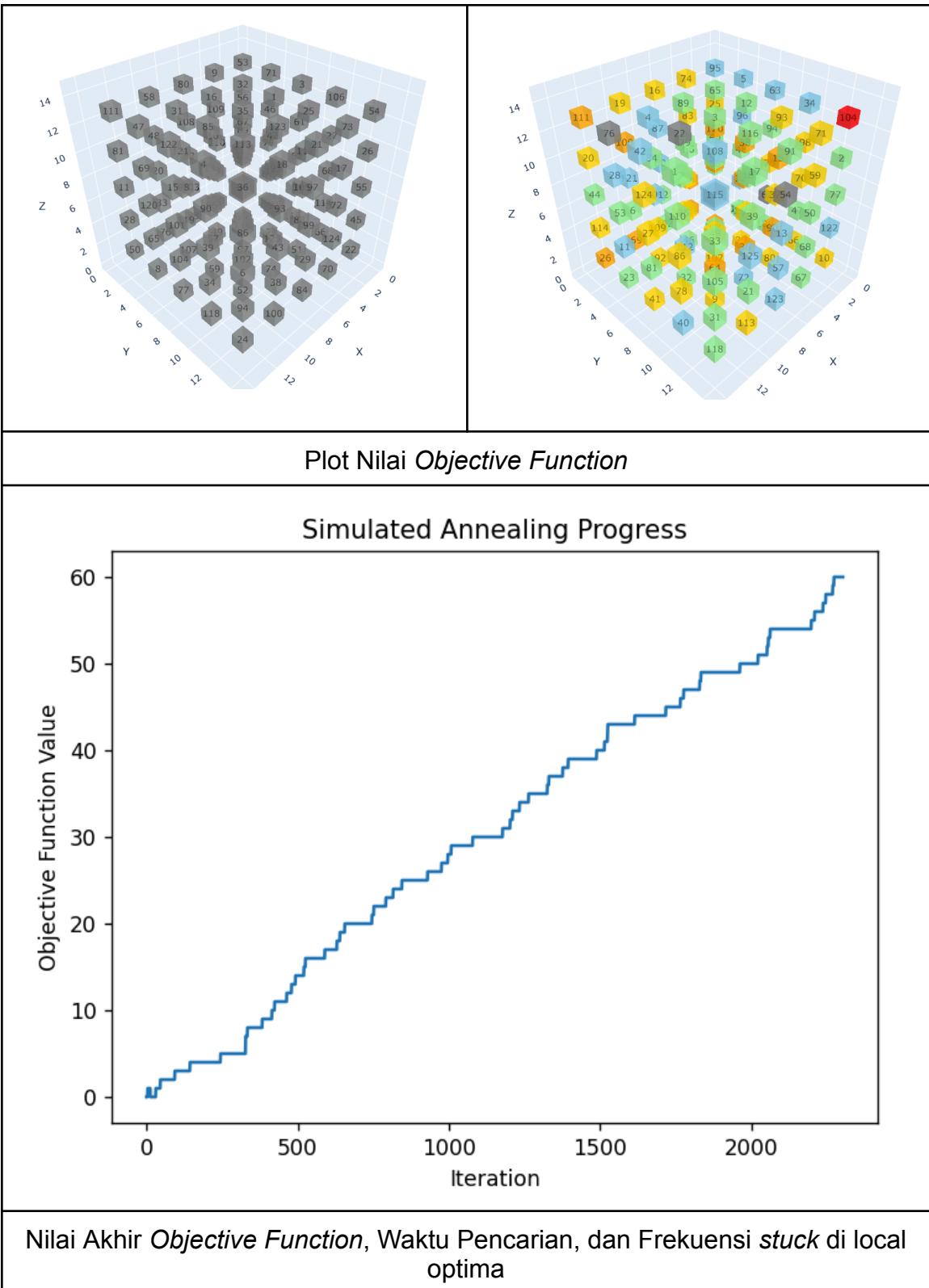
Plot $e^{\frac{\Delta E}{T}}$



- Iterasi 3

Initial Temperature : 100
Cooling Rate : 0.8
Maximum Iteration : 160000

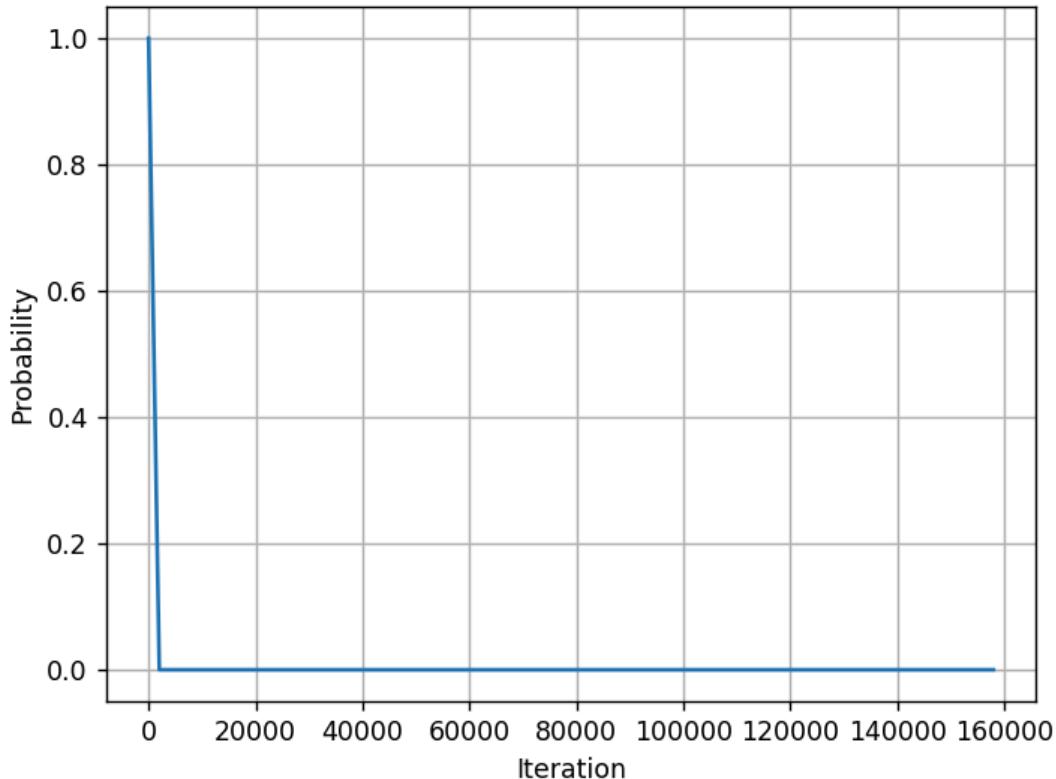
| State Awal | State Akhir |
|------------|-------------|
|------------|-------------|



Final Objective Function Value: 60
Search Duration: 9.96 s
Number of Stuck Moves: 159940

Plot $e^{\frac{\Delta E}{T}}$

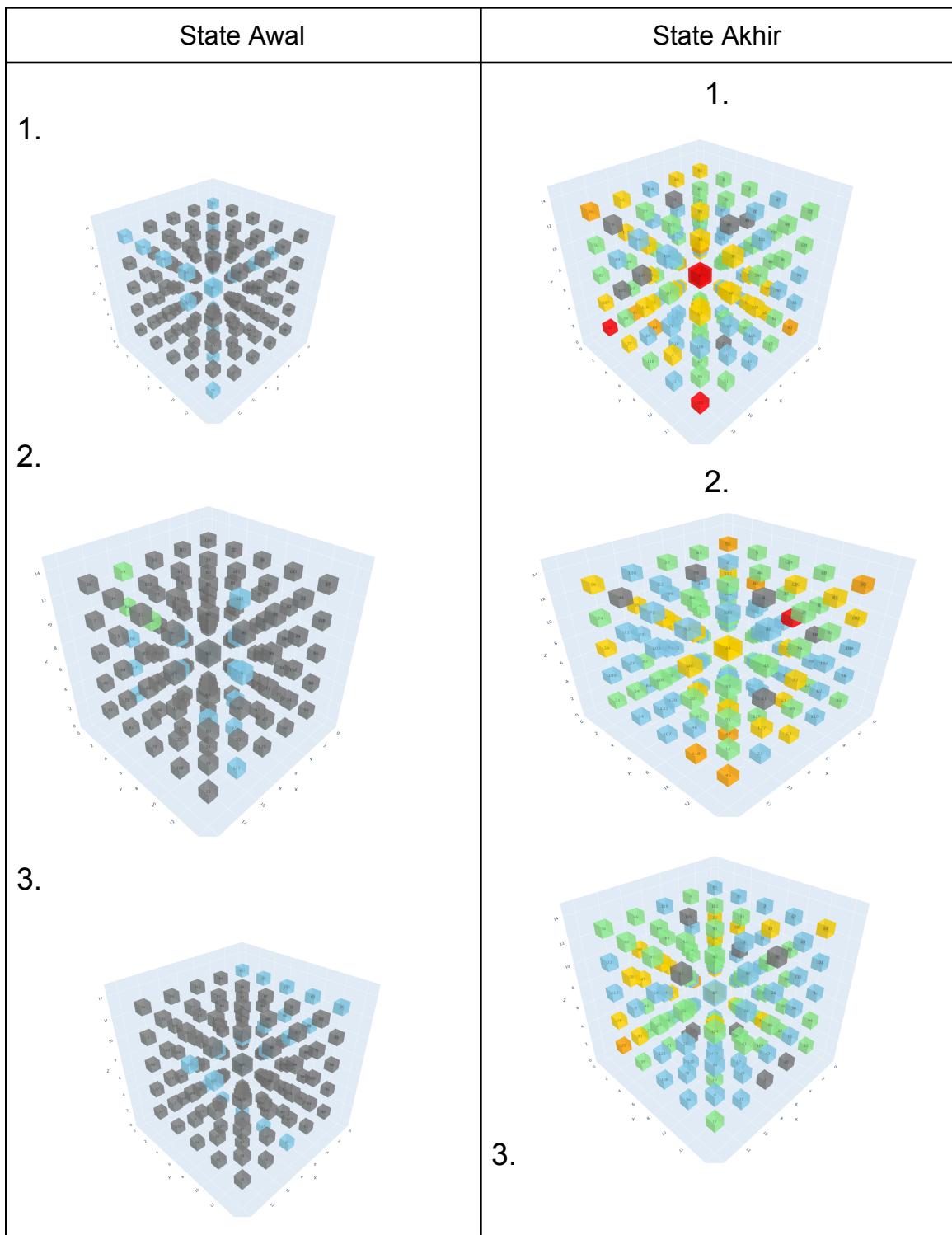
Probability Progress



Pada eksperimen ini, dibuat variasi nilai *initial temperature*, *cooling rate*, dan *maximum iteration* pada setiap iterasi percobaan. Berdasarkan hasil percobaan, cooling rate yang lebih rendah (0.8), terutama pada initial temperature yang lebih rendah (100) menghasilkan nilai objective function tertinggi. Sedangkan nilai *maximum iteration* mempengaruhi waktu yang dibutuhkan, semakin tinggi nilainya maka semakin lama juga waktu yang dibutuhkan. Berdasarkan percobaan ini juga terlihat bahwa algoritma ini memiliki nilai hasil *objective function* yang relatif lebih tinggi dibandingkan algoritma lain serta memiliki waktu yang relatif lebih cepat juga jika dibandingkan dengan algoritma yang lain.

2.3.6. Genetic Algorithm

1. Populasi sebagai kontrol
 - Variasi 1



Plot Nilai Objective Function

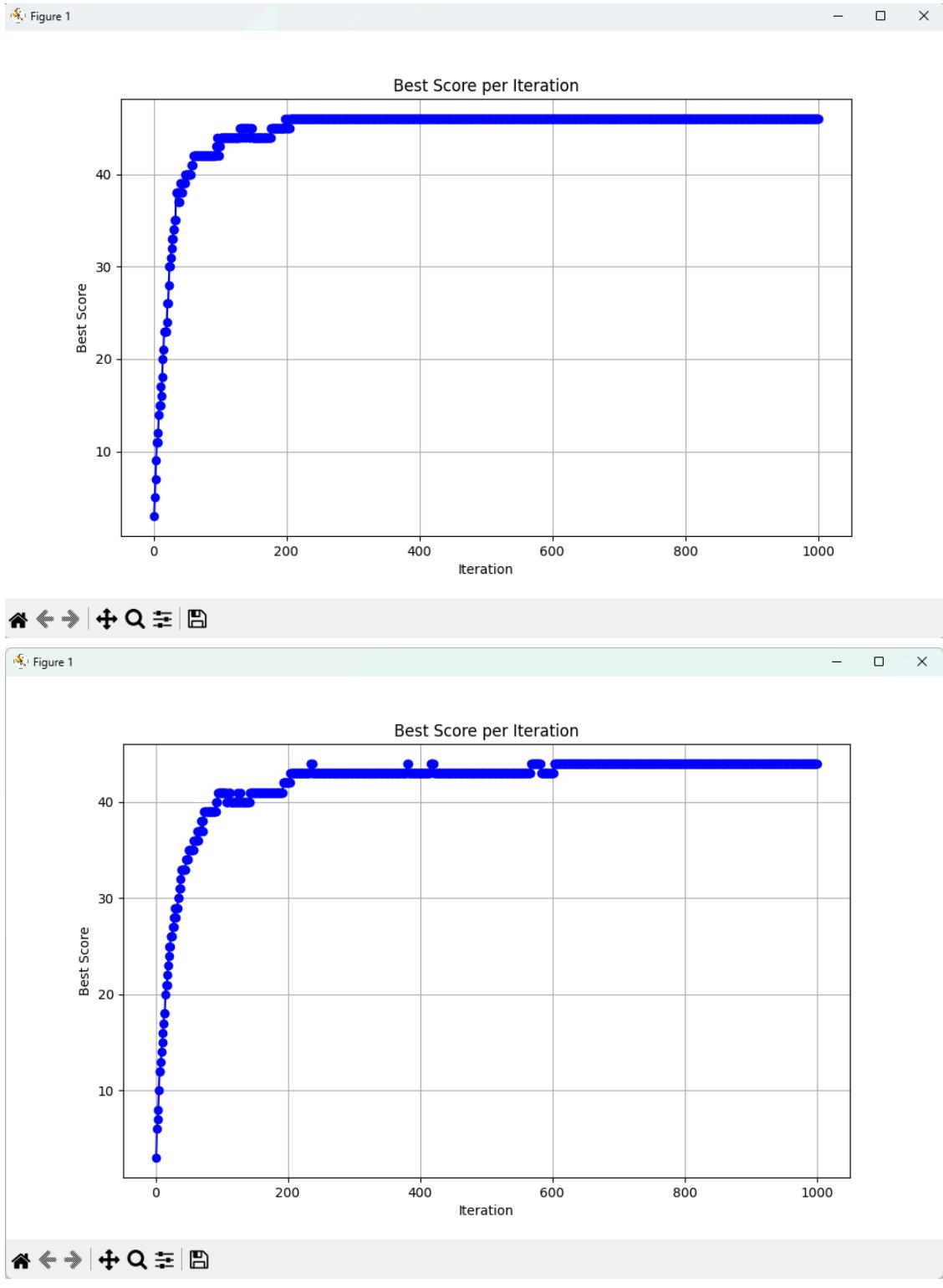
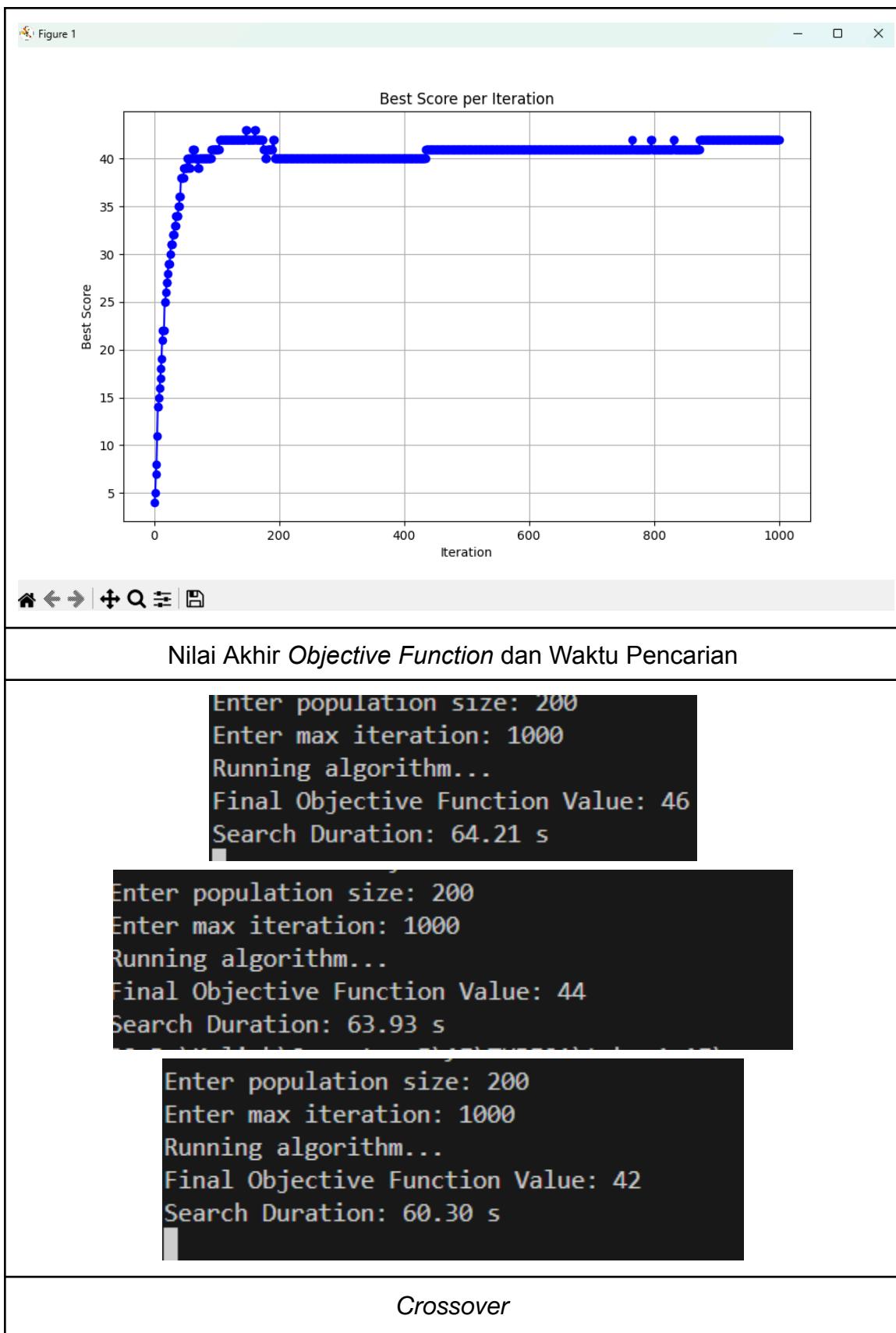
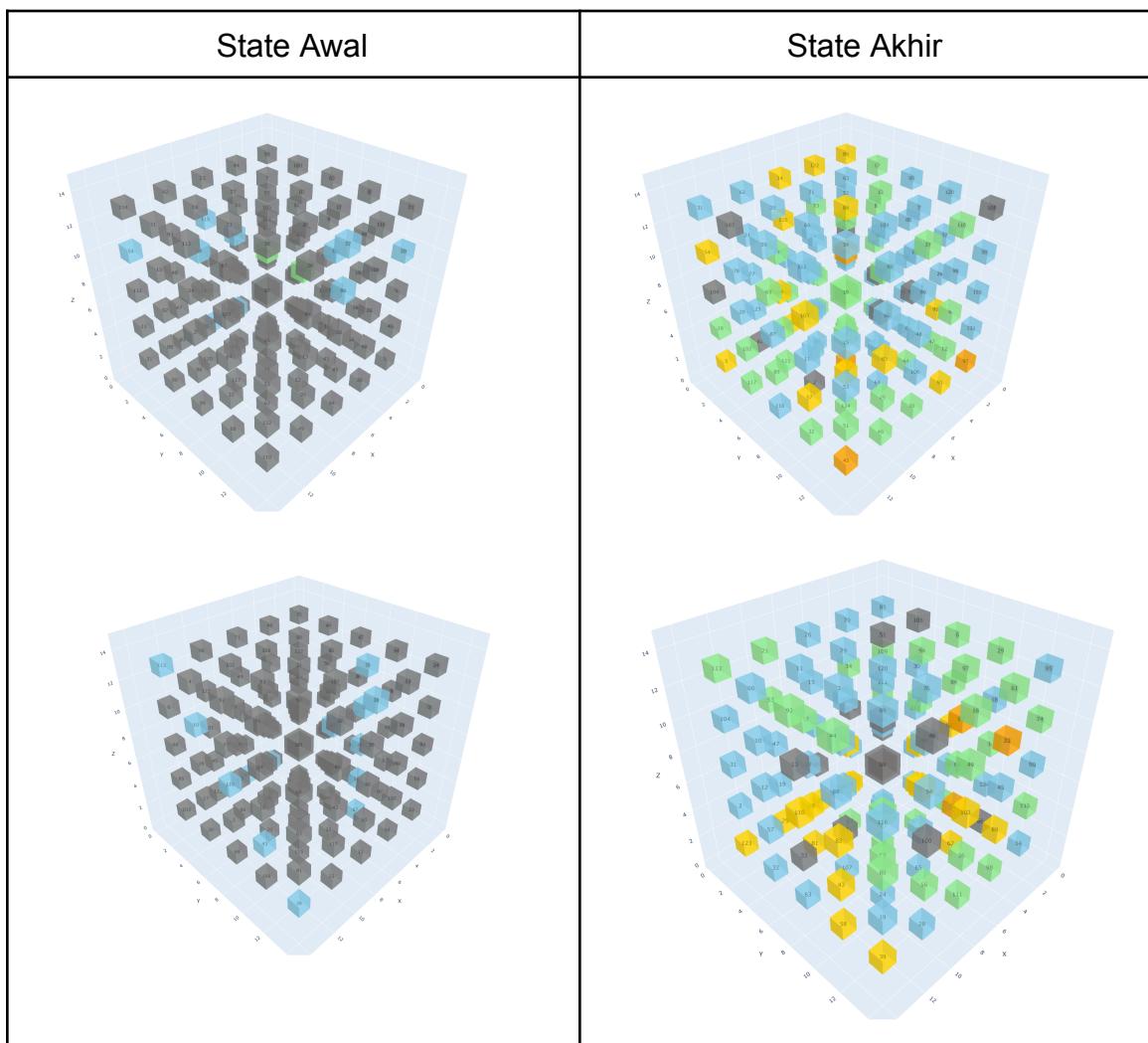


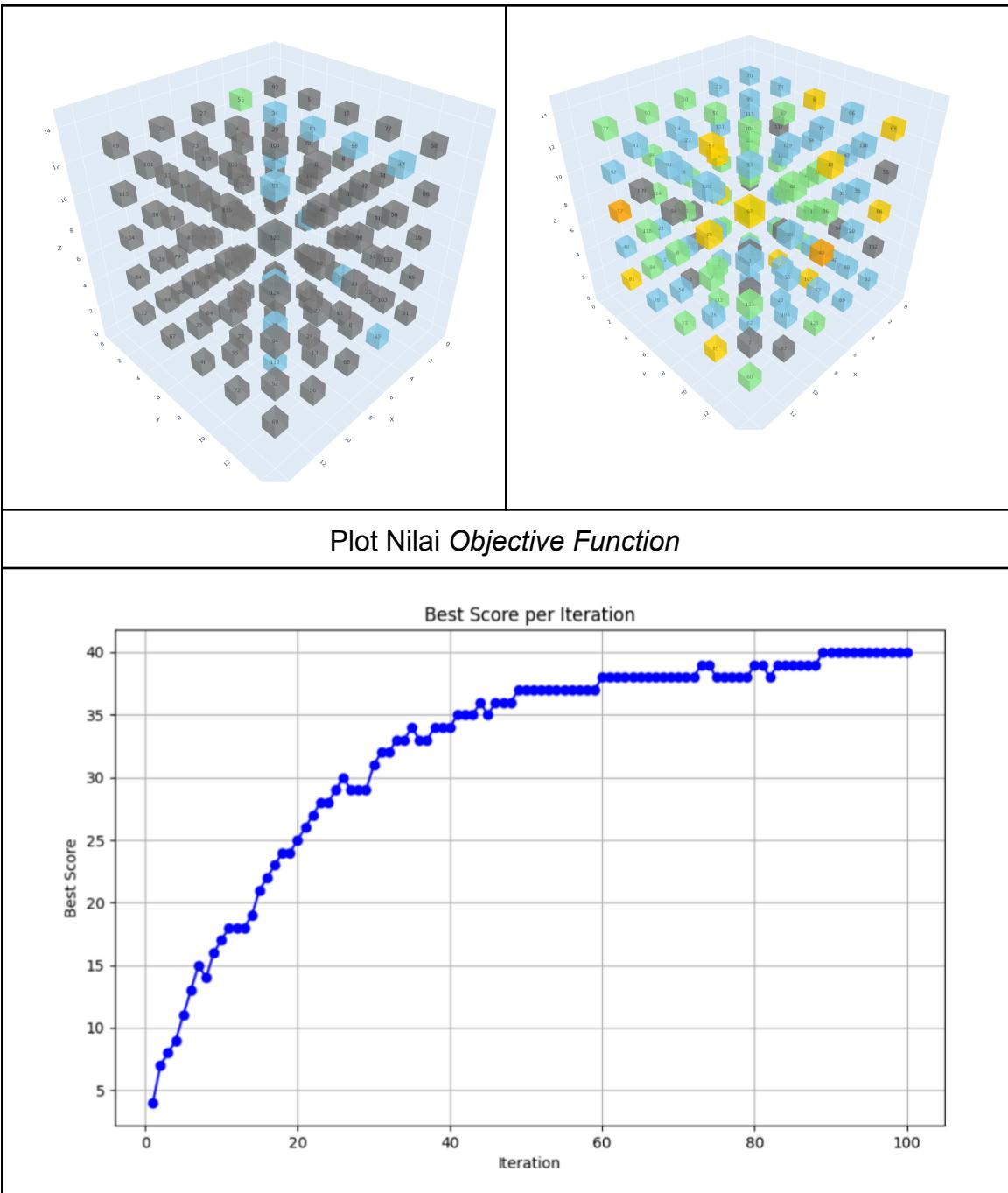
Figure 1

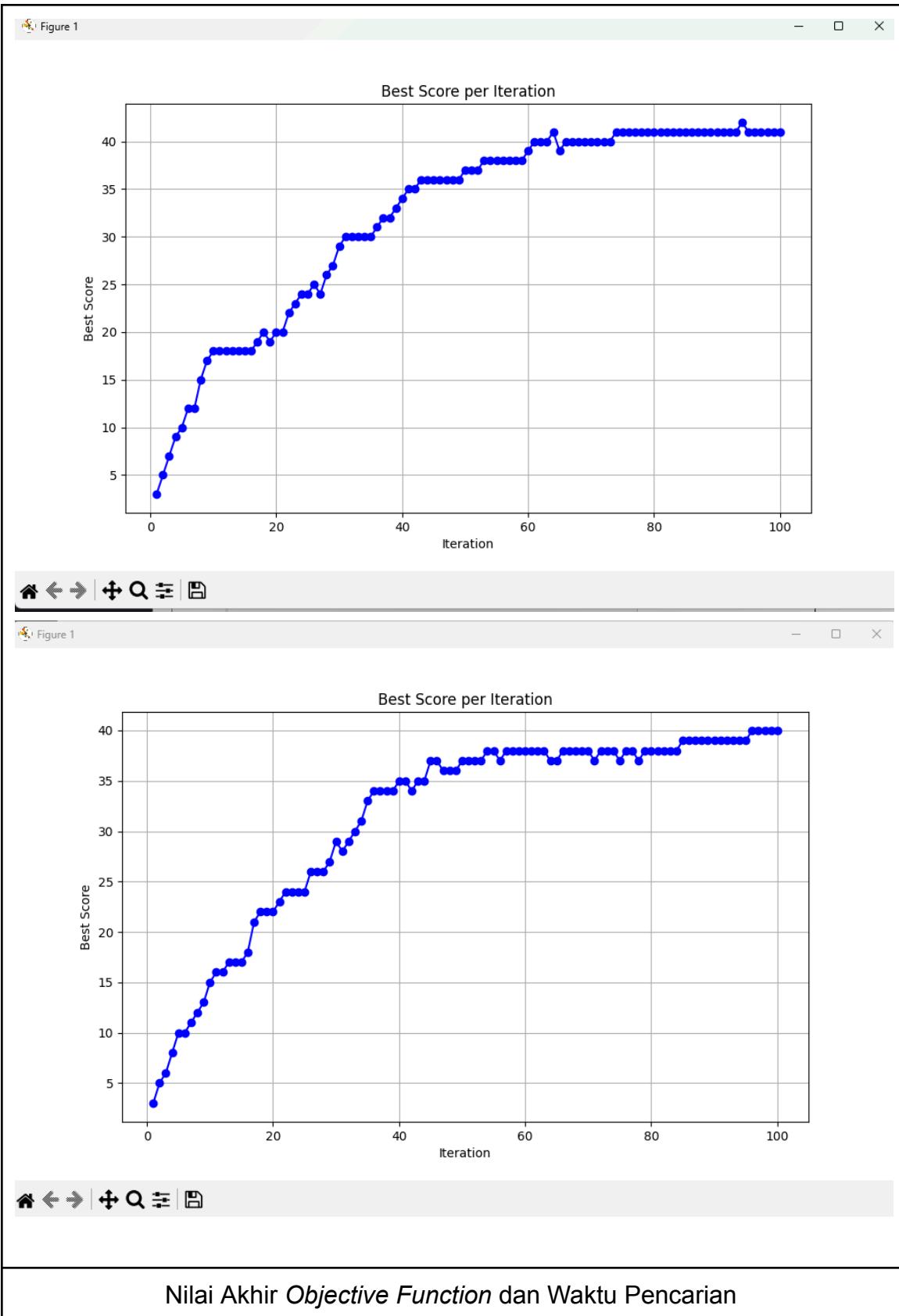


| |
|-------------------------------------|
| custom_segment_preserving_crossover |
| Jumlah Populasi |
| 200 |
| Banyak Iterasi |
| 1000 |

- Variasi 2







```
Enter population size: 200
Enter max iteration: 100
Running algorithm...
Final Objective Function Value: 40
Search Duration: 6.82 s
```

```
Enter population size: 200
Enter max iteration: 100
Running algorithm...
Final Objective Function Value: 41
Search Duration: 7.22 s
```

```
Enter population size: 200
Enter max iteration: 100
Running algorithm...
Final Objective Function Value: 40
Search Duration: 7.07 s
```

Crossover

custom_segment_preserving_crossover

Jumlah Populasi

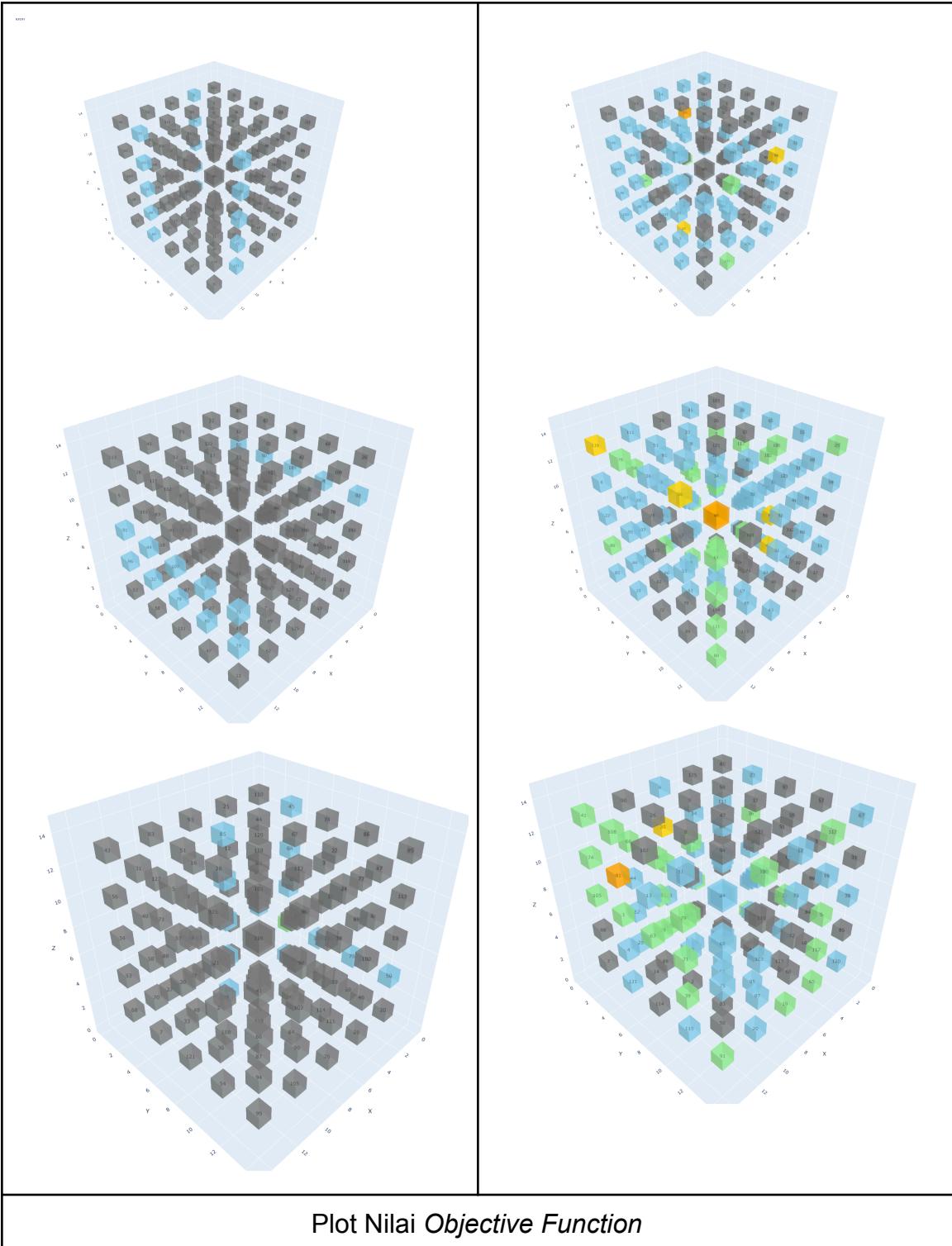
200

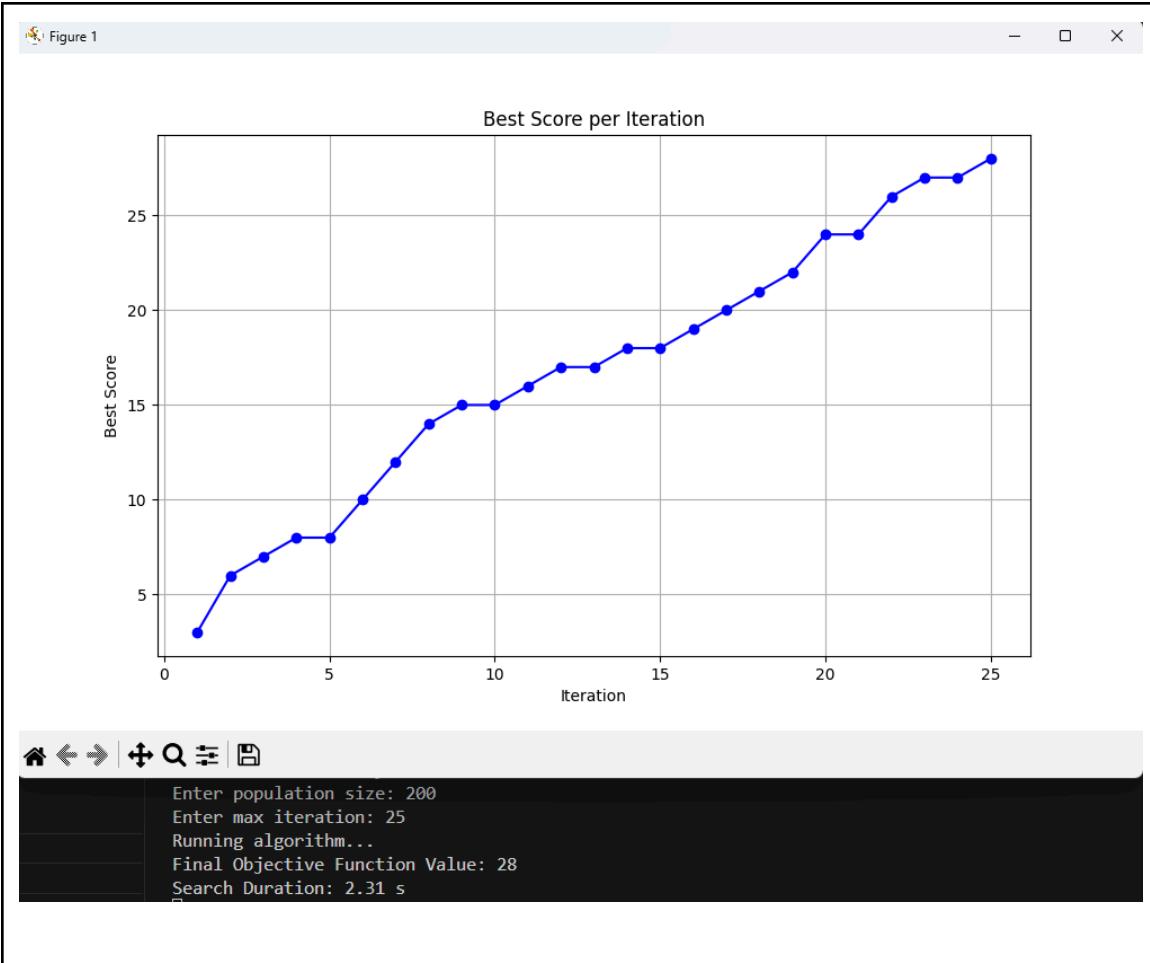
Banyak Iterasi

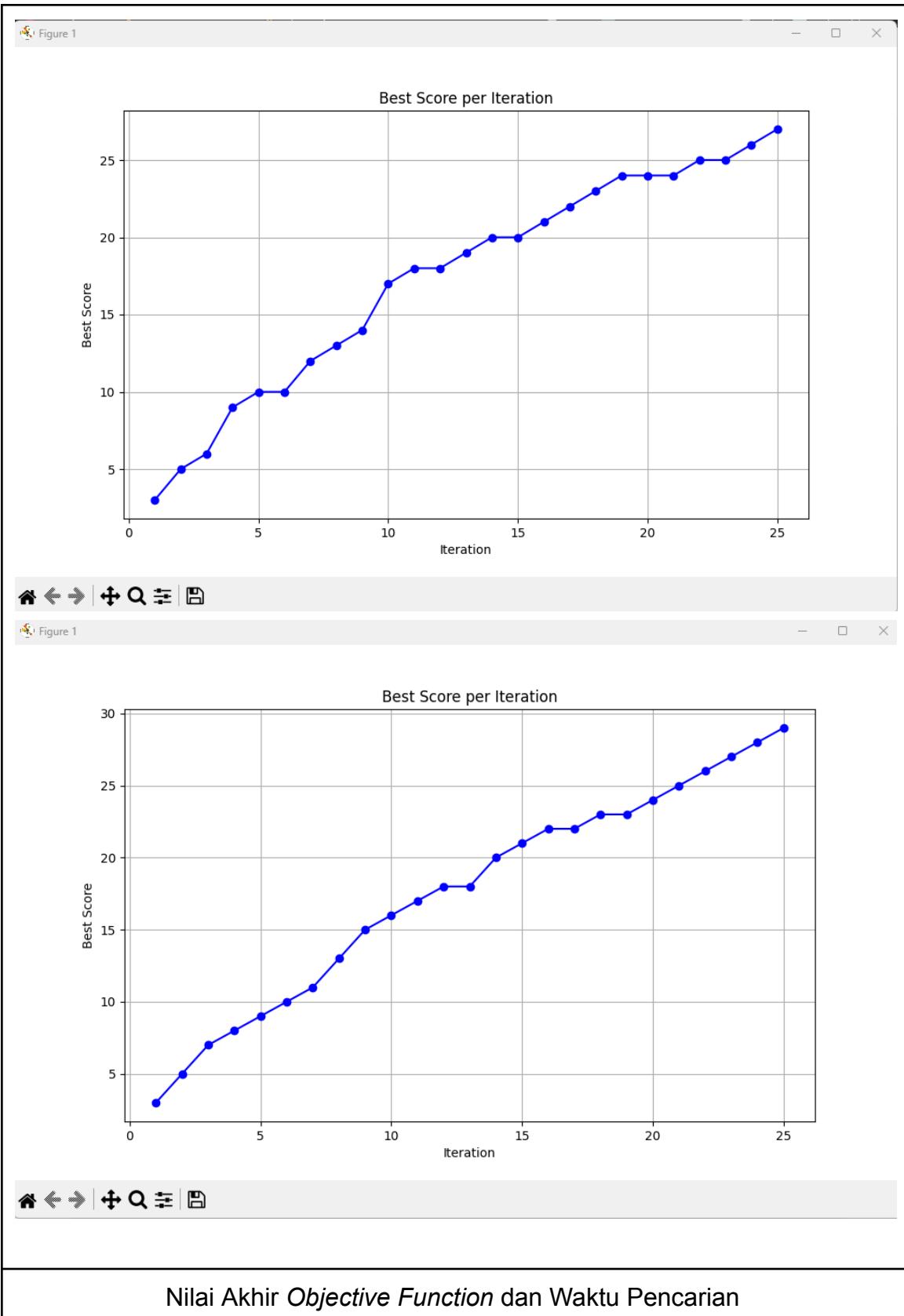
100

- Variasi 3

| | |
|------------|-------------|
| State Awal | State Akhir |
|------------|-------------|







```
Enter the number of your choice: 2
Enter population size: 200
Enter max iteration: 25
Running algorithm...
Final Objective Function Value: 28
Search Duration: 2.31 s
```

```
Enter population size: 200
Enter max iteration: 25
Running algorithm...
Final Objective Function Value: 27
Search Duration: 2.54 s
```

```
Enter population size: 200
Enter max iteration: 25
Running algorithm...
Final Objective Function Value: 29
Search Duration: 2.30 s
```

Crossover

custom_segment_preserving_crossover

Jumlah Populasi

200

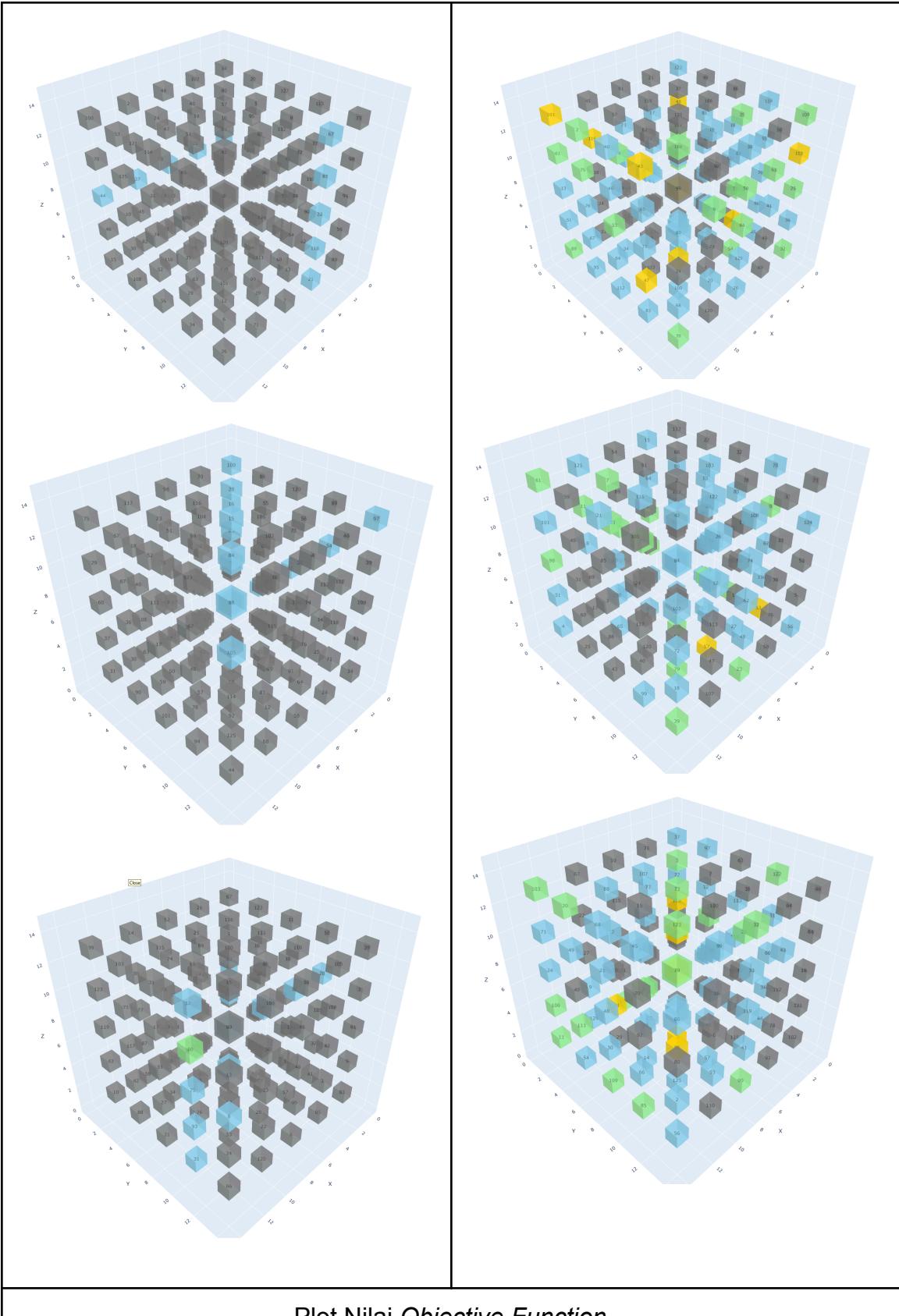
Banyak Iterasi

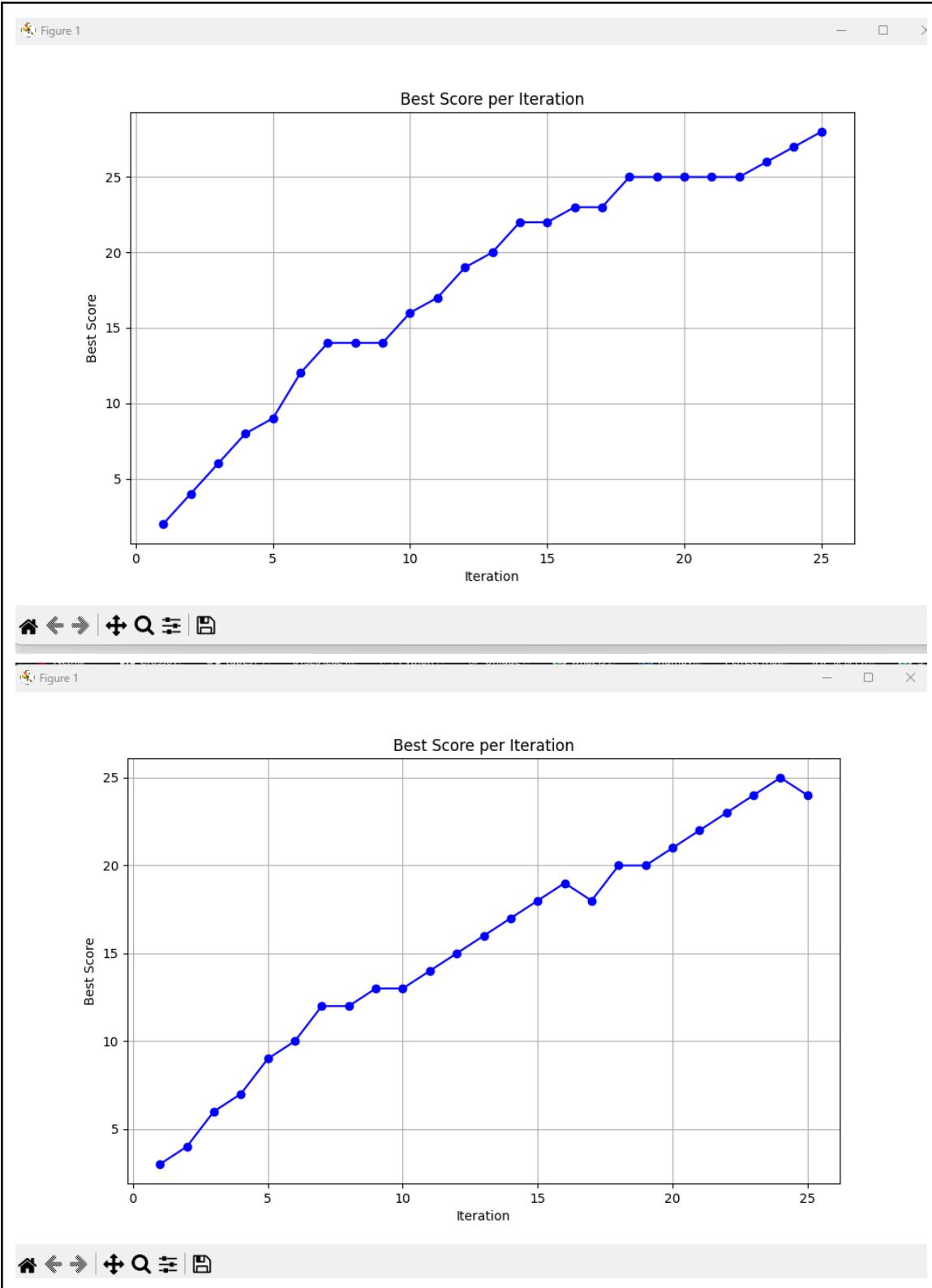
25

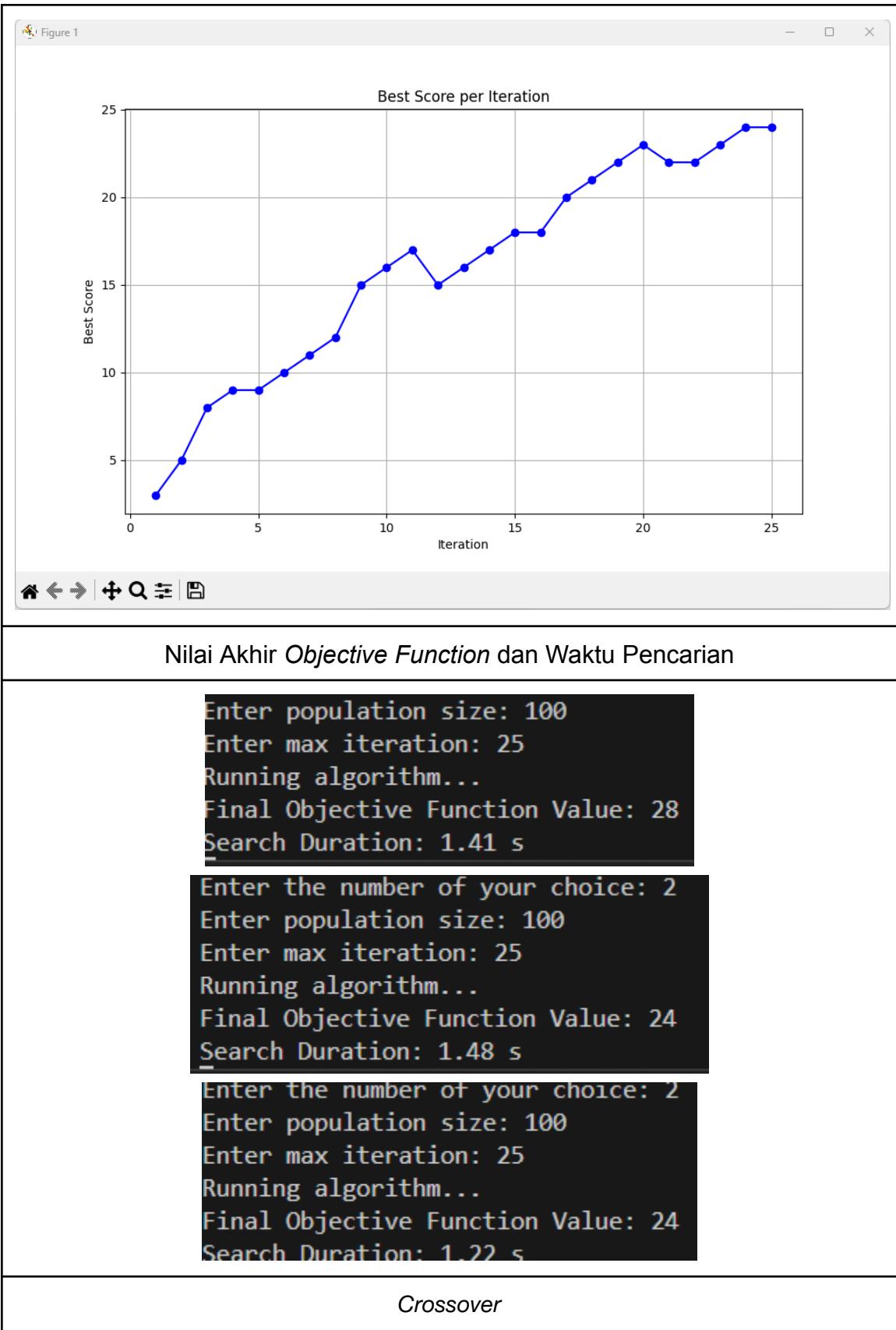
2. Populasi sebagai kontrol

- Variasi 1

| | |
|------------|-------------|
| State Awal | State Akhir |
|------------|-------------|

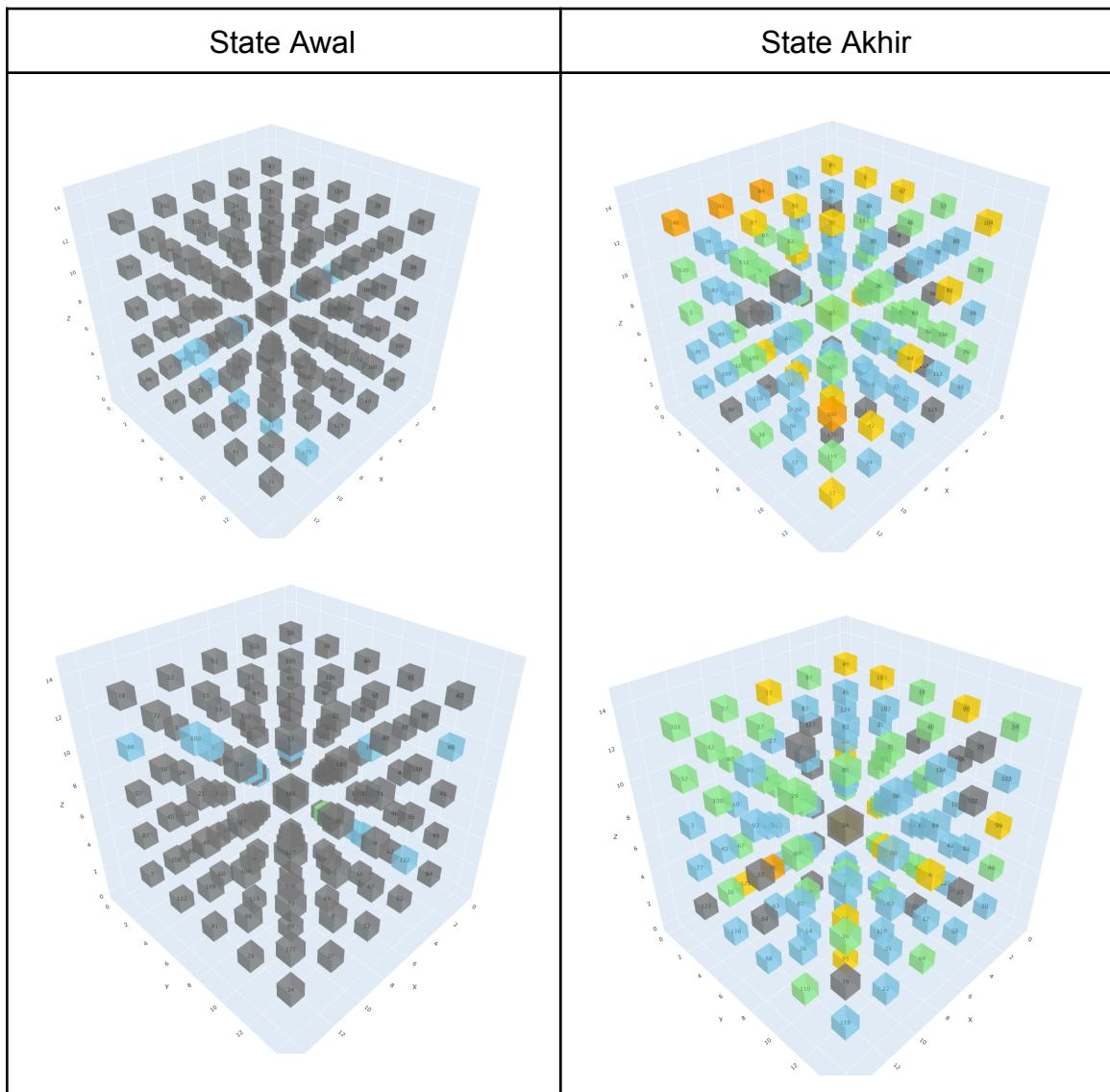


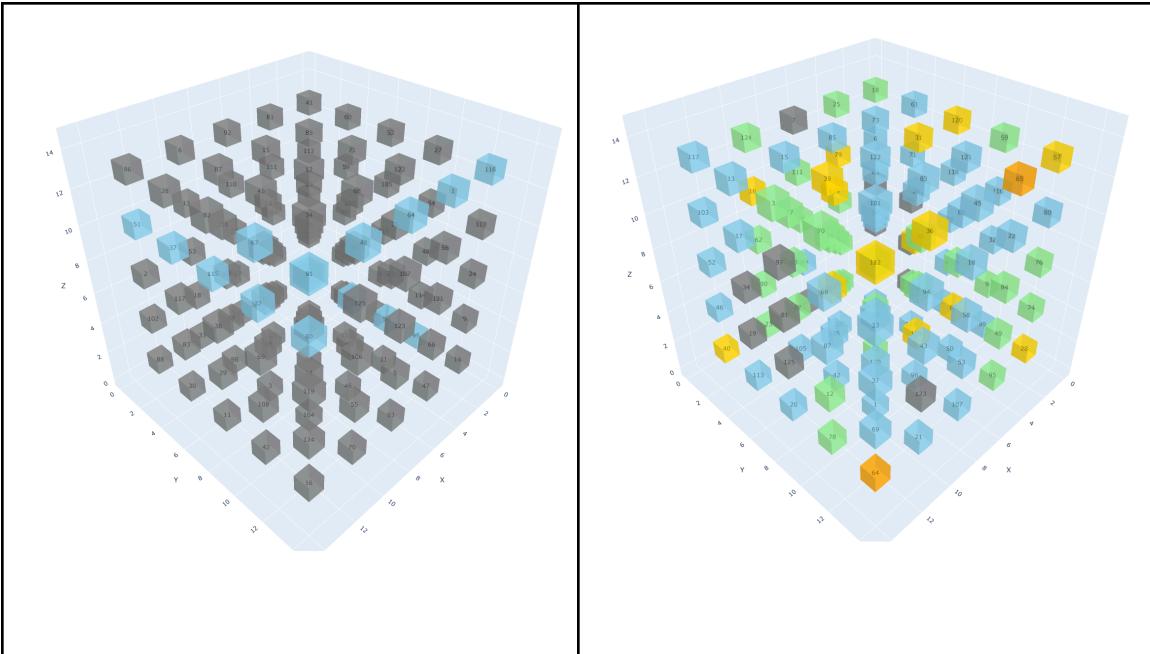




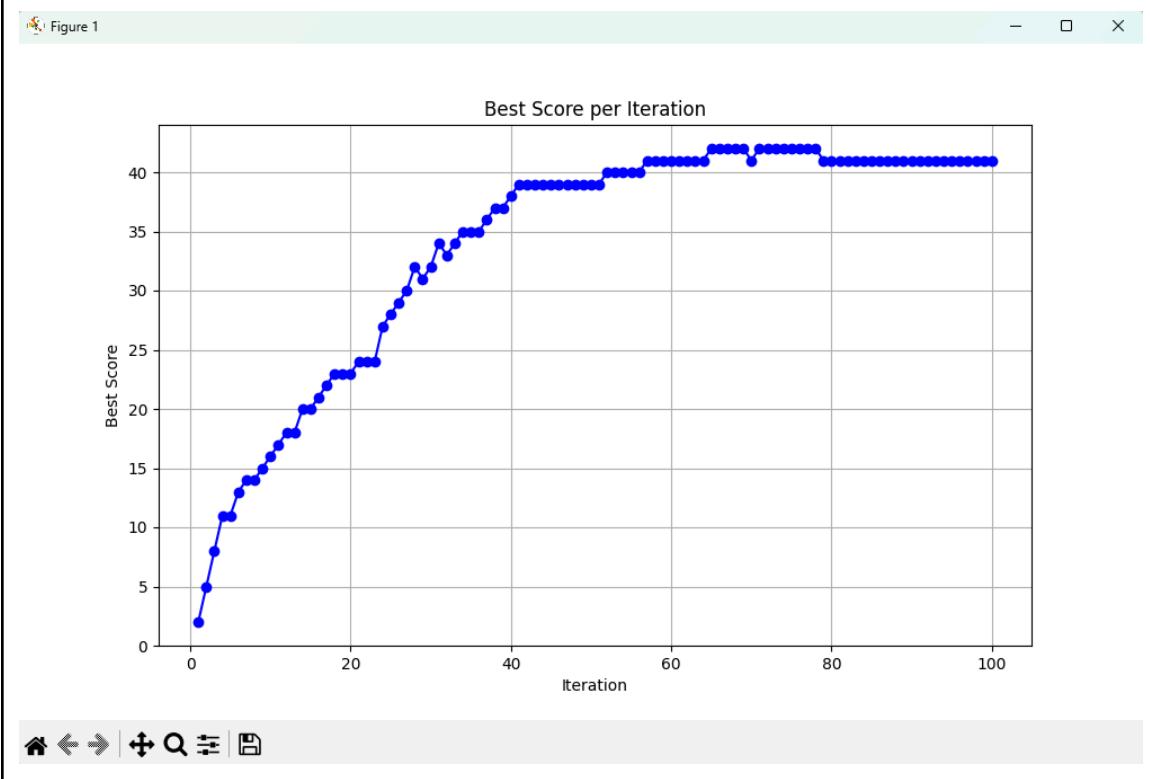
| |
|-------------------------------------|
| custom_segment_preserving_crossover |
| Jumlah Populasi |
| 100 |
| Banyak Iterasi |
| 25 |

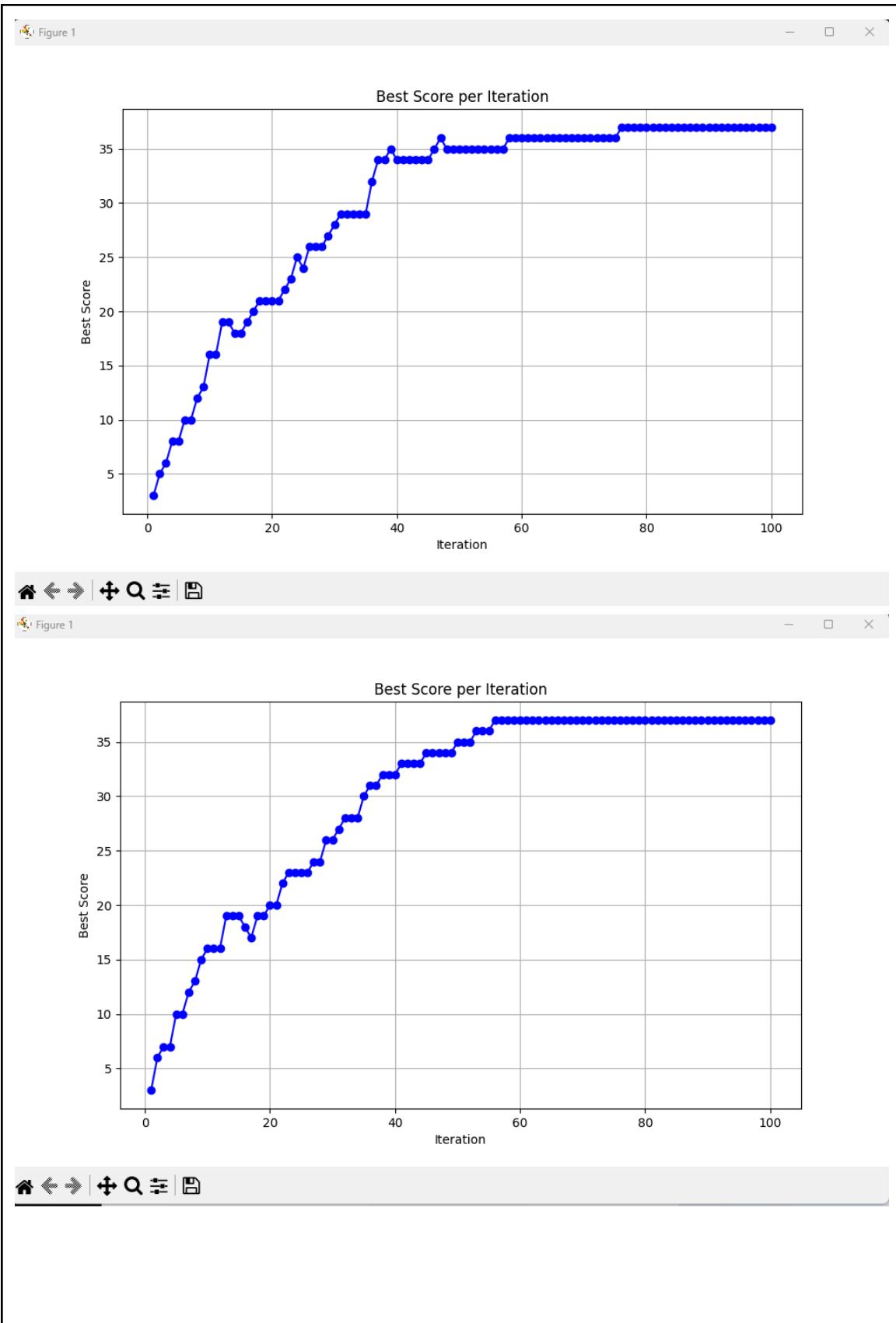
- Variasi 2





Plot Nilai *Objective Function*





Nilai Akhir Objective Function dan Waktu Pencarian

```
Enter the number of your choice: 2
Enter population size: 100
Enter max iteration: 100
Running algorithm...
Final Objective Function Value: 41
Search Duration: 3.97 s
```

```
Enter the number of your choice: 2
Enter population size: 100
Enter max iteration: 100
Running algorithm...
Final Objective Function Value: 37
Search Duration: 3.90 s
```

```
Enter the number of your choice: 2
Enter population size: 100
Enter max iteration: 100
Running algorithm...
Final Objective Function Value: 37
Search Duration: 4.10 s
```

Crossover

custom_segment_preserving_crossover

Jumlah Populasi

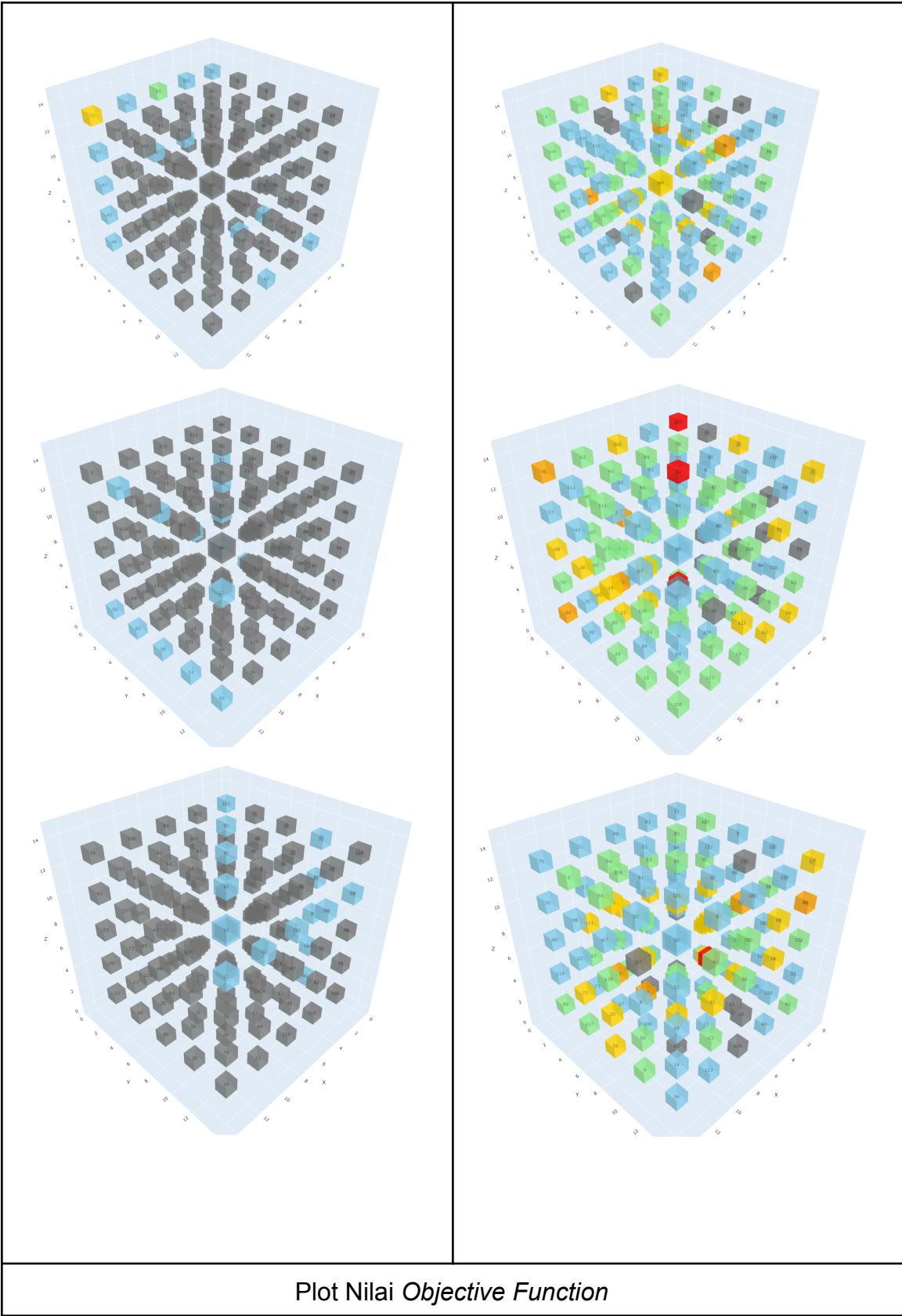
100

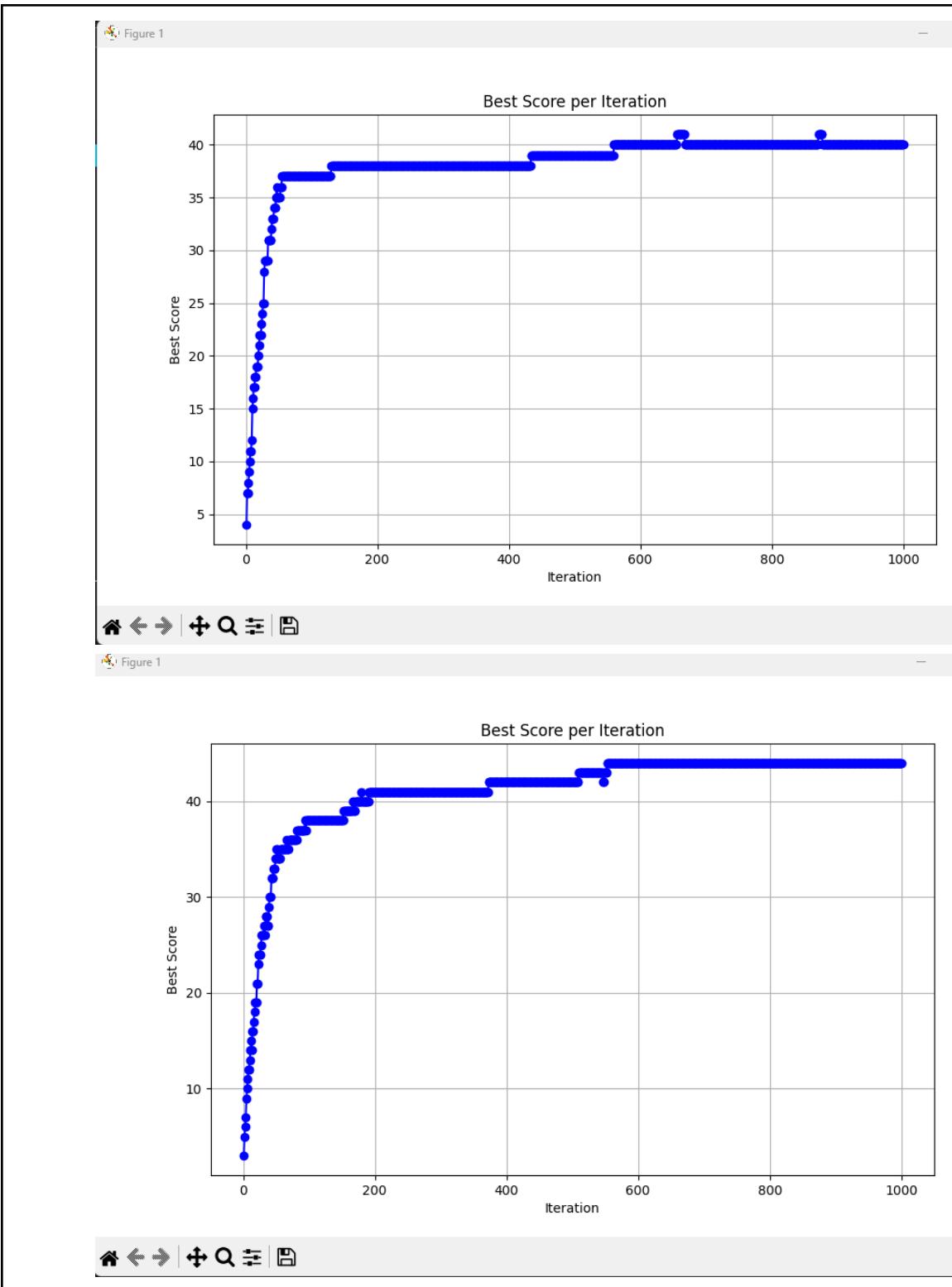
Banyak Iterasi

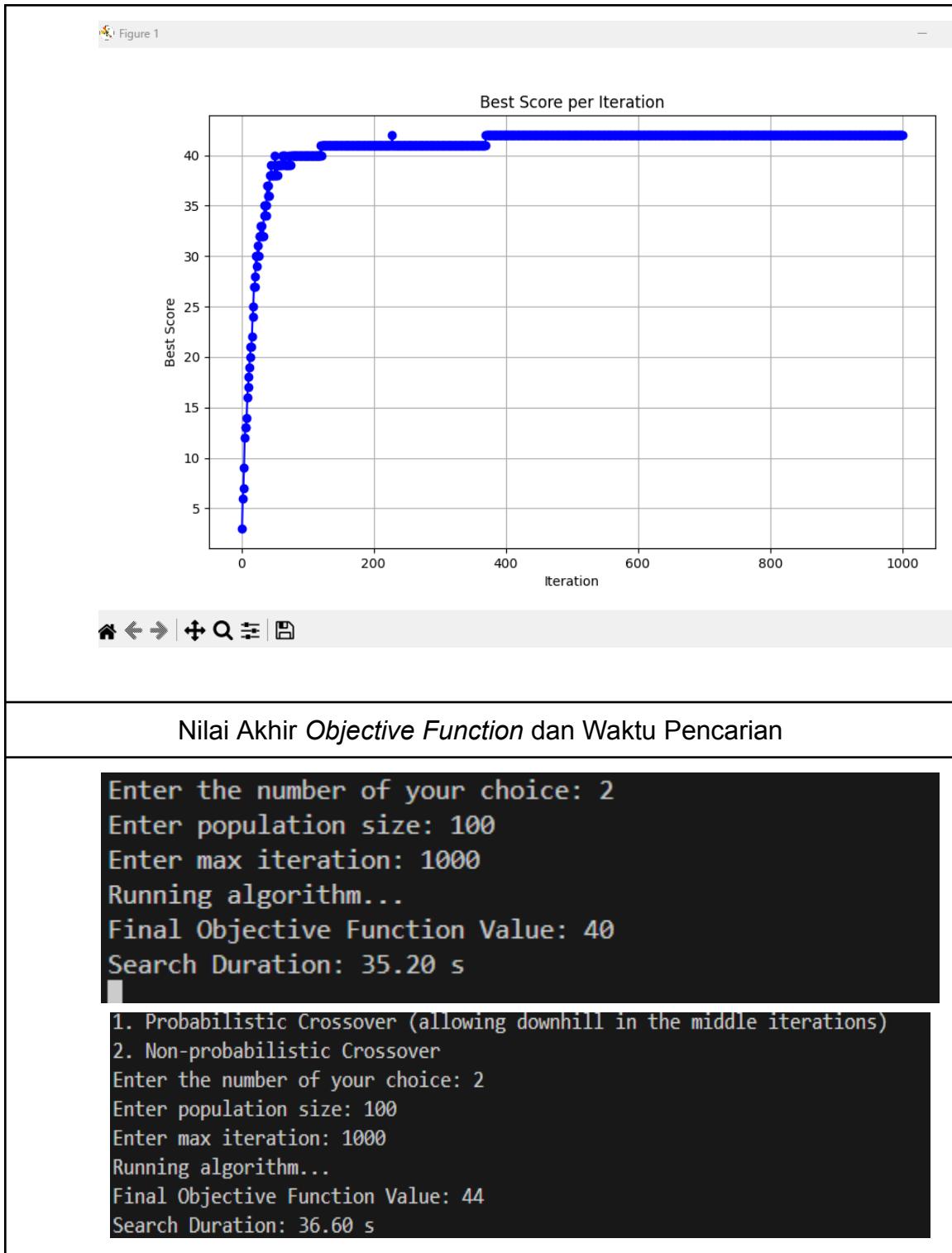
100

- Variasi 3

| | |
|------------|-------------|
| State Awal | State Akhir |
|------------|-------------|







```
Enter the number of your choice: 2
Enter population size: 100
Enter max iteration: 1000
Running algorithm...
Final Objective Function Value: 42
```

Crossover

custom_segment_preserving_crossover

Jumlah Populasi

100

Banyak Iterasi

1000

3. Menggunakan *crossover* yang lain

Selain itu, kami mencoba menggunakan *crossover* yang lain, yaitu *custom_probabilistic_segment_preserving_crossover*. *Crossover* ini memungkinkan terjadinya perpindahan ke solusi yang lebih rendah (*downhill move*) di tengah proses dengan menerapkan *threshold* pada segmen yang sudah *match*. Dengan demikian, segmen yang sudah *match* masih memiliki kemungkinan untuk di-*crossover*.

Nilai Akhir *Objective Function* dan Waktu Pencarian

```
Enter the number of your choice: 6
1. Probabilistic Crossover (allowing downhill in the middle iterations)
2. Non-probabilistic Crossover
Enter the number of your choice: 1
Enter population size: 200
Enter max iteration: 400
Running algorithm...
Final Objective Function Value: 46
Search Duration: 25.18 s
```

| |
|---|
| Crossover |
| custom_probabilistic_segment_preserving_crossover |
| Jumlah Populasi |
| 200 |
| Banyak Iterasi |
| 400 |

Genetic Algorithm memiliki kemampuan untuk mendekati *global optimum* karena sifatnya yang mengeksplorasi ruang solusi secara acak, terutama melalui kombinasi operasi *crossover* dan *mutation*. Untuk *Magic Cube*, global optimum tercapai saat semua baris, kolom, dan diagonal memiliki jumlah angka yang sama, yaitu *magic number*. Namun, pencapaian *global optimum* sangat tergantung pada keragaman populasi awal, strategi *crossover* dan *mutation*, dan jumlah iterasi dan ukuran populasi.

Secara keseluruhan, *Genetic Algorithm* sering kali lebih efektif untuk permasalahan seperti *Magic Cube* karena keragaman populasinya memungkinkan pencarian yang lebih luas dan probabilitas keluar dari *local optimum* yang lebih tinggi dibandingkan dengan algoritma pencarian lokal lainnya. Namun, kembali lagi, tergantung *crossover function* nya. *Crossover function* yang kami gunakan menurut kami terlalu cepat *convergent (convergent prematur)*, tetapi kami anggap sudah cukup karena *match* dan durasi nya sudah bisa menyamai algoritma local search lain. Dalam kasus *Magic Cube*, *Genetic Algorithm* mungkin lebih lambat dibandingkan algoritma lokal lainnya tetapi menawarkan hasil yang lebih baik, terutama untuk ruang solusi besar. Pada *Magic Cube*, *Genetic Algorithm* cenderung memberikan hasil yang lebih konsisten dengan pengaturan parameter yang tepat, meskipun ada sedikit variasi karena sifat stokastik dari *crossover* dan *mutation*.

Pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada *Genetic Algorithm* adalah sebagai berikut:

1. Jumlah Iterasi
 - Lebih banyak iterasi memungkinkan eksplorasi solusi yang lebih baik, tetapi ada titik di mana penambahan iterasi tidak lagi memberikan peningkatan kualitas yang signifikan karena algoritma sudah mendekati konvergensi.
 - Pada *Magic Cube*, iterasi tambahan seringkali memberikan hasil yang lebih baik hingga titik tertentu, tetapi setelah itu peningkatan kualitas solusi mungkin minim (*convergent*).
2. Ukuran Populasi

- Populasi yang besar meningkatkan keragaman solusi, yang membantu menghindari *local optimum* dan memungkinkan eksplorasi ruang solusi lebih baik. Namun, ukuran populasi yang terlalu besar bisa memperlambat proses pencarian secara signifikan.
- Untuk *Magic Cube*, ukuran populasi yang lebih besar membantu *Genetic Algorithm* mencapai solusi yang lebih mendekati optimal, tetapi dengan waktu komputasi yang lebih tinggi.

Dalam kombinasi, kedua faktor ini sangat memengaruhi keseimbangan antara kualitas solusi dan waktu komputasi. Untuk *Magic Cube*, pengaturan jumlah iterasi dan populasi perlu disesuaikan untuk mencapai solusi optimal dengan efisiensi waktu yang layak.

2.4. Perbandingan Seluruh Algoritma Local Search

| Steepest Ascent | | Stochastic | | Sideways Move | | Random Restart | | Simulated Annealing | | Genetic Algorithm | |
|-----------------|---------|------------|---------|---------------|---------|----------------|---------|---------------------|---------|-------------------|---------|
| Nilai | Duras i | Nilai | Duras i | Nilai | Duras i | Nilai | Duras i | Nilai | Duras i | Nilai | Duras i |
| 41 | 26.63 | 39 | 12.36 | 41 | 68.57 | 36 | 2.42 | 55 | 9.43 | 46 | 64.21 |
| 40 | 32.62 | 41 | 4.37 | 39 | 68.62 | 35 | 25.90 | 56 | 8.12 | 41 | 7.22 |
| 39 | 34.01 | 37 | 2.70 | 38 | 62.99 | 37 | 35.23 | 60 | 9.96 | 46 | 25.18 |
| 40 | 31.08 | 39 | 6.47 | 39.3 | 66.7 | 36 | 21.1 | 57 | 9.17 | 44.3 | 32.2 |

Tabel 2.4.1. Perbandingan Seluruh Algoritma Local Search

Dari tabel di atas, terlihat bahwa algoritma dengan nilai *objective function* terbaik adalah *Simulated Annealing*, yang mencapai nilai tertinggi 60 dengan waktu 9,96 detik. *Simulated Annealing* juga memiliki rata-rata nilai sekitar 57 dengan waktu rata-rata yang sangat cepat, yaitu 9,17 detik. Selanjutnya, *Genetic Algorithm* menempati posisi kedua dengan rata-rata nilai 44,3 meskipun waktu pencarinya relatif lama. Di urutan berikutnya, tiga algoritma—*Steepest*, *Stochastic*, dan *Sideways Move*—memiliki skor yang hampir setara. Namun, jika dilihat dari segi kecepatan, *Stochastic* unggul dengan waktu rata-rata sekitar 6 detik, dibandingkan *Steepest* dengan 31 detik dan *Sideways Move* dengan 66 detik. Terakhir, algoritma *Random Restart* memiliki rata-rata skor paling rendah, yaitu 36, dengan waktu pencarian yang cukup lama, sekitar 21 detik.

Bab 3

Kesimpulan dan Saran

Algoritma *Simulated Annealing* menjadi pilihan terbaik dalam pencarian solusi optimal *Diagonal Magic Cube* dengan kecepatan dan stabilitas yang tinggi. Algoritma ini mampu mencapai nilai *objective function* terbaik dalam waktu singkat dibandingkan algoritma lainnya. Sementara itu, algoritma *Genetic Algorithm* juga menunjukkan performa yang baik dalam mendekati global optimum, meskipun membutuhkan waktu lebih lama karena karakteristik eksploratifnya. Algoritma lain, seperti *Steepest Ascent*, *Stochastic*, dan *Sideways Move*, menunjukkan stabilitas yang cukup baik namun rentan terhadap local optima, sementara *Random Restart* memiliki variasi yang besar tergantung pada jumlah restart yang dilakukan.

Sebagai saran, *Simulated Annealing* dapat dipilih sebagai metode pencarian utama ketika diperlukan keseimbangan antara kecepatan dan ketepatan hasil. Namun, untuk masalah yang lebih kompleks seperti *Magic Cube*, *Genetic Algorithm* dengan pengaturan parameter yang tepat misalnya, ukuran populasi dan jumlah iterasi yang cukup akan lebih efektif dalam mencapai hasil yang lebih mendekati global optimum. Selain itu, disarankan untuk mengevaluasi kembali *crossover function* pada *Genetic Algorithm* agar dapat mengurangi kemungkinan *premature convergence* dan meningkatkan eksplorasi ruang solusi.

Bab 4

Pembagian Tugas

| NIM | Nama | Tugas |
|----------|--|---|
| 13522067 | Randy Verdian | Steepest Ascent HC, Stochastic HC, Sideways Move HC, Random Restart HC |
| 13522079 | Emery Fathan Zwageri  | Genetic Algorithm |
| 13522092 | Sa'ad Abdul Hakim | Simulated Annealing |
| 13522097 | Ellijah Darrellshane S. | Visualisasi |

Tabel 4.1. Pembagian Tugas

Bab 5

Referensi

"Magic Features." *Magischvierkant*,

<https://www.magischvierkant.com/three-dimensional-eng/magic-features/>.

"Magic Cube." *Wikipedia*, https://en.wikipedia.org/wiki/Magic_cube.

ITB. *IF3170 Artificial Intelligence* PowerPoint Teknik Informatika ITB: Beyond Classical Search.

Russell, Stuart, and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed., Upper Saddle River, NJ: Prentice Hall, 2010.

"Crossover (Genetic Algorithm)." *Wikipedia*,

[https://en.wikipedia.org/wiki/Crossover_\(genetic_algorithm\)](https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)).

Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*.

Bab 6

Lampiran

Link repository github: <https://github.com/randyver/tubes1-AI>