

Laporan Tugas Besar 1 IF3170 Inteligensi Artifisial
Semester Ganjil Tahun Ajaran 2024/2025

PENCARIAN SOLUSI DIAGONAL *MAGIC CUBE* DENGAN LOCAL SEARCH

Oleh:

13221011 JAZILA FAZA ALIYYA NURFAUZI

13221055 AHMAD HAFIDZ ALIIM

13221065 CAITLEEN DEVINA

18221130 RAYHAN MAHESWARA PRAMANDA

18321008 JASMINE CALLISTA AURELLIE IRFAN



**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
NOVEMBER 2024**

DAFTAR ISI

DAFTAR ISI	1
1. DESKRIPSI persoalan	2
2. PEMBAHASAN	2
a. PEMILIHAN OBJECTIVE FUNCTION	2
b. IMPLEMENTASI ALGORITMA LOCAL SEARCH	4
● Steepest Ascent Hill-climbing	4
● Hill-climbing with Sideways Move	10
● Random Restart Hill-climbing	14
● Stochastic Hill-climbing	16
● Simulated Annealing	18
● Genetic Algorithm	21
c. EKSPERIMEN SKEMA UNTUK TIAP LOCAL SEARCH	34
● Steepest Ascent Hill-climbing	34
● Hill-climbing with Sideways Move	39
● Random Restart Hill-climbing	45
● Stochastic Hill-climbing	49
● Simulated Annealing	56
● Genetic Algorithm	63
3. KESIMPULAN DAN SARAN	106
4. PEMBAGIAN TUGAS	107
5. REFERENSI	108

1. DESKRIPSI PERSOALAN

Diberikan sebuah *magic cube* berukuran $5 \times 5 \times 5$. Kondisi awal dari kubus berisi angka 1 hingga 5^3 secara acak tanpa pengulangan. Sebuah *magic cube* definisikan sebagai kubus yang memiliki kriteria berikut:

1. Terdapat satu *magic number* yang ditentukan lebih dahulu (*predetermined*), *magic number* tidak harus terdapat pada kubus dan tidak harus ada di dalam *range* 1 hingga 5^3
2. Angka-angka untuk setiap baris ketika dijumlahkan harus sesuai dengan *magic number*
3. Angka-angka untuk setiap kolom ketika dijumlahkan harus sesuai dengan *magic number*
4. Angka-angka untuk setiap tiang ketika dijumlahkan harus sesuai dengan *magic number*
5. Angka-angka untuk seluruh diagonal ruang ketika dijumlahkan harus sesuai dengan *magic number*
6. Angka-angka untuk seluruh diagonal bidang ketika dijumlahkan harus sesuai dengan *magic number*

Diperlukan algoritma *local search* yang mampu mengubah kondisi *cube* yang awalnya berisi angka acak menjadi kondisi yang memenuhi kriteria *magic cube*. Tiap iterasi algoritma diperbolehkan menukar posisi antara dua angka pada kubus (tidak harus pada bidang yang sama). 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).

2. PEMBAHASAN

a. PEMILIHAN OBJECTIVE FUNCTION

Fungsi objektif adalah fungsi yang didesain sebagai parameter kedekatan suatu kondisi terhadap kondisi ideal. Pada permasalahan ini, kami mendesain fungsi objektif sebagai berikut:

$$\begin{aligned} \text{Total Error} = & \sum_{i=1}^n |\text{Sum}_{\text{row}} - MN| + \sum_{j=1}^n |\text{Sum}_{\text{col}} - MN| + \sum_{k=1}^n |\text{Sum}_{\text{pil}} - MN| \\ & + \sum_{l=1}^n |\text{Sum}_{\text{diagFace}} - MN| + \sum_{l=1}^n |\text{Sum}_{\text{diagSpace}} - MN| \end{aligned}$$

Fungsi di atas akan menjumlahkan nilai absolut dari seluruh error pada tiap baris, kolom, pilar, diagonal bidang, maupun diagonal ruang. Nilai yang

didapat akan mencerminkan kondisi *state* secara menyeluruh (sesuai semua *constraint*). Fungsi tersebut dipilih dibandingkan *Sum-Squared Error* (SSE) karena meskipun SSE lebih cepat mencapai konvergensi (*error* besar akan lebih “terhukum”), tetapi rentan terjebak *local minima*. Oleh sebab itu, maka total *absolute* error menjadi fungsi objektif yang kami pilih. Nilai fungsi objektif yang merupakan solusi adalah 0. Artinya, semakin kecil nilai fungsi objektif, semakin baik konfigurasi kubus.

Algoritma untuk implementasi *objective function* dapat dituliskan sebagai berikut.

```
import numpy as np

# Determine Magic Number for N Cube
def magic_number(n):
    return (n * (n**3 + 1)) // 2

# Make a 3D cube with random unique numbers from 1..N^3
def make_cube(n):
    cube = np.arange(1, n**3 + 1)
    np.random.shuffle(cube)
    return cube.reshape((n, n, n))

# Objective function for calculating a magic cube state value
def objective_function(cube, magic_number):
    n = cube.shape[0]
    state_value = 0

    # Calculate difference for depth, rows, and columns
    for i in range(n):
        for j in range(n):
            state_value += abs(np.sum(cube[i, j, :]) -
magic_number)
            state_value += abs(np.sum(cube[i, :, j]) -
magic_number)
            state_value += abs(np.sum(cube[:, i, j])) -
magic_number)
```

```
# Calculate difference for face diagonals in each xy, xz,  
and yz plane  
for i in range(n):  
    # Main diagonals  
    state_value += abs(np.sum(cube[i, range(n),  
range(n)]) - magic_number)  
    state_value += abs(np.sum(cube[range(n), i,  
range(n)]) - magic_number)  
    state_value += abs(np.sum(cube[range(n), range(n),  
i]) - magic_number)  
  
    # Opposite diagonals  
    state_value += abs(np.sum(cube[i, range(n),  
range(n-1, -1, -1)]) - magic_number)  
    state_value += abs(np.sum(cube[range(n), i,  
range(n-1, -1, -1)]) - magic_number)  
    state_value += abs(np.sum(cube[range(n-1, -1, -1),  
range(n), i]) - magic_number)  
  
    # Calculate difference for all the space diagonals  
    state_value += abs(np.sum(cube[range(n), range(n),  
range(n)]) - magic_number)  
    state_value += abs(np.sum(cube[range(n), range(n),  
range(n-1, -1, -1)]) - magic_number)  
    state_value += abs(np.sum(cube[range(n), range(n-1, -1,  
-1), range(n)]) - magic_number)  
    state_value += abs(np.sum(cube[range(n-1, -1, -1),  
range(n), range(n)]) - magic_number)  
  
return state_value
```

b. IMPLEMENTASI ALGORITMA LOCAL SEARCH

- **Steepest Ascent Hill-climbing**

Algoritma Steepest Ascent Hill-climbing bekerja dengan melacak suatu keadaan atau *state* dan berpindah pada setiap iterasi ke arah *neighboring state* yang memiliki nilai fungsi objektif yang lebih baik. Algoritma ini melacak ke nilai yang lebih tinggi dan "pendakian" akan

berhenti saat mencapai puncak tertinggi dari *initial state*, sehingga disebut "steepest ascent" atau pendakian paling curam. Namun, dalam satu penelusuran fungsi objektif, tidak hanya terdapat satu puncak. Steepest Ascent Hill-climbing tidak mengakomodasi pencarian di luar "bukit" *initial state* sehingga nilai puncak yang didapat mungkin merupakan *local optima*, bukan *global optima*. Steepest Ascent Hill-climbing juga tidak mengakomodasi *shoulder*, yaitu grafik landai pada Gambar 1.1 sebab penelusuran tidak berlanjut ketika *neighboring state* bernilai sama dengan nilai saat ini.

Diketahui kubus berada di suatu *initial state* yaitu berisi kombinasi angka yang tersusun secara acak dari 1 hingga n^3 sebagai kondisi awal. Algoritma dimulai dengan membandingkan nilai *objective function* dari seluruh *successor*, yaitu seluruh kemungkinan konfigurasi dari penukaran posisi dua angka pada kubus. Lalu dipilih sebuah *neighbour* yaitu *successor* yang menghasilkan nilai *objective function* terbaik dari *successor* lain. Setelah itu, nilai *objective function* dari *neighbour* akan dibandingkan dengan nilai *objective function current state*. Apabila *objective function neighbour* memiliki nilai yang **lebih kecil**, maka algoritma akan **melanjutkan iterasi**. Algoritma akan berhenti dan **mengembalikan current state** ketika nilai *objective function neighbour* **tidak lebih kecil** daripada nilai *objective function current state*.

Algoritma *steepest ascent hill-climbing* dapat dituliskan sebagai berikut.

```
from magic_utils import objective_function

def hc_steepest_ascent(cube, magic_number):
    n = cube.shape[0]
    num_of_iteration = 0

    # Initialise the current state value
    current_state_value = objective_function(cube,
    magic_number)
    best_state_value = current_state_value

    # Initialise the best cube as the initial cube
```

```
best_cube = cube.copy()

while True:
    num_of_iteration += 1
    best_neighbour_state_value =
current_state_value
    best_neighbour_swap = None

    # Iterate over the search space for swapping
two cells
    # TODO: Optimize search algorithm complexity or
use parallel processing
    for i in range(n):
        for j in range(n):
            for k in range(n):
                for x in range(n):
                    for y in range(n):
                        for z in range(n):
                            if (i, j, k) != (x, y,
z):
                                # Swap two cells
                                cube[i, j, k],
cube[x, y, z] = cube[x, y, z], cube[i, j, k]

                            # Evaluate the
candidate neighbour state value

neighbour_state_value = objective_function(cube,
magic_number)

                            # Check if this is
the best swap so far
                            if
neighbour_state_value < best_neighbour_state_value:

best_neighbour_state_value = neighbour_state_value
```

```
best_neighbour_swap = (i, j, k), (x, y, z)

print(best_neighbour_state_value)

        # Revert the swap
        cube[i, j, k],
cube[x, y, z] = cube[x, y, z], cube[i, j, k]

        # Check if a better swap was found
        if best_neighbour_swap and
best_neighbour_state_value < current_state_value:
            (i, j, k), (x, y, z) = best_neighbour_swap

        # Perform the best swap
        cube[i, j, k], cube[x, y, z] = cube[x, y,
z], cube[i, j, k]
        current_state_value =
best_neighbour_state_value

        if best_neighbour_state_value <
best_state_value:
            best_state_value =
best_neighbour_state_value
            best_cube = cube.copy()
        else:
            break

return best_cube, best_state_value,
num_of_iteration
```

Algoritma ini bekerja dengan melakukan pencarian dengan *looping* pada variabel i, j, k, x, y, z. Kemudian, lakukan *swap* secara temporer, cek *state value*-nya, jika lebih baik dibandingkan *current best candidate neighbour* maka pertahankan. Apabila hasilnya lebih buruk, maka kembalikan kubus dengan konformasi yang lama dan lanjutkan pencarian. Kemudian, hasil *candidate neighbour* yang baru dibandingkan dengan *best state*, jika lebih baik ditukar, jika tidak maka akhiri algoritma. Algoritma ini memiliki *time complexity* yang sangat tinggi, yakni $O(n^6)$ sehingga sangat boros komputasi.

Algoritma untuk implementasi melihat visualisasinya adalah sebagai berikut.

```
import plotly.graph_objects as go
import numpy as np

# Fungsi untuk visualisasi state kubus
def visualize_magic_cube(cube, title="Magic Cube
Visualization"):

    n = cube.shape[0]
    x_position, y_position, z_position, text_values =
[], [], [], []

    for i in range(n):
        for j in range(n):
            for k in range(n):
                x_position.append(i)
                y_position.append(j)
                z_position.append(k)
                text_values.append(str(cube[i, j, k]))

    fig = go.Figure(data=[

        go.Scatter3d(
            x=x_position,
            y=y_position,
            z=z_position,
            mode='markers',
            marker=dict(size=5,
```

```
color=np.arange(len(text_values)),
colorscale='Viridis', opacity=0.6)
),
go.Scatter3d(
    x=x_position,
    y=y_position,
    z=z_position,
    mode='text',
    text=text_values,
    textposition="top center",
    textfont=dict(size=10, color="black"),
)
])

fig.update_layout(scene=dict(
    xaxis=dict(nticks=n, range=[-0.5, n - 0.5]),
    yaxis=dict(nticks=n, range=[-0.5, n - 0.5]),
    zaxis=dict(nticks=n, range=[-0.5, n - 0.5]),
    aspectmode="cube"
), title=title)

fig.show()
```

```
n = 5
magic_num = magic_number(n)
initial_cube = make_cube(n)

# Visualize Initial State
visualize_magic_cube(initial_cube, title="Initial State
of Magic Cube")

max_restarts = 1
final_cube, best_state_value, num_of_iterations,
state_values = hc_steepest_ascent(initial_cube,
magic_num)

# Print State Value
```

```
print(state_values)

# Visualize Final State
visualize_magic_cube(final_cube, title="Final State of
Magic Cube")
```

- **Hill-climbing with Sideways Move**

Algoritma Hill-climbing with Sideways Move bekerja dengan melacak suatu keadaan atau *state* dan berpindah pada setiap iterasi ke arah *neighboring state* yang memiliki nilai fungsi objektif yang lebih baik atau sama dengan nilai fungsi objektif keadaan saat ini. Berbeda dengan algoritma Hill-climbing jenis Steepest Ascent, algoritma Hill-climbing with Sideways Move melacak ke nilai yang lebih tinggi atau sama dengan nilai saat ini. Dengan itu, Hill-climbing with Sideways Move mengakomodasi *shoulder* karena dapat bergerak ke nilai yang sama dengan nilai sebelumnya. "Pendakian" akan berhenti saat mencapai puncak tertinggi dari *initial state*. Steepest Ascent Hill-climbing juga tidak mengakomodasi pencarian di luar "bukit" *initial state* sehingga nilai puncak yang didapat mungkin merupakan *local optima*, bukan *global optima*.

Diketahui kubus berada di suatu *initial state* yaitu berisi kombinasi angka yang tersusun secara acak dari 1 hingga n^3 sebagai kondisi awal. Algoritma dimulai dengan membandingkan nilai *objective function* dari seluruh *successor*, yaitu seluruh kemungkinan konfigurasi dari penukaran posisi dua angka pada kubus. Lalu dipilih sebuah *neighbour* yaitu *successor* yang menghasilkan nilai *objective function* terbaik dari *successor* lain. Setelah itu, nilai *objective function* dari *neighbour* akan dibandingkan dengan nilai *objective function current state*. Apabila *objective function neighbour* memiliki nilai yang **lebih kecil atau sama** dengan nilai *objective function current state*, maka algoritma akan **melanjutkan iterasi**. Algoritma akan berhenti dan **mengembalikan current state** ketika nilai *objective function neighbour* **lebih besar** daripada nilai *objective function current state*.

Algoritma *hill-climbing with sideways move* dapat dituliskan sebagai berikut.

```
def hc_sideways_move(cube, magic_number,
max_sideways_moves):
    n = cube.shape[0]

    # Initialise the current state value
    current_state_value = objective_function(cube,
magic_number)
    best_state_value = current_state_value

    # Initialise the best cube as the initial cube
    best_cube = cube.copy()

    sideways_moves = 0

    while True:
        best_neighbour_state_value =
current_state_value
        best_neighbour_swap = None

        # Iterate over the search space for swapping
        two cells
        for i in range(n):
            for j in range(n):
                for k in range(n):
                    for x in range(n):
                        for y in range(n):
                            for z in range(n):
                                if (i, j, k) != (x, y,
z):
                                    # Swap two cells
                                    cube[i, j, k],
cube[x, y, z] = cube[x, y, z], cube[i, j, k]

                                    # Evaluate the
candidate neighbour state value

neighbour_state_value = objective_function(cube,
```

```
magic_number)

                                # Check if this is
the best swap so far

                if
neighbour_state_value < best_neighbour_state_value:

best_neighbour_state_value = neighbour_state_value

best_neighbour_swap = (i, j, k), (x, y, z)

                                # Revert the swap
cube[i, j, k],
cube[x, y, z] = cube[x, y, z], cube[i, j, k]

                                # Check if a swap with better or equally good
state value was found

                if best_neighbour_swap and
best_neighbour_state_value <= current_state_value:
                    # Check if the best neighbour is better
than the current state

                    if best_neighbour_state_value <
current_state_value:
                        sideways_moves = 0    # Reset the counter
                    else:
                        sideways_moves += 1

                                # End search if the maximum sideways moves
has been reached

                if sideways_moves >= max_sideways_moves:
                    break

(i, j, k), (x, y, z) = best_neighbour_swap

                                # Perform the best swap
cube[i, j, k], cube[x, y, z] = cube[x, y,
z], cube[i, j, k]
```

```
        current_state_value =
best_neighbour_state_value

        if best_neighbour_state_value <
best_state_value:
            best_state_value =
best_neighbour_state_value
            best_cube = cube.copy()
        else:
            break

    return best_cube, best_state_value
```

Algoritma di atas bekerja dengan cara yang sangat mirip dengan *hill climbing steepest ascent*, hanya saja algoritma ini dapat menerima solusi yang setara dengan *best state value* sebagai *neighbour*. Pada algoritma ini, kami juga menerapkan batas maksimum algoritma boleh melakukan *sideways move* **berturut-turut**.

Algoritma untuk implementasi melihat visualisasinya adalah sebagai berikut dengan fungsi visualisasi sama dengan di steepest.

```
n = 5
magic_num = magic_number(n)
initial_cube = make_cube(n)

# Visualize Initial State
visualize_magic_cube(initial_cube, title="Initial State
of Magic Cube")

max_sideways_moves = 2
cube_result, best_state_value, state_values =
hc_sideways_move(initial_cube, magic_num,
max_sideways_moves)

# Print State Value
print(state_values)
```

```
# Visualize Final State
visualize_magic_cube(cube_result, title="Final State of
Magic Cube")
```

- **Random Restart Hill-climbing**

Algoritma Random Restart Hill-climbing berjalan seperti Algoritma Steepest Ascent Hill-climbing atau Hill-climbing with Sideways Move, dipilih penggunaan Hill-climbing with Sideways Move sebab memiliki peluang keberhasilan yang lebih baik berdasarkan penjelasan sebelumnya. Namun seperti yang telah dibahas sebelumnya, Algoritma *local search* rentan terjebak di *local minima/maxima*, *shoulder*, maupun “*flat*” *local minima/maxima*. Oleh sebab itu, ketika satu siklus algoritma dilakukan sampai mendapat *local minima*, dilakukan *restart* yaitu mengulangi algoritma dari tahap *random initial state* dengan tujuan mencari *global minima* yaitu nilai *objective function* total error terendah yaitu 0. Restart dapat dilakukan sebanyak q kali, dengan q ditentukan sebelumnya, maupun dilakukan terus-menerus sampai ditemukan *global minima*. Nilai q didapat dari perhitungan $q = 1/p$ dengan p adalah probabilitas keberhasilan. Nilai p didapat melalui eksperimen menggunakan Hill-climbing with Sideways Move.

Algoritma *random restart hill-climbing* dapat dituliskan sebagai berikut.

```
def hc_random_restart(cube, magic_number,
max_restarts):
    num_of_restart = 0
    num_of_iterations = []

    # Initialise the best state value and cube
    best_state_value = objective_function(cube,
magic_number)
    best_cube = cube.copy()

    # Repeat the random restart for n times
    while (num_of_restart < max_restarts):
        # Randomly shuffle the cube
```

```
np.random.shuffle(cube)

cube, state_value, num_of_iteration =
hc_steepest_ascent(cube, magic_number)

num_of_iterations.append(num_of_iteration)

# Update the best state value
if state_value < best_state_value:
    best_state_value = state_value
    best_cube = cube.copy()

num_of_restart += 1

return best_cube, best_state_value,
num_of_iterations
```

Algoritma di atas bekerja dengan cara yang sangat serupa dengan algoritma *hill climbing steepest ascent*, bahkan kami memanggil ulang fungsi tersebut. Perbedaannya hanya kami melakukan *shuffle* pada kubus tiap kali dilakukan *restart*, menyimpan jumlah iterasi tiap *restart*, dan membatasi jumlah *restart*. Sisanya sama dengan *hill climbing steepest ascent*.

Algoritma untuk implementasi melihat visualisasinya adalah sebagai berikut dengan fungsi visualisasi sama dengan di steepest.

```
n = 5
magic_num = magic_number(n)
initial_cube = make_cube(n)

# Visualize Initial State
visualize_magic_cube(initial_cube, title="Initial State
of Magic Cube")

max_restarts = 1
final_cube, best_state_value, all_state_values =
hc_random_restart(initial_cube, magic_num,
```

```
max_restarts)

# Print State Value
print(state_values)

# Visualize Final State
visualize_magic_cube(final_cube, title="Final State of
Magic Cube")
```

- **Stochastic Hill-climbing**

Algoritma Stochastic Hill Climbing memilih *neighbor state* secara acak, bukan menelusuri seluruh ruang pencarian yang ada. Diketahui kubus berada di suatu *initial state* yaitu berisi kombinasi angka yang tersusun secara acak dari 1 hingga n^3 sebagai kondisi awal. Diketahui pula algoritma akan melakukan iterasi sebanyak q kali. Algoritma dimulai dengan memilih sebuah *random successor* sebagai *neighbour*, yaitu konfigurasi baru dari pertukaran posisi dua angka pada kubus secara acak. Setelah itu, nilai *objective function* dari *neighbour* akan dibandingkan dengan nilai *objective function current state*. Apabila *objective function neighbour* baru memiliki nilai yang lebih kecil daripada nilai *objective function current state*, maka algoritma akan memilih konfigurasi *neighbour* sebagai *current state* baru. Apabila algoritma sudah melaksanakan iterasi sebanyak q kali, maka algoritma akan berhenti dan mengembalikan konfigurasi dari *current state*.

Algoritma *stochastic hill-climbing* dapat dituliskan sebagai berikut.

```
def hc_stochastic(cube, magic_number, max_iteration):
    n = cube.shape[0]

    # Initialise the current state value
    best_state_value = objective_function(cube,
    magic_number)

    for i in range(max_iteration):
        # Generate random indices for swapping two
```

```
cells

    i, j, k = random.randint(0, n-1),
random.randint(0, n-1), random.randint(0, n-1)
    x, y, z = random.randint(0, n-1),
random.randint(0, n-1), random.randint(0, n-1)

    # Swap two cells
    cube[i, j, k], cube[x, y, z] = cube[x, y, z],
cube[i, j, k]

    # Evaluate the candidate neighbour state value
    neighbour_state_value =
objective_function(cube, magic_number)

    # Check the neighbour state value is better
    than the current best state value
    if neighbour_state_value < best_state_value:
        best_state_value = neighbour_state_value
    else:
        # Revert the swap
        cube[i, j, k], cube[x, y, z] = cube[x, y,
z], cube[i, j, k]

return cube, best_state_value
```

Algoritma di atas bekerja dengan menentukan panjang sisi kubus (n), menghitung **best_state_value** dari kubus awal, kemudian menginisiasi *array* untuk menyimpan *state* dari kubus pada tiap iterasi. Lalu, hasilkan angka acak untuk indeks i, j, k, x, y, z dan tukar posisinya. Panggil fungsi objektif untuk mengevaluasi *candidate neighbour*, jika lebih baik maka pertahankan, jika tidak maka kembalikan *swap*. Tidak lupa, fungsi juga menyimpan *state* dari tiap iterasi. Terakhir, kembalikan kubus terbaik, nilai *state value* terbaik, dan larik berisi *state* dari tiap iterasi.

Algoritma untuk implementasi melihat visualisasinya adalah sebagai berikut dengan fungsi visualisasi sama dengan di *steepest*.

```
n = 5
magic_num = magic_number(n)
initial_cube = make_cube(n)

# Visualize Initial State
visualize_magic_cube(initial_cube, title="Initial State
of Magic Cube")

num_of_iterations = 15000
final_cube, final_state_value, state_values =
hc_stochastic(initial_cube, magic_num,
num_of_iterations)

# Print State Value
print(state_values)

# Visualize Final State
visualize_magic_cube(final_cube, title="Final State of
Magic Cube")
```

- **Simulated Annealing**

Simulated Annealing merupakan algoritma yang mengadopsi proses *annealing* yaitu proses memanaskan logam, menahannya pada temperatur tinggi, kemudian menurunkan temperaturnya secara bertahap untuk memperkuat struktur logam tersebut. Algoritma *Simulated Annealing* memiliki persamaan probabilitas umum yaitu

$P = e^{\Delta E/T}$ dengan P adalah probabilitas, ΔE adalah selisih *objective function* konfigurasi saat ini dengan konfigurasi baru, dan T adalah "temperatur" yang diinisialisasi dengan nilai tinggi dan turun secara bertahap. *Current value* akan digantikan oleh *neighbor value* apabila probabilitasnya melebihi *threshold* (ambang batas) yang telah ditetapkan sebelumnya.

Diketahui kubus berada di suatu *initial state* yaitu berisi kombinasi angka yang tersusun secara acak dari 1 hingga n^3 sebagai kondisi awal. Algoritma dimulai dengan menginisialisasi nilai temperatur (T) dan threshold. Selanjutnya dipilih sebuah *random successor* sebagai *neighbour*, yaitu konfigurasi baru dari penukaran posisi dua angka pada kubus secara acak. Setelah itu, dihitung nilai *objective function* dari *neighbour state* dan *current state*. Dihitung pula ΔE dari *current value* dikurangi dengan *neighbour value* (karena diharapkan nilai error rendah). Jika *neighbour value* lebih baik daripada *current value* ($\Delta E > 0$), maka algoritma akan memilih konfigurasi *neighbour* sebagai *current state* baru. Apabila *neighbour value* tidak lebih baik daripada *current value* ($\Delta E \leq 0$), akan dihitung probabilitas $P = e^{\Delta E/T}$ dengan T adalah temperatur saat ini. Apabila probabilitas P melebihi *threshold*, maka algoritma akan memilih konfigurasi *neighbour* sebagai *current state* baru. Namun, apabila P tidak melebihi *threshold*, maka algoritma akan melanjutkan iterasi disertai pengurangan nilai T.

Simulated Annealing memungkinkan untuk *escape* dari *local minima/maxima* karena memungkinkan untuk pindah ke *neighbour* dengan *value* yang lebih buruk (error yang lebih tinggi), namun frekuensinya akan berkurang seiring berkurangnya T.

Algoritma *simulated annealing* dapat dituliskan sebagai berikut.

```
# Implementasi Algoritma Simulated Annealing
def simulated_annealing(cube, magicNumber, temperature,
cooling_rate, iterations, threshold):
    n = cube.shape[0]
    current_cube = cube.copy()
    current_score = objective_function(current_cube,
magicNumber)
    best_cube = current_cube.copy()
```

```
best_score = current_score

for _ in range(iterations):
    # Pilih dua posisi acak untuk ditukar
    i, j, k = np.random.randint(0, n, size=3)
    x, y, z = np.random.randint(0, n, size=3)

    # Tukar elemen
    current_cube[i, j, k], current_cube[x, y, z] =
current_cube[x, y, z], current_cube[i, j, k]

    # Hitung nilai objective function baru
    new_score = objective_function(current_cube,
magicNumber)

    # Hitung delta E dan probabilitas
    delta_E = current_score - new_score
    probability = math.exp(min(max(delta_E /
temperature, -700), 700)) # Dibatasi untuk rentang
[-700, 700]

    # Tentukan apakah menerima solusi baru
(corrected Metropolis criterion)
    if delta_E > 0 or probability > threshold:
        current_score = new_score
        if new_score < best_score:
            best_cube = current_cube.copy()
            best_score = new_score
        else:
            # Kembalikan ke kondisi sebelumnya jika
tidak diterima
            current_cube[i, j, k], current_cube[x, y,
z] = current_cube[x, y, z], current_cube[i, j, k]

    # Kurangi temperatur
    temperature *= cooling_rate
```

```
return best_cube, best_score
```

Dalam mencari probabilitas $P = e^{\Delta E/T}$, nilai $\Delta E/T$ dibatasi pada rentang [-700, 700] untuk menghindari overflow error akibat batas maksimum kapasitas floating-point untuk tipe float64 di Python yaitu sekitar 1.03e+308, sekitar math.exp(709).

- **Genetic Algorithm**

Algoritma genetika adalah algoritma *local search* yang didasarkan pada mekanisme evolusi/seleksi alami di dunia nyata. Pada algoritma genetika, terdapat dua mekanisme utama, yakni *cross-over* dan *random mutation*. Kondisi diawali dengan inisialisasi sejumlah individu (populasi) yang memiliki beragam *state*. Lalu, masing-masing individu dihitung *fitness function*-nya (istilah *objective function* di algoritma genetika). Selanjutnya, akan dipilih sejumlah individu dalam populasi sebagai *parent* melalui mekanisme selection. Metode selection yang umum digunakan adalah Roulette Wheel Selection, dimana semakin tinggi *fitness function*-nya, semakin tinggi kemungkinan individu tersebut terpilih sebagai *parent* (berarti ada pula individu yang terpilih >1 kali maupun tidak terpilih sama sekali). Selanjutnya, dilakukan mekanisme *cross-over* yaitu mengkombinasikan sebagian gen dari *parent* pertama dan sebagian gen dari *parent* kedua untuk menghasilkan individu baru. Setelah *cross-over*, dilakukan mekanisme mutasi, yaitu perubahan pada sebuah gen secara acak. Setelah individu baru terbentuk, dilakukan evaluasi menggunakan *objective function* yang sama. Algoritma berhenti ketika tercapai kondisi terminasi, dapat berupa menemukan solusi, mencapai jumlah generasi tertentu, atau kondisi lain yang ditentukan sebelumnya.

Dalam persoalan Diagonal Magic Cube, individu berupa *array of integer* berisi kombinasi angka yang tersusun secara acak dari 1 hingga n^3 yang mewakili sebuah kubus. Selanjutnya, seluruh individu sejumlah yang ditentukan sebelumnya akan dievaluasi *fitness function* atau *objective function*-nya. Setelah penentuan nilai *fitness function* masing-masing individu, akan dilakukan *parent selection* menggunakan metode Roulette Wheel Selection. Selanjutnya, dilakukan *cross-over* pada *cross-over point* masing-masing *parent* dan menghasilkan individu baru. Algoritma dilanjutkan dengan

mekanisme *swap mutation*, yaitu menukar dua angka pada *array* untuk menjaga angka tetap muncul satu kali pada rentang 1 hingga n^3 . Algoritma akan berhenti ketika tercapai *threshold objective function* yang ditentukan di awal program.

Algoritma *genetic algorithm* dapat dituliskan sebagai berikut.

```
import plotly.graph_objects as go
import numpy as np

# Fungsi untuk visualisasi state kubus
def visualize_magic_cube(cube, title="Magic Cube Visualization"):
    n = cube.shape[0]
    x_position, y_position, z_position, text_values = [], [], [], []

    for i in range(n):
        for j in range(n):
            for k in range(n):
                x_position.append(i)
                y_position.append(j)
                z_position.append(k)
                text_values.append(str(cube[i, j, k]))

    fig = go.Figure(data=[
        go.Scatter3d(
            x=x_position,
            y=y_position,
            z=z_position,
            mode='markers',
            marker=dict(size=5,
color=np.arange(len(text_values)), colorscale='Viridis',
opacity=0.6),
        ),
        go.Scatter3d(
            x=x_position,
            y=y_position,
            z=z_position,
            mode='text',
            text=text_values,
            textposition="top center",
            textfont=dict(size=10, color="black"),
        )
    ])

    fig.update_layout(scene=dict(
        xaxis=dict(nticks=n, range=[-0.5, n - 0.5]),
        yaxis=dict(nticks=n, range=[-0.5, n - 0.5]),
        zaxis=dict(nticks=n, range=[-0.5, n - 0.5]),
        aspectmode="cube"
    ), title=title)
```

```
fig.show()
```

```
import numpy as np
import random
import math
import time
from visualize import visualize_magic_cube

# crossover method: Order Crossover (OX)
# mutation method: swapping between 2 element
def ga():
    start_time = time.time()
    # define
    max_iteration = 1000
    cube_side = 5           # number of cube's side
    max_population = 500     # max population number,
    GA start with max population
    mutation_probability = 20   # probability of gene
    mutation base 100
    states_for_vid_player = []

    def generate_random(N):
        # Calculate the total number of elements
        total_elements= N ** 3

        # Generate a list of integers from 1 to N^3
        numbers = list(range(1, total_elements + 1))

        # Shuffle the list to randomize the order of
        the integers
        random.shuffle(numbers)

        # Create a 3D matrix and fill it with the
        shuffled integers
        matrix_3d = np.zeros((N, N, N), dtype=int)
```

```
index = 0
for i in range(N):
    for j in range(N):
        for k in range(N):
            matrix_3d[i][j][k] = numbers[index]
            index += 1

return matrix_3d

def calculate_posibilities(population):
    possibilities = []
    max_value = 0

    for i in range(max_population):
        max_value += population[i][1]

    for i in range(max_population):
        value =
math.floor(population[i][1]/max_value * 100)
        possibilities.append(value)

    return possibilities

# Fungsi untuk menghitung Magic Number untuk kubus
n x n x n
def MagicNumber(n):
    return (n * (n**3 + 1)) // 2

# Fungsi untuk menghitung value dari suatu state
kubus -> smaller better
def ObjectiveFunction(cube, magic_number):
    n = cube.shape[0]
    state_value = 0

    # Calculate difference for rows, columns, and
depths
```

```
        for i in range(n):
            for j in range(n):
                # Sum across depth (z-axis), rows
                # (y-axis), and columns (x-axis)
                state_value += abs(np.sum(cube[i, j,
                :]) - magic_number)
                state_value += abs(np.sum(cube[i, :, j]) - magic_number)
                state_value += abs(np.sum(cube[:, i, j]) - magic_number)

                # Calculate difference for face diagonals in
                # each xy, xz, and yz plane
                for i in range(n):
                    state_value += abs(np.sum(cube[i, range(n),
                    range(n)]) - magic_number)
                    state_value += abs(np.sum(cube[range(n), i,
                    range(n)]) - magic_number)
                    state_value += abs(np.sum(cube[range(n),
                    range(n), i]) - magic_number)
                    # Secondary diagonals
                    state_value += abs(np.sum(cube[i, range(n),
                    range(n-1, -1, -1)]) - magic_number)
                    state_value += abs(np.sum(cube[range(n), i,
                    range(n-1, -1, -1)]) - magic_number)
                    state_value += abs(np.sum(cube[range(n-1,
                    -1, -1), range(n), i]) - magic_number)

                # Calculate difference for 4 main space
                # diagonals
                state_value += abs(np.sum(cube[range(n),
                range(n), range(n)]) - magic_number)
                state_value += abs(np.sum(cube[range(n),
                range(n), range(n-1, -1, -1)]) - magic_number)
                state_value += abs(np.sum(cube[range(n),
                range(n-1, -1, -1), range(n)]) - magic_number)
                state_value += abs(np.sum(cube[range(n-1,
                -1, -1), -1, -1])) - magic_number)
```

```
-1), range(n), range(n)]) - magic_number)

    return state_value


# initial begin
max_number = cube_side ** 3
magicNumber = MagicNumber(cube_side)
new_population = []
all_population = []
for i in range(max_population):
    random_chromosom = generate_random(cube_side)
    chromosom_value =
ObjectiveFunction(random_chromosom, magicNumber)
    if chromosom_value == 0:
        print("best possible case found when first
generating\n")
        print(random_chromosom)
        SystemExit()
    new_population.append((random_chromosom,
chromosom_value))
    new_population = sorted(new_population, key=lambda
x: x[1], reverse=False)

    visualize_magic_cube(new_population[0][0], f"best
of initialize iteration from {max_population}
population(s) with value: {new_population[0][1]}")

    print("best value at iteration 0 =",
new_population[0][1])

    for a in range(max_iteration):
        possibilities =
calculate_possibilities(new_population)

        generation = []
        new_generation = []
```

```
        for i in range(max_population):
            roulette = random.randint(0, 100)
            value = 0
            for j in range(max_population):
                value += new_population[j][1]
                if (value <= roulette) or (j ==
(max_population - 1)):

        generation.append(new_population[j][0])
            break

        # crossover + mutation
        for i in range(0, max_population, 2):
            ## crossover
            #initiate
            crossover_point_min = random.randint(0,
max_number - 1)
            crossover_point_max =
random.randint(crossover_point_min, max_number - 1)

            parent_1 = generation[i].flatten()
            parent_2 = generation[i + 1].flatten()

            child_1 = np.zeros((max_number), dtype=int)
            child_2 = np.zeros((max_number), dtype=int)

            child_1 -= 1
            child_2 -= 1

            #child 1
            child_1[crossover_point_min :
crossover_point_max] = parent_1[crossover_point_min :
crossover_point_max]
            pointer_parent = crossover_point_max
            pointer_child = crossover_point_max
            for j in range(max_number):
                if parent_2[pointer_parent] not in
```

```
child_1:
    child_1[pointer_child] =
parent_2[pointer_parent]
    pointer_child = (pointer_child + 1)
% (max_number)

    pointer_parent = (pointer_parent + 1) %
(max_number)

#child 2
child_2[crossover_point_min :
crossover_point_max] = parent_2[crossover_point_min :
crossover_point_max]
    pointer_parent = crossover_point_max
    pointer_child = crossover_point_max
    for j in range(max_number):
        if parent_1[pointer_parent] not in
child_2:
    child_2[pointer_child] =
parent_1[pointer_parent]
    pointer_child = (pointer_child + 1)
% (max_number)

    pointer_parent = (pointer_parent + 1) %
(max_number)

## mutation
# does child 1 mutate?
roulette = random.randint(0, 100)
if roulette < mutation_probability:
    mutation_point_1 = random.randint(0,
max_number - 1)
    mutation_point_2 = random.randint(0,
max_number - 1)

    # Swap the elements in the flattened
array
```

```
        child_1[mutation_point_1],
child_1[mutation_point_2] = child_1[mutation_point_2],
child_1[mutation_point_1]

        # does child 2 mutate?
roulette = random.randint(0, 100)
if roulette < mutation_probability:
    mutation_point_1 = random.randint(0,
max_number - 1)
    mutation_point_2 = random.randint(0,
max_number - 1)

        # Swap the elements in the flattened
array
        child_2[mutation_point_1],
child_2[mutation_point_2] = child_2[mutation_point_2],
child_2[mutation_point_1]

        # insert to new generation

new_generation.append(child_1.reshape((cube_side,
cube_side, cube_side)))

new_generation.append(child_2.reshape((cube_side,
cube_side, cube_side)))

for i in range(max_population):
    chromosom_value =
ObjectiveFunction(new_generation[i], magicNumber)

    if chromosom_value == 0:
        print("best possible case found in", a,
"iteration\n")
        print(new_generation[i])
        SystemExit()
```

```
        all_population.append((new_generation[i],
chromosom_value))

        # add the old population
        all_population += new_population

        all_population = sorted(all_population,
key=lambda x: x[1], reverse=False)

        new_population =
all_population[0:max_population]

        # print("best value at iteration", a, "=",
new_population[0][1])

states_for_vid_player.append(new_population[0].flatten())

        finish_time = time.time()
        elapsed_time = finish_time - start_time

        print(f"After {max_iteration} iteration(s) in
{elapsed_time:.6f} second(s) resulting in its best
value of {new_population[0][1]} (smaller better). The
magic cube is as follows:\n")
        print(new_population[0][0]) #print as matrix

        visualize_magic_cube(new_population[0][0], f"best
last iteration with value: {new_population[0][1]}")

        return new_population[0][0], new_population[0][1],
states_for_vid_player
```

- **Video Player**

Dalam rangka menunjukkan proses iterasi dari tiap algoritma *local search*, kami juga membuat sebuah *video player*. Fitur ini dibuat dengan memanfaatkan pustaka Plotly yang juga kami gunakan untuk

visualisasi dari kondisi kubus. Berikut merupakan kode implementasi terkait.

```
# Write states to file
def write_states_to_file(states, filename):
    state_dict = {}
    for i in range(len(states)):
        state_dict[str(i)] = states[i]
    np.savez(filename, **state_dict)

# Read states from file
def read_states_from_file(filename):
    with np.load(filename) as data:
        states = [data[key] for key in data]
    return states
```

Fungsi penulisan dan pembacaan di atas menggunakan fungsi **load** dan **savez** dari numpy. Fungsi *write* menerima *states* berupa *array of 1D array* dan string berupa *filename* (nama berkas). Fungsi *write* bekerja dengan membuat *dictionary*, kemudian melakukan *looping* untuk memindahkan isi *states* ke *dictionary* tersebut. Terakhir, dilakukan pemanggilan fungsi *savez* dengan *unpacking dictionary* yang ada. Kemudian, pada fungsi *read*, dilakukan pembukaan berkas sebagai data, kemudian digunakan *list comprehension* untuk mengambil data dari *dictionary* ke sebuah *array states*.

Penggunaan **load** dan **savez** dipilih karena pada dasarnya banyak fungsi program ini yang bekerja berbasis Numpy sehingga akan mempermudah. Proses penulisan dilakukan dengan **savez** dan bukan *save* karena bersifat *multiple-array*.

```
def video_player(states, n, title="Magic Cube State
Evolution", initial_duration=100):
    # Reshape the states to 3D cube since the states are
    flattened
    states_reshaped = [np.array(state).reshape(n, n, n) for
state in states]

    # Initialise the frame array
    frames = []

    # Enumerate for all state in states
    for state_index, cube in enumerate(states_reshaped):
        x_position, y_position, z_position, text_values =
[], [], [], []
```

```
# Append the position of each cell to the
respective axis
for i in range(n):
    for j in range(n):
        for k in range(n):
            x_position.append(i)
            y_position.append(j)
            z_position.append(k)
            text_values.append(str(cube[i, j, k]))

# Create the frame for each state
frame = go.Frame(
    data=[
        go.Scatter3d( # Create scatter plot as
bullet data point
            x=x_position,
            y=y_position,
            z=z_position,
            mode='markers',
            marker=dict(size=5,
color=np.arange(len(text_values)), colorscale='Viridis',
opacity=0.3)
        ),
        go.Scatter3d( # Create scatter plot for
text data point
            x=x_position,
            y=y_position,
            z=z_position,
            mode='text',
            text=text_values,
            textposition="top center",
            textfont=dict(size=10, color="black")
        )
    ],
    name=state_index
)
frames.append(frame) # Append the frame

# Create the figure with initial state
fig = go.Figure(
    data=frames[0].data,
    frames=frames,
    layout=go.Layout(
        scene=dict(
            xaxis=dict(nticks=n, range=[-0.5, n - 0.5],
title="X"),
            yaxis=dict(nticks=n, range=[-0.5, n - 0.5],
title="Y"),
            zaxis=dict(nticks=n, range=[-0.5, n - 0.5],
title="Z"),
            aspectmode="cube" # Set the aspect mode to
cube
        )
    )
)
```

```
        ),
        title=dict(
            text=title,
            x=0.5,
            y=0.8
        ),
        # Initialise menu for the buttons
        updatemenus=[{
            buttons
            'type': 'buttons',
            'showactive': True,
            # Define the relative position of the
            'x': 0.05,
            'y': 0.2,
            'buttons': [
                {
                    'label': 'Play',
                    'method': 'animate', # Method for
button to enable start/stop, check Plotly documentation for
more details
                    'args': [None, {
                        'frame': {'duration':
initial_duration, 'redraw': True},
                        'fromcurrent': True,
                    }]
                },
                {
                    'label': 'Pause',
                    'method': 'animate', # Method for
button to enable start/stop, check Plotly documentation for
more details
                    'args': [[None], {
                        'frame': {'duration': 0,
'redraw': False},
                        'mode': 'immediate',
                    }]
                }
            ],
            # Initialise slider
            sliders=[{
                'currentvalue': {
                    'prefix': 'State: ',
                    'visible': True
                },
                # Define the steps for the slider
                'steps': [
                    {
                        'label': f'{i}', # Label for each
step (what will be shown on the slider)
                        'method': 'animate',
                        'args': [[i], { # What the step
will be based on, in this case it's the state_index (see
above)
                    }
                ]
            }
        ]
    }
]
```

```
        'frame': {'duration': 0,
'redraw': True},
            'mode': 'immediate',
        }
    }
    for i in range(len(states)) # Iterate
over the number of states
    ]
}
)
)

return fig
```

Kemudian, pada fungsi *video_player* di atas dapat dilihat bahwa kami menggunakan pustaka Plotly. Fungsi *video_player* menerima masukan berupa *array of 1D array* bernama *states*, ukuran kubus (*n*), judul visualisasi, dan durasi video. Fungsi ini akan melakukan *reshape* dari *array 1D* menjadi *array 3 dimensi* berukuran *nxnxn*. Lalu, fungsi akan memasukan data posisi di tiap-tiap sumbu untuk setiap *data point*. Setelahnya, dibuatlah *frame*: berupa titik data transparan dan angka pada ruang. *Frame* tadi akan dibuat sejumlah data yang ada di *states_reshaped* dan dimasukan ke dalam *array frames*. Lalu, dibuat Figure yang berisi tata letak keseluruhan, mulai dari latar berbentuk kubus, tombol-tombol terkait, dan sebagainya. *Slider* dibuat *step*-nya berdasarkan **state_index** yang didapat dari proses enumerasi di atas.

c. EKSPERIMEN SKEMA UNTUK TIAP LOCAL SEARCH

- **Steepest Ascent Hill-climbing**

Berikut hasil eksperimen yang dilakukan untuk *steepest ascent hill-climbing*.

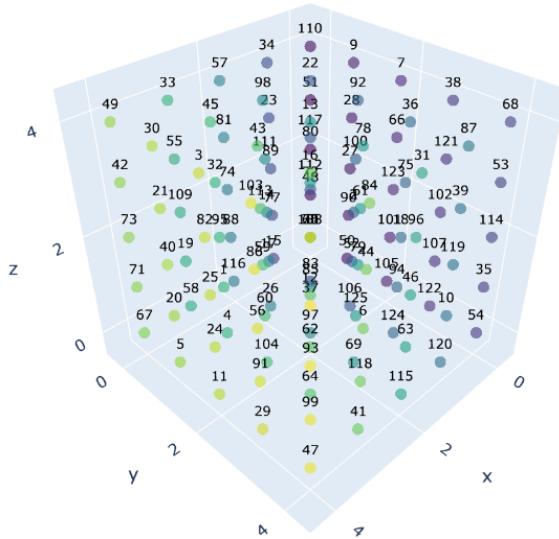
Percobaan 1

Iterasi = 593

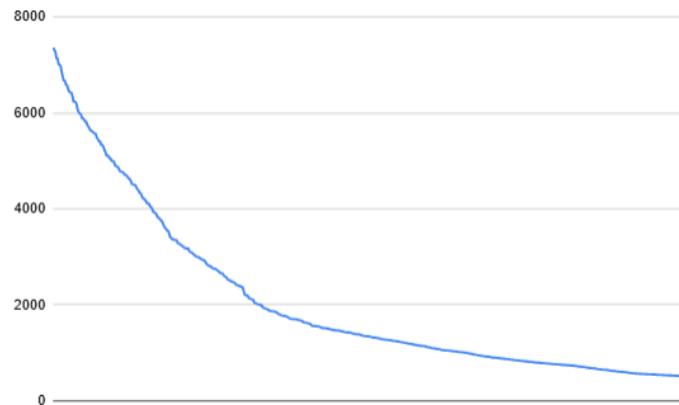
Mendapatkan state value = 518

Waktu yang dibutuhkan = 25 menit

State Awal



Gambar 1. Percobaan 1 Initial State



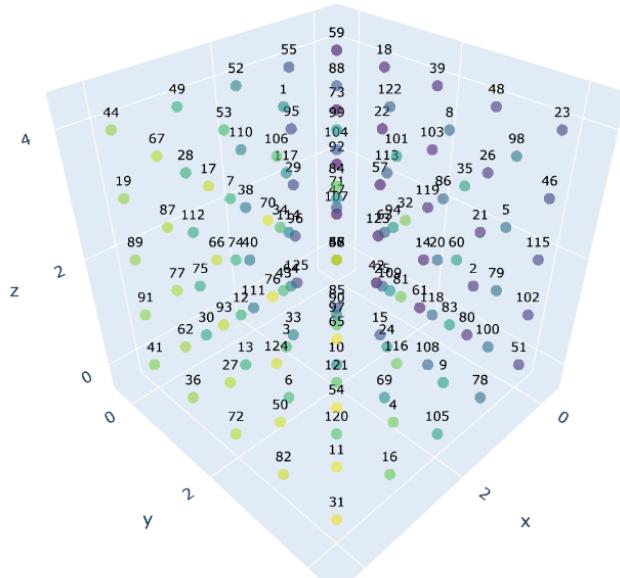
Gambar 2. Percobaan 1 Plot State Value

Percobaan 2

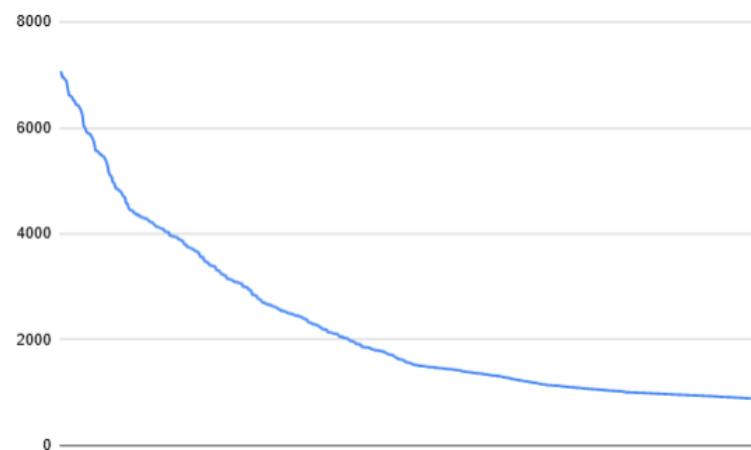
Iterasi = 524

Mendapatkan state value = 890

Waktu yang dibutuhkan = 36 menit



Gambar 3. Percobaan 2 Initial State



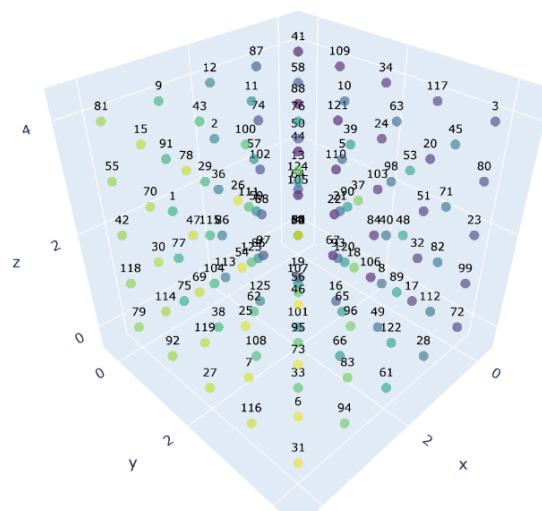
Gambar 4. Percobaan 2 Plot State Value

Percobaan 3

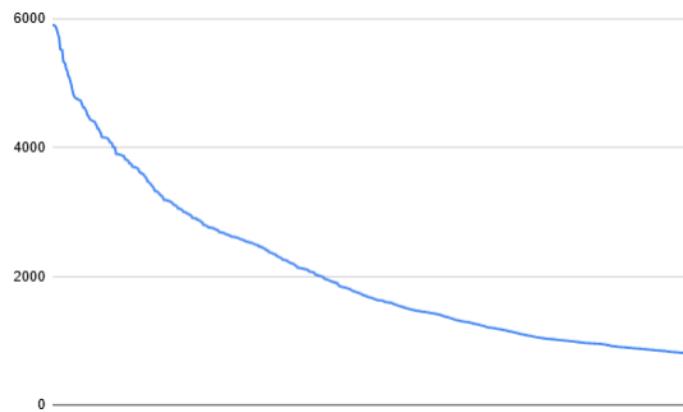
Iterasi = 626

Mendapatkan state value = 808

Waktu yang dibutuhkan = 30 menit



Gambar 5. Percobaan 3 Initial State



Gambar 6. Percobaan 3 Plot State Value

Berdasarkan hasil yang didapatkan, algoritma steepest ascent hill-climbing menghasilkan state value yang mendekati global optima, namun tetap tidak dapat mencapai global optima, di mana akan terjebak di local optima. Dengan menghasilkan state value yang memiliki range di 500an - 600an, algoritma ini akan dapat terjebak di local optima. Hal ini dikarenakan algoritma ini akan terus mencari state value terbaik yaitu dengan value yang kecil di mana tidak akan keluar dari local optima dengan melakukan pencarian di semua neighbour yang ada. Untuk algoritma ini akan cocok untuk digunakan pada persoalan yang sederhana dan tidak kompleks dengan local optima yang sedikit, karena akan mencari solusi terbaik. Selain itu,

untuk initial state sangat berpengaruh karena jika initial state sudah dekat dengan global optima, maka akan dapat menghasilkan hasil yang baik dengan waktu yang lebih cepat dibandingkan saat initial state yang sangat jauh dari global optima.

Perbandingan hasil algoritma steepest ascent hill-climbing dengan algoritma local search lainnya adalah hasilnya akan tetap terjebak di local optima dan sulit untuk mencapai global optima, namun dapat menghasilkan solusi yang baik. Namun, dapat memberikan solusi yang cukup baik jika persoalan tidak kompleks. Jika dibandingkan dengan simulated annealing dan genetic algorithm, steepest akan kurang.

Perbandingan durasi proses pencarian algoritma steepest ascent hill-climbing dengan yang lainnya adalah jauh lebih lama dibandingkan dengan stochastic hill-climbing. Namun secara keseluruhan, durasinya termasuk lama karena akan berhenti setelah mencapai local optima yang mana akan tetap mengevaluasi semua solusi yang ada dengan mengecek pencarian di semua neighbour. Jika dibandingkan dengan simulated annealing akan lebih cepat steepest dan untuk genetic algorithm akan lebih cepat juga.

Hasil akhir yang didapatkan dari ketiga percobaan adalah ditunjukkan dari tabel di bawah ini.

Tabel 1. Hasil Percobaan Algoritma

Iterasi	State Value
593	518
524	890
626	808
Standard dev = 42,5	Standard dev = 159,59
Average = 681	Average = 738,67

Berdasarkan hasil tersebut, terlihat bahwa konsistensi hasil akhir memiliki nilai standar deviasi sebesar 159,59 dengan rata-rata 738,67 untuk state value serta banyak iterasi yang dilakukan memiliki standar deviasi sebesar 42,5 dengan rata-rata 681. Hal ini

menunjukkan bahwa untuk state value menghasilkan hasil yang lumayan konsisten karena menghasilkan state value yang masih dekat dengan jumlah iterasi yang lebih konsisten lagi walaupun dengan hasil yang masih local optima. Namun, termasuk hasil akhir dengan konsisten yang akan selalu berubah di mana disebabkan algoritma ini akan selalu mencari solusi terbaik namun tetap terjebak di local optima.

- **Hill-climbing with Sideways Move**

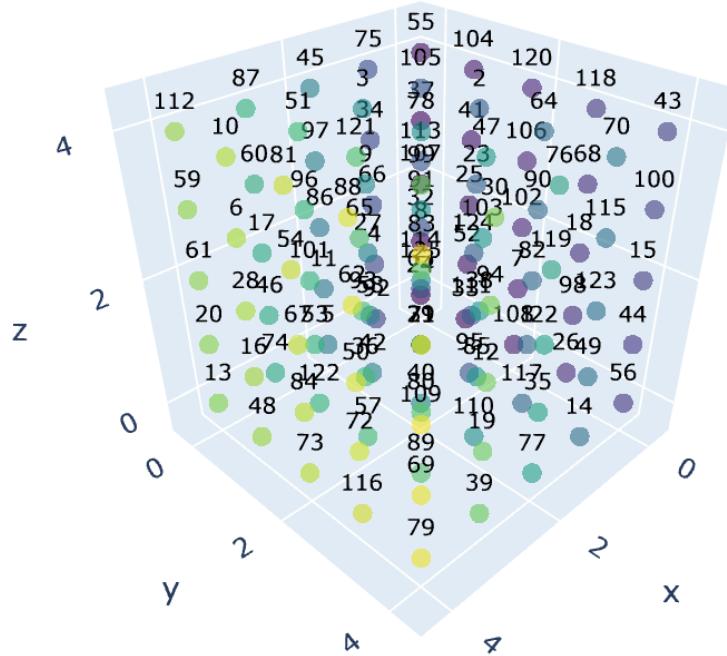
Berikut hasil eksperimen yang dilakukan untuk hill-climbing with sideways move.

Percobaan 1

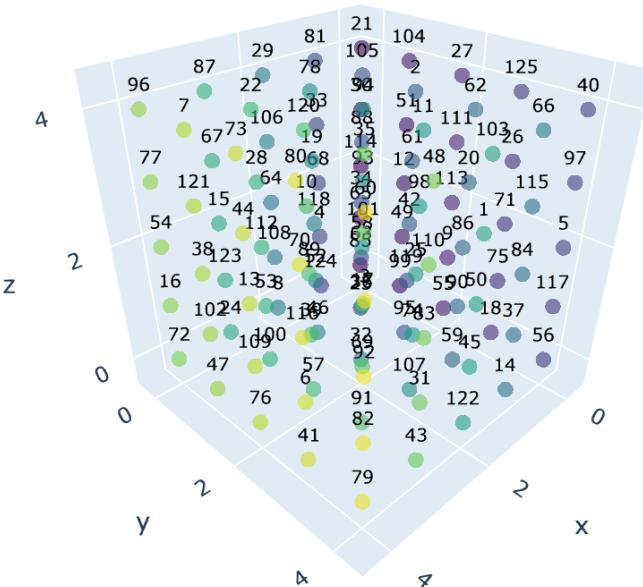
Iterasi = 132

Mendapatkan state value = 463

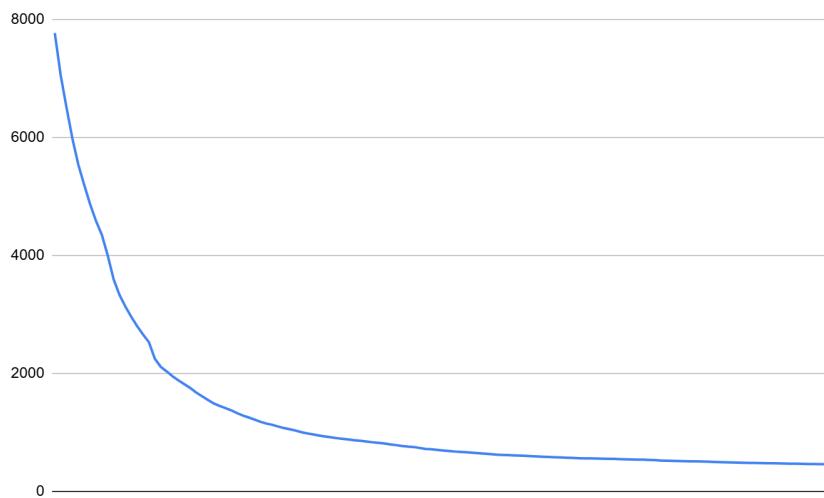
Waktu yang dibutuhkan = 12 menit



Gambar 7. Percobaan 1 Initial State



Gambar 8. Percobaan 1 Final State



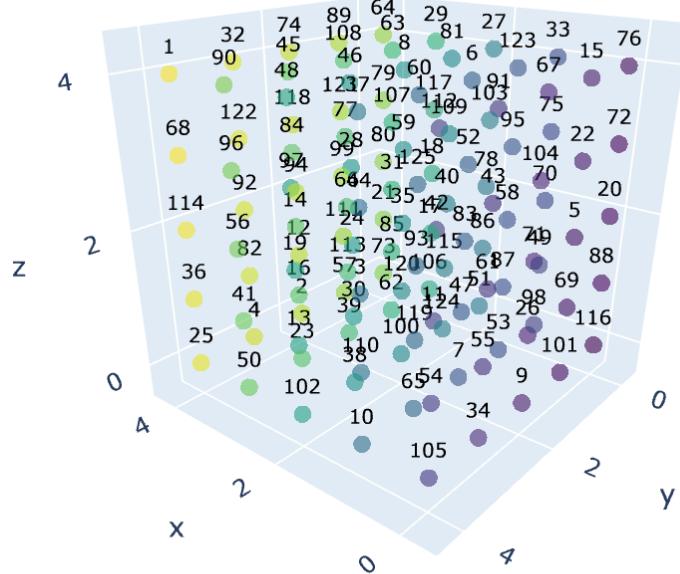
Gambar 9. Percobaan 1 Plot State Value

Percobaan 2

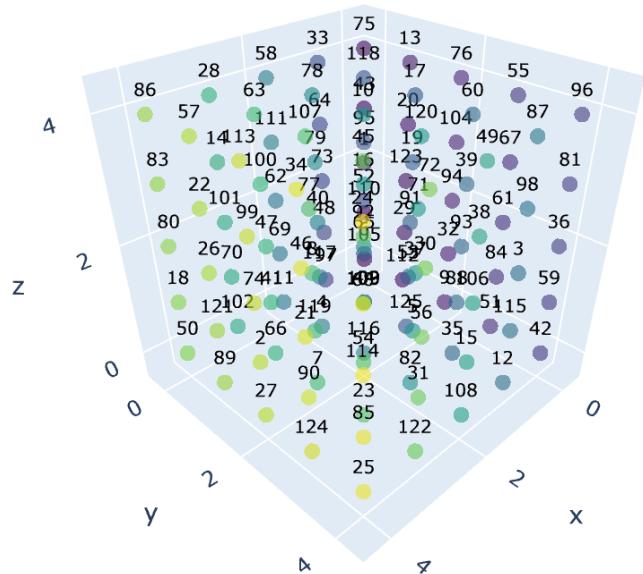
Iterasi = 106

Mendapatkan state value = 803

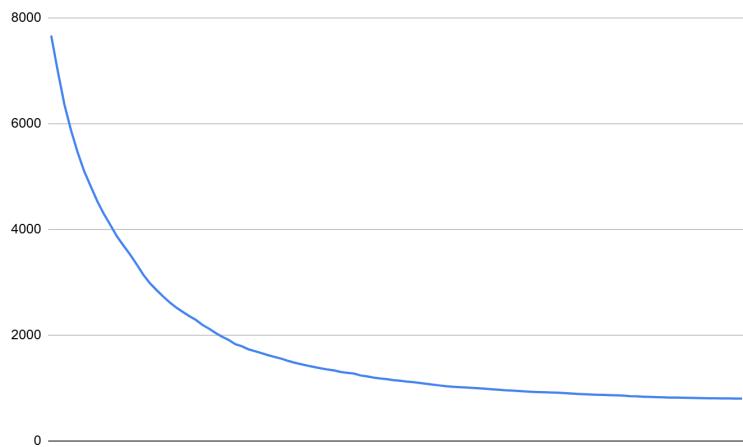
Waktu yang dibutuhkan = 26 menit



Gambar 10. Percobaan 2 Initial State



Gambar 11. Percobaan 2 Final State



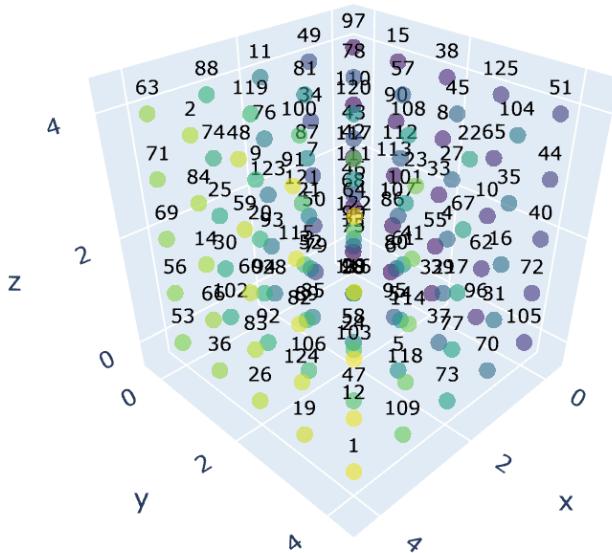
Gambar 12. Percobaan 2 Plot State Value

Percobaan 3

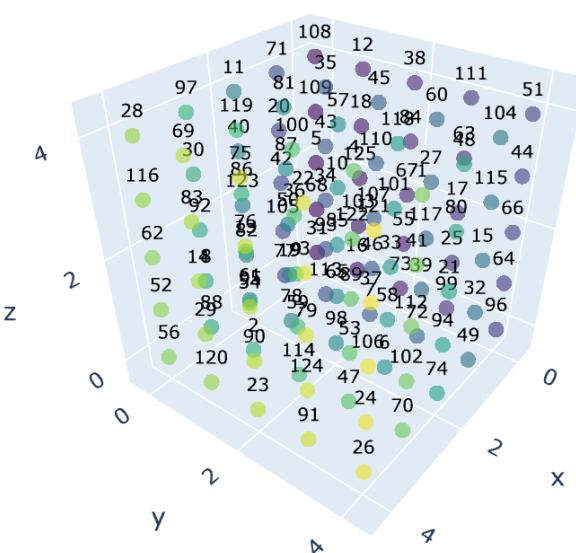
Iterasi = 110

Mendapatkan state value = 779

Waktu yang dibutuhkan = 28 menit



Gambar 13. Percobaan 3 Initial State



harus dilakukan sehingga tetap dapat terjebak di local optima walaupun masih dapat terlepas darinya.

Perbandingan algoritma hill-climbing with sideways move dengan algoritma local search yang lain adalah hasilnya akan jauh lebih baik dibandingkan hill-climbing lain yang langsung terjebak di local optima. Hill-climbing with sideways move dapat melakukan pergerakan bergeser ke samping yang dapat menghasilkan state value yang lebih baik dengan terdapat kemungkinan untuk keluar dari local optima yang ada. Selain itu, dengan terdapat jumlah sideways move yang ditentukan, dapat menghasilkan hasil state value yang mungkin yang masih lebih jauh dari global optima namun telah keluar dari local optima yang ada. Dan juga untuk algoritma ini juga terbatas dengan persoalan yang tidak kompleks. Jika dibandingkan dengan simulated annealing dan genetic algorithm, akan kurang karena tidak fleksibel dan hanya bergantung pada pergeseran ke kiri.

Perbandingan durasi proses pencarian algoritma hill-climbing sideways move dengan algoritma lain adalah memiliki waktu yang lebih lama dibandingkan steepest ascent hill-climbing dan stochastic hill-climbing karena memiliki pergerakan ke samping sehingga dapat keluar dari local optima. Namun lebih cepat dibandingkan dengan random restart hill-climbing karena hanya bergantung di pergerakan ke samping saja dan bukan untuk melakukan restart setiap bertemu dengan local optima. Jika dibandingkan dengan simulated annealing akan sama lamanya dan jika dengan genetic algorithm akan lebih cepat.

Dari hasil yang didapatkan dapat dilihat di tabel berikut ini.

Tabel 2. Hasil Percobaan Algoritma

Iterasi	State Value	Waktu (menit)
132	463	12
106	803	26
110	779	28
Standard dev = 14	Standard dev = 189,75	Standard dev = 8,72

Average = 116	Average = 681,67	Average = 22
---------------	------------------	--------------

Berdasarkan hasil yang didapatkan, terlihat bahwa nilai state value yang memiliki standar deviasi yang cukup tinggi dibandingkan algoritma lain dengan iterasi yang standar deviasinya kecil serta waktu dalam menit yang memiliki standar deviasi yang kecil juga. Hal ini menunjukkan bahwa algoritma memiliki konsistensi yang kurang di mana akan bergantung kepada jumlah pergerakan ke samping yang dilakukan serta tergantung dengan initial state yang diberikan di awal. Selain itu, dihasilkan konsistensi yang kurang diakibatkan walaupun dengan pergerakan ke samping, tetapi dapat terjebak di local optima. Berdasarkan dari iterasi dan waktu, terlihat bahwa masih konsisten dengan tetap ada pengaruh dari initial state.

- **Random Restart Hill-climbing**

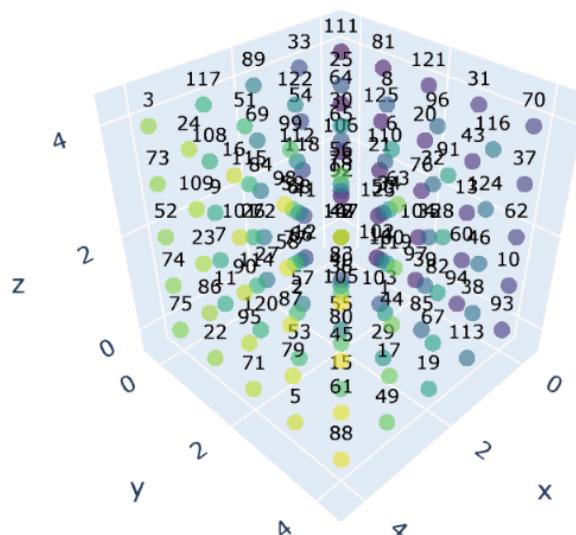
Berikut hasil eksperimen yang dilakukan untuk random restart hill-climbing.

Percobaan 1

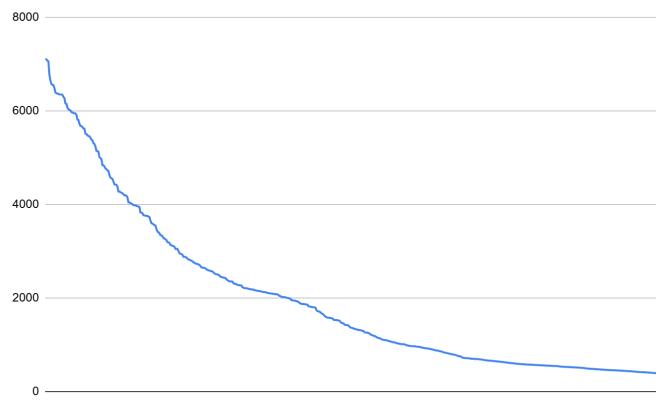
Iterasi = 617 iterasi sampai nilai state valuenya 388

Mendapatkan state value = terakhir di 388

Waktu yang dibutuhkan = 46 mnt



Gambar 16. Percobaan 1 Initial State



Gambar 17. Percobaan 1 Plot State Value

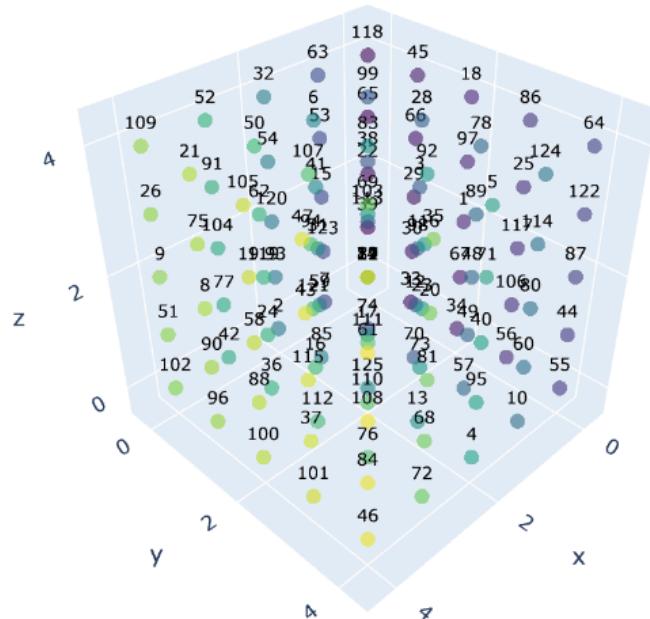
Percobaan 2

Iterasi = 608

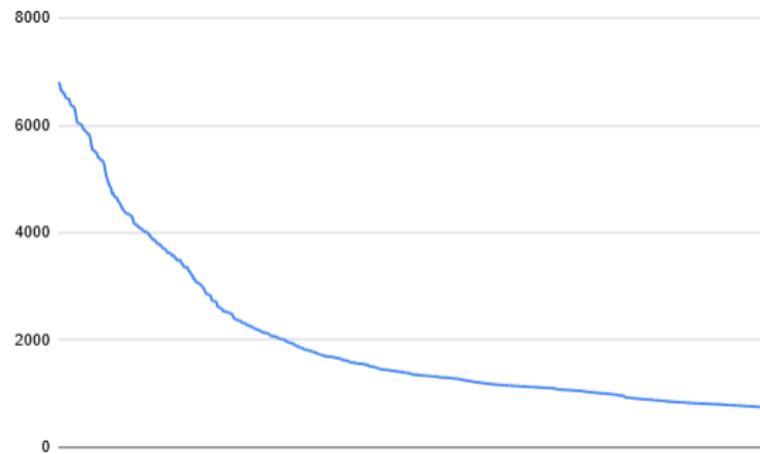
Mendapatkan state value = 748

Waktu yang dibutuhkan = 40 menit

State Awal



Gambar 18. Percobaan 2 Initial State



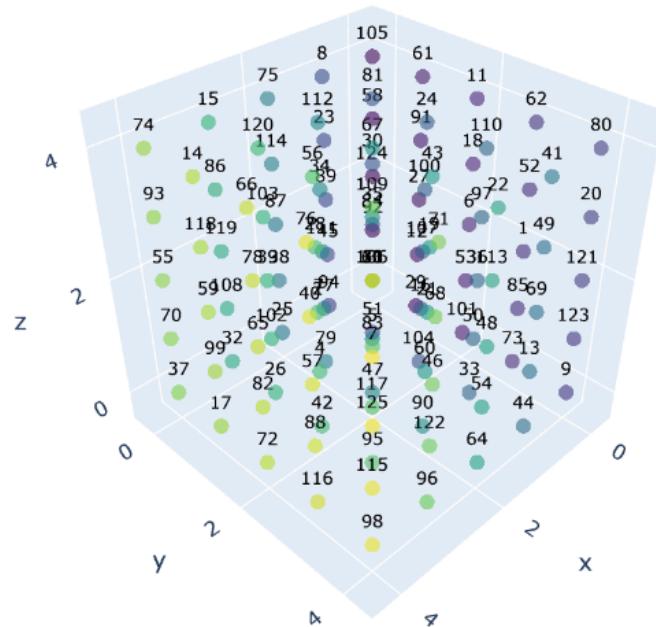
Gambar 19. Percobaan 1 Plot State Value

Percobaan 3

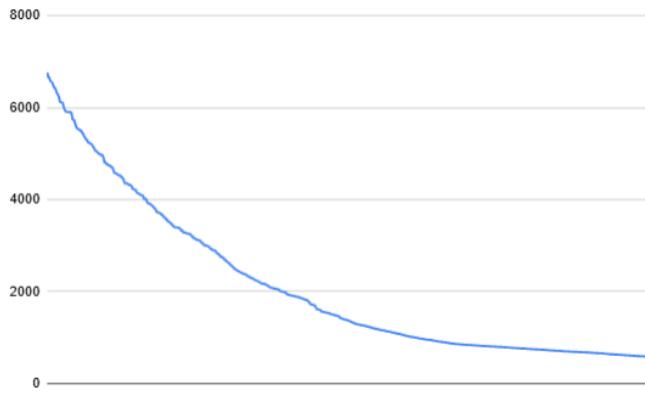
Iterasi = 625

Mendapatkan state value = 579

Waktu yang dibutuhkan = 48 menit



Gambar 20. Percobaan 3 Initial State



Gambar 21. Percobaan 3 Plot State Value

Berdasarkan hasil yang didapatkan, algoritma random restart hill-climbing memiliki hasil yang dapat cukup dekat dengan global optima. Hal ini disebabkan oleh algoritma ini akan melakukan restart saat ditemukan local optima, sehingga akan terus mengubah hingga mendapatkan hasil yang mendekati global optima. Namun walaupun dapat keluar dari local optima, tetap akan tergantung apakah akan mencapai sedekat dengan global optima karena random yang dilakukan tidak dapat ditentukan. Selain itu, terbatas karena jika restart sudah tercapai semua maka tidak dapat mencari solusi yang mendekati global optima lagi. Random restart hill-climbing menghasilkan state value terendah yaitu 388 sehingga mendapatkan algoritma ini dapat mencapai global optima. Random restart hill-climbing akan bergantung kepada jumlah restart yang ditentukan sehingga jika semakin banyak maka akan dapat ditemukan state value yang dekat dengan global optima yang akan menghindari local optima.

Perbandingan algoritma random restart hill-climbing dengan algoritma local search yang lain hasilnya adalah akan menghasilkan state value yang mendekati global optima karena dapat melakukan restart setiap terjebak di local optima. Dibandingkan dengan hill-climbing lainnya, jauh lebih mendekati state valuenya ke global optima sehingga hasilnya lebih baik, kecuali jika dibandingkan dengan stochastic karena setiap pencarian akan diulang dari posisi yang random sehingga belum tentu terjadi penurunan state valuenya. Namun tetap untuk persoalan yang kompleks, tetap tidak efektif karena akan melakukan restart yang sangat banyak jika menemukan

local optima yang banyak. Untuk simulated annealing akan memiliki hasil yang serupa karena memiliki restart dan untuk genetic algorithm akan kurang karena tidak adanya seleksi populasi.

Perbandingan durasi proses pencarian algoritma random restart hill-climbing dengan algoritma lainnya adalah menghasilkan waktu yang lebih lama dibandingkan dengan hill climbing lain karena akan menghindari local optima dengan melakukan restart sehingga akan memberikan waktu yang lama. Dengan durasi yang lama ini, sesuai dengan hasil state value yang diberikan karena dapat keluar dari local optima. Jika dengan simulated annealing akan lebih lama dan jika dengan genetic algorithm akan lebih cepat random restart hill-climbing.

Dari hasil yang didapatkan dapat dilihat di tabel berikut ini.

Tabel 3. Hasil Percobaan Algoritma

Iterasi	State Value	Waktu (menit)
617	388	46
608	748	40
625	579	48
Standard deviation = 8,5	Standard deviation = 180,11	Standard deviation = 4,16
Average = 616,67	Average = 571,67	Average = 44,67

Berdasarkan hasil tersebut, terlihat bahwa konsistensi algoritma random restart hill-climbing menghasilkan hasil yang kurang konsisten di state valuenya karena menghasilkan range yang cukup tinggi. Hal ini disebabkan oleh adanya kegiatan random untuk keluar dari local optima yang cukup banyak sehingga kurang konsisten.

- **Stochastic Hill-climbing**

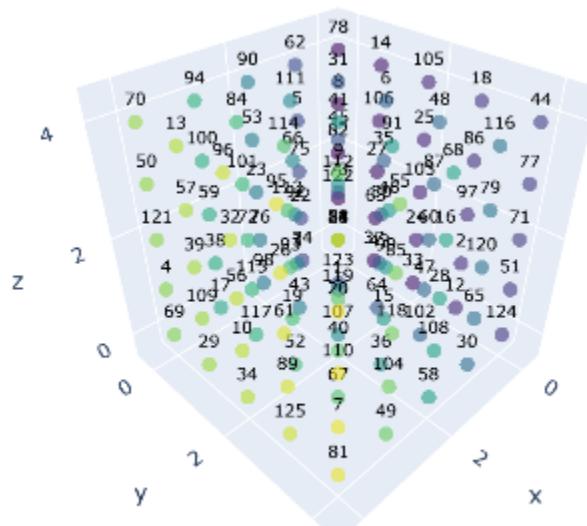
Berikut hasil eksperimen yang dilakukan untuk hill-climbing with sideways move.

Percobaan 1

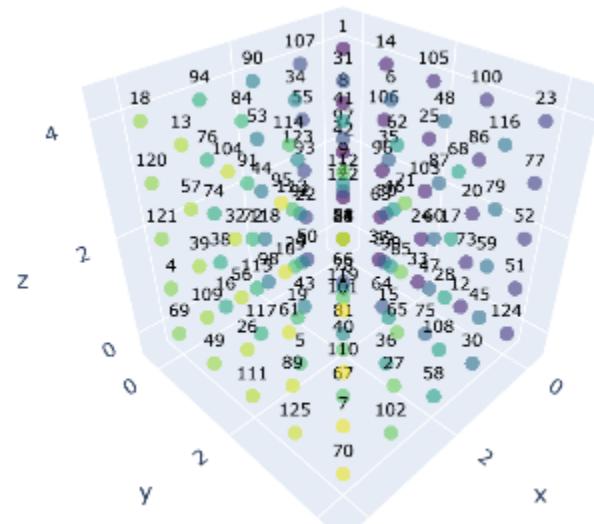
Iterasi = 1000

Mendapatkan state value = 4919

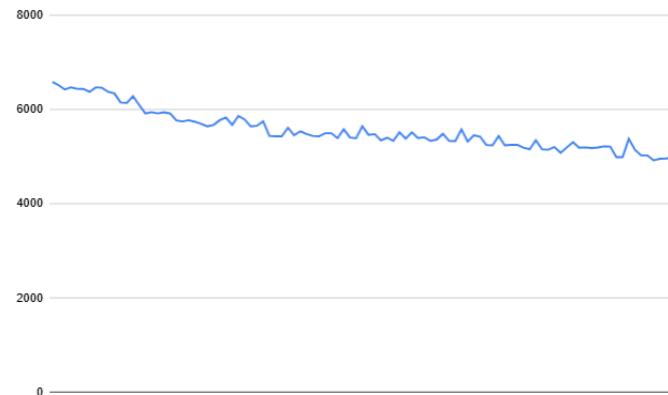
Waktu yang dibutuhkan = 1 s



Gambar 22. Percobaan 1 Initial State



Gambar 23. Percobaan 1 Final State



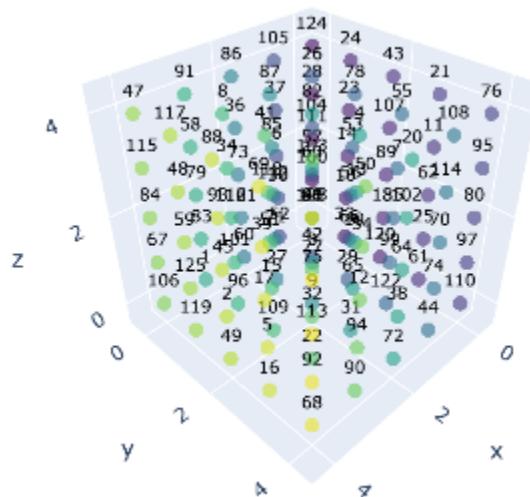
Gambar 24. Percobaan 1 Plot State Value

Percobaan 2

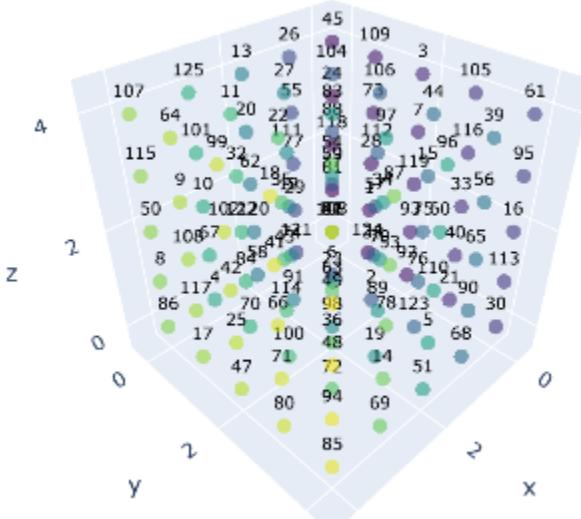
Iterasi = 10000

Mendapatkan state value = 832

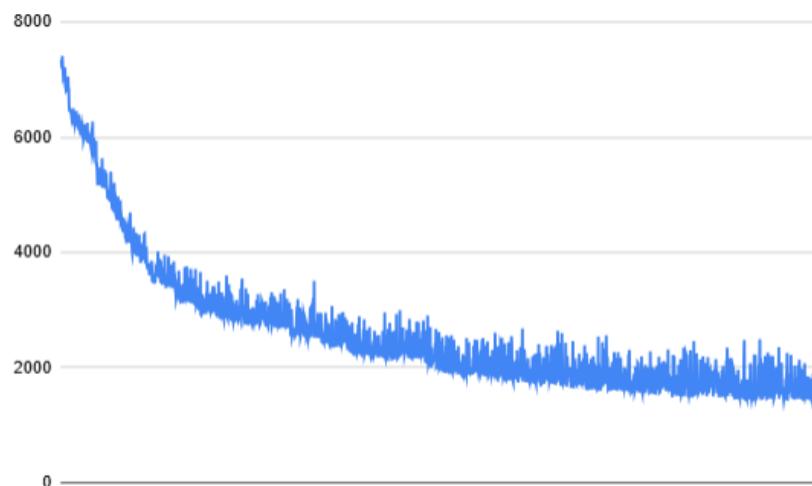
Waktu yang dibutuhkan = 9 s



Gambar 25. Percobaan 2 Initial State



Gambar 26. Percobaan 2 Final State



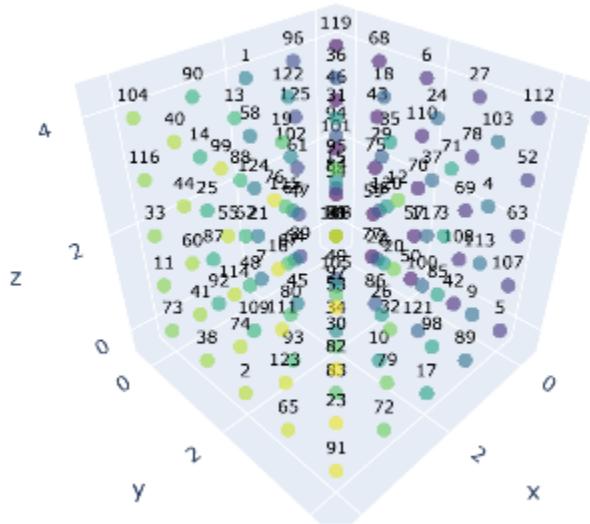
Gambar 27. Percobaan 2 Plot State Value

Percobaan 3

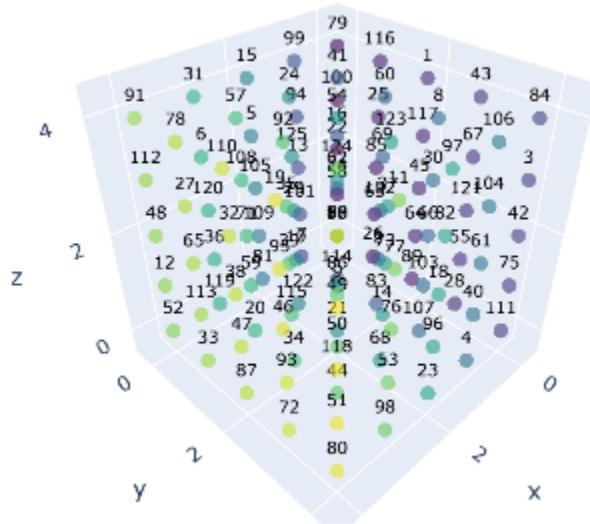
Iterasi = 15000

Mendapatkan state value = 788

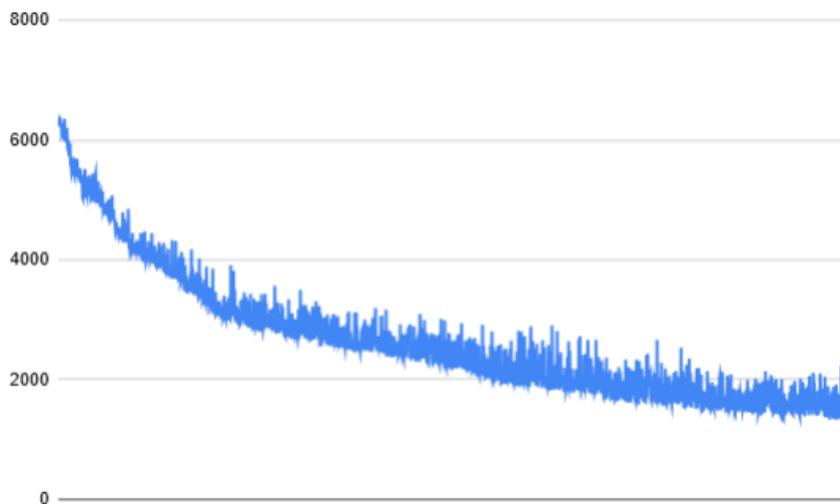
Waktu yang dibutuhkan = 15 s



Gambar 28. Percobaan 3 Initial State



Gambar 29. Percobaan 3 Final State



Gambar 30. Percobaan 3 Plot State Value

Berdasarkan hasil yang didapatkan, untuk kedekatan algoritma dengan global optima terlihat bahwa untuk percobaan 1 masih jauh dari optima yang terlihat bahwa state value sebesar 4919 dalam waktu 1 s, untuk percobaan 2 yang dilakukan peningkatan jumlah iterasi menghasilkan state value yang cukup berkurang menjadi sebesar 832 dalam waktu 9 s, yang mana dengan iterasi yang bertambah menghasilkan algoritma dapat melakukan eksekusi menghasilkan banyak pilihan solusi dan mulai mendekati optima, dan untuk percobaan 3 dengan iterasi yang meningkat juga menghasilkan state value yang mendekati optima yaitu 788 dalam waktu 15 s di mana lebih baik dari 2 percobaan sebelumnya. Dilakukan pemberian iterasi karena algoritma membutuhkan waktu yang sangat lama untuk menghasilkan final state dari magic cubenya, sehingga dilakukan pemberian nilai iterasi yang diinginkan untuk mempercepat algoritma dengan melihat perbedaan antara state value yang dihasilkan dan waktu untuk algoritma berjalan. Dengan dilakukan perubahan nilai iterasi, terlihat bahwa semakin banyak iterasi yang dilakukan, maka akan membuat nilai state value yang semakin baik juga. Namun, algoritma ini akan terjebak di local optima yang mana tidak dapat berpindah kembali. Stochastic hill-climbing ini dapat mengeksplor persoalan yang kompleks karena dapat melakukannya secara cepat dengan tidak perlu evaluasi neighbour yang ada.

Jika dibandingkan dengan algoritma local search lain, kekurangannya adalah harus memiliki jumlah iterasi yang sangat banyak untuk mendapatkan hasil state value yang rendah. Selain itu, karena terjebak di local optima jika dibandingkan dengan yang lain kurang optimal sehingga nilai state valuenya masih cenderung tinggi. Hasilnya dapat dilihat di plot di mana akan terus menurun grafiknya karena mencari state value yang rendah. Untuk simulated annealing dan genetic algorithm, hasil stochastic akan kurang optimal.

Jika dibandingkan dengan algoritma lainnya, durasi pencarinya memiliki waktu yang paling singkat, di mana dengan menggunakan iterasi yang cukup tinggi seperti contohnya adalah 15000 iterasi menghasilkan waktu 14 s. Hal ini didukung bahwa untuk algoritma ini akan melakukan pertukaran dengan *neighbour* yang lebih baik saja hingga menghasilkan state value terbaik, namun tetap mudah terjebak di local optima. Hal ini dikarenakan pencarian neighbour dilakukan secara acak. Jika dibandingkan dengan simulated annealing dan genetic algorithm akan menghasilkan waktu yang lebih cepat untuk stochastic hill-climbing.

Untuk mengecek konsistensinya, akan dilakukan percobaan untuk iterasi sebanyak 15000 beberapa kali dengan hasilnya sebagai berikut.

Tabel 4. Hasil Percobaan Algoritma

Iterasi	Waktu
815	14
788	15
753	14
1234	14
760	14
922	14
672	14
941	14
753	14

677	14
Standard dev = 167,03	Standard dev = 0,32
Average = 831,5	Average = 14,1

Berdasarkan hasil tersebut, terlihat algoritma dihasilkan cukup konsisten di mana untuk iterasi dihasilkan standar deviasi 167,03 dengan rata-rata 831,5 dan untuk waktu dihasilkan standar deviasi 0,32 dengan rata-rata 14,1. Untuk iterasi, terlihat bahwa menghasilkan jumlah iterasi yang cukup konsisten walaupun terjadi fluktuasi dan untuk durasi sangat konsisten di mana standar deviasi yang kecil. Terjadi fluktuasi pada iterasi dikarenakan terjadinya proses pertukaran yang dapat menyebabkan perbedaan langkah yang dipilih selanjutnya hingga mendapatkan solusi yang ada. Sedangkan untuk waktu walaupun jumlah iterasi yang dilakukan berfluktuasi, waktu hampir selalu stabil.

- **Simulated Annealing**

Berikut hasil eksperimen yang dilakukan untuk simulated annealing.

Percobaan 1

Temperatur awal : 10000

Cooling rate : 0.97

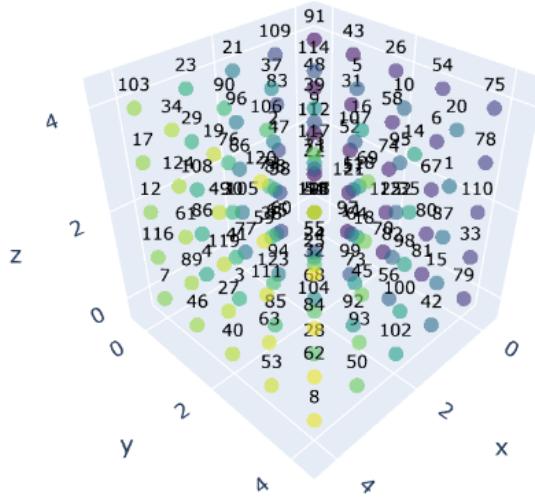
Jumlah iterasi : 100000

Threshold : 7×10^{-300}

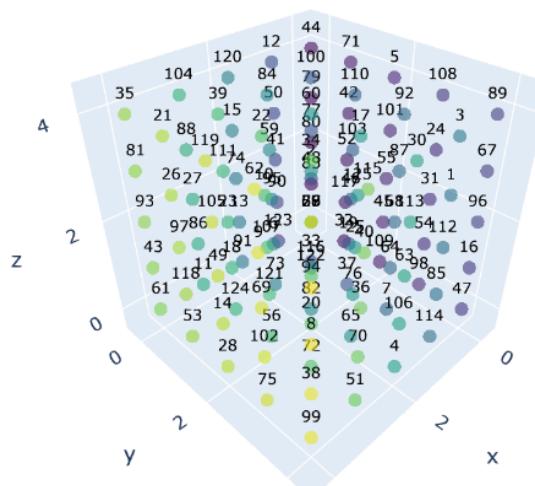
Durasi proses pencarian : 109.212 s

Nilai Objective Function akhir yang dicapai : 428

Frekuensi escape di local optima : 183

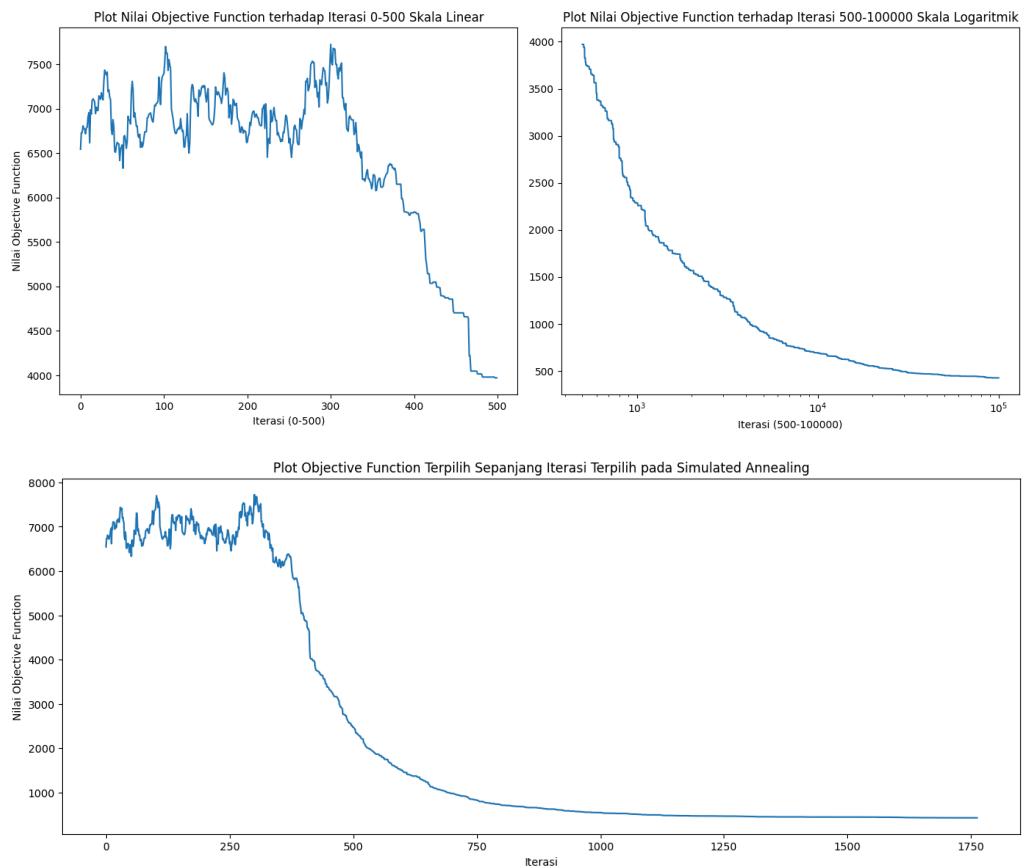


Gambar 31. Initial State



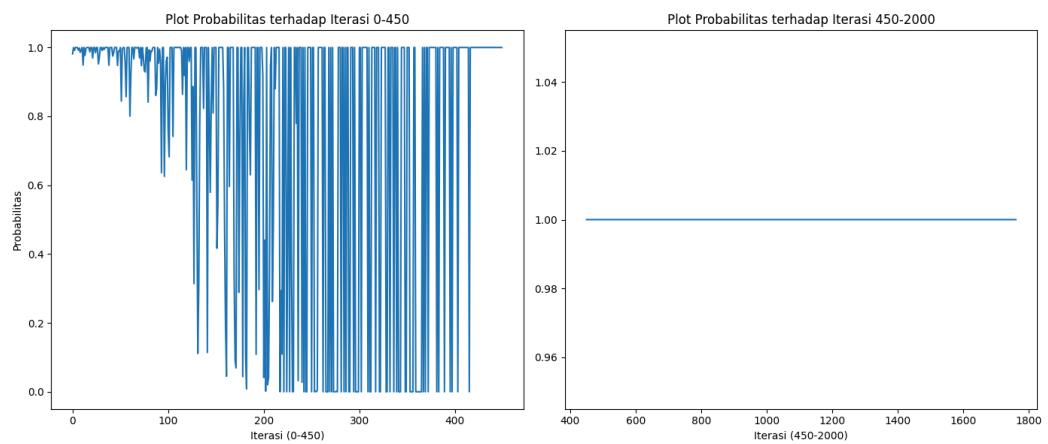
Gambar 32. Final State

Plot nilai objective function terhadap banyak iterasi yang telah dilewati:



Gambar 33. Plot Objective Function

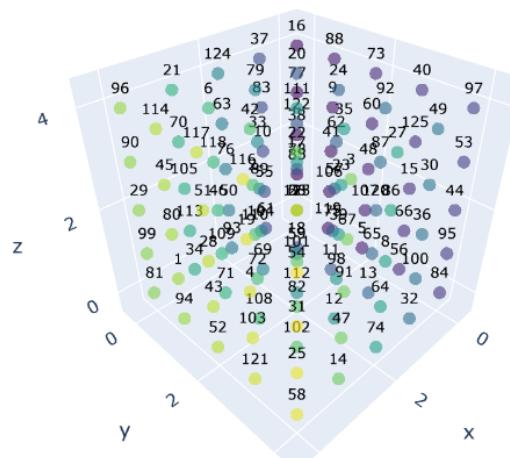
Plot nilai probabilitas $P = e^{\Delta E/T}$ terhadap banyak iterasi yang telah dilewati



Gambar 34. Plot Probabilitas

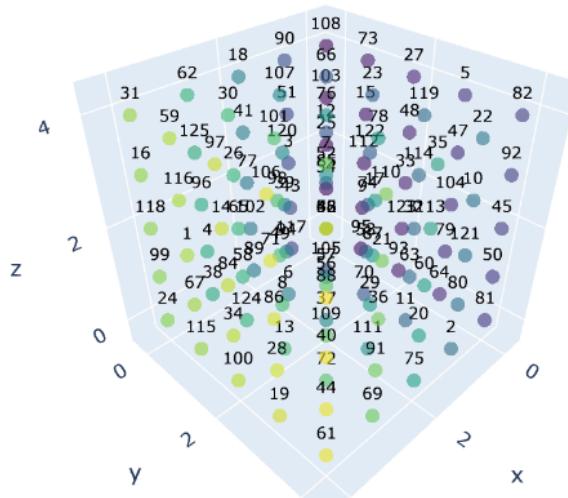
Percobaan 2

Temperatur awal : 1000
Cooling rate : 0.97
Jumlah iterasi : 1000
Durasi proses pencarian : 1.188 s
Nilai Objective Function akhir yang dicapai : 2392
Frekuensi escape dari local optima : 151
State awal kubus:



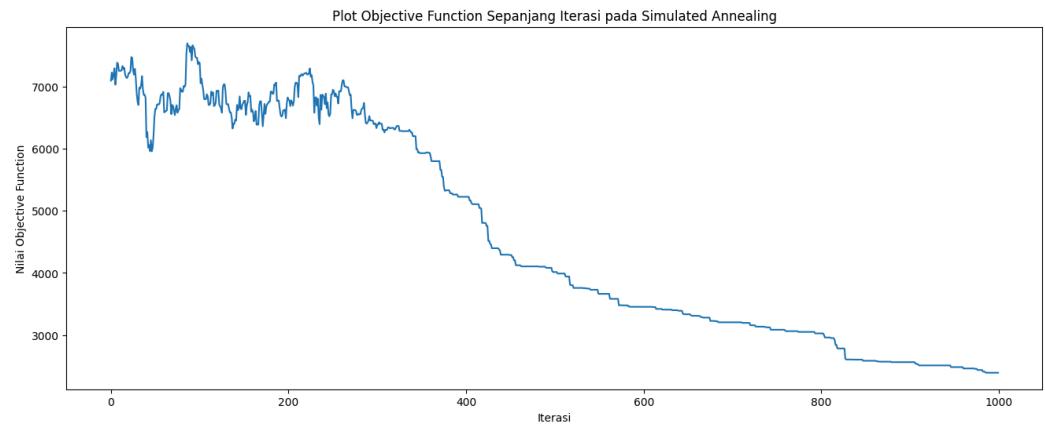
Gambar 35. Initial State

State akhir kubus:



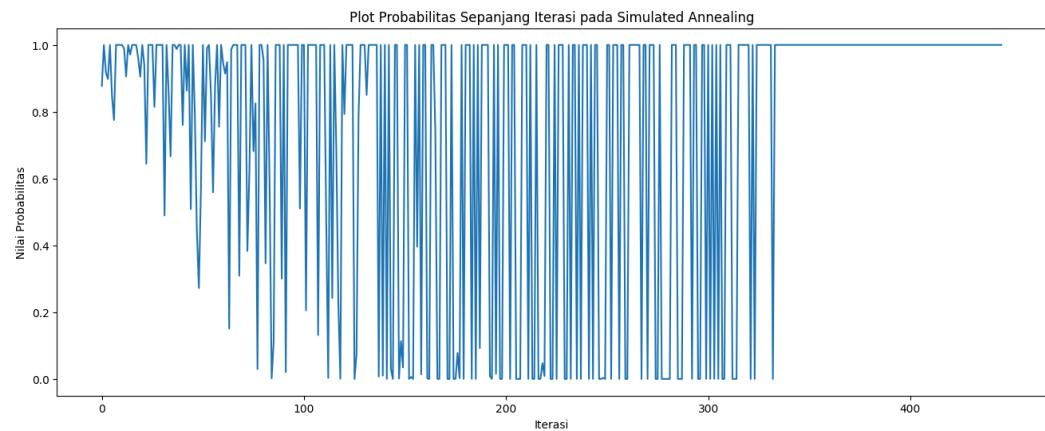
Gambar 36. Final State

Plot nilai objective function terhadap banyak iterasi yang telah dilewati:



Gambar 37. Plot Objective Function

Plot nilai probabilitas $P = e^{\Delta E/T}$ terhadap banyak iterasi yang telah dilewati:

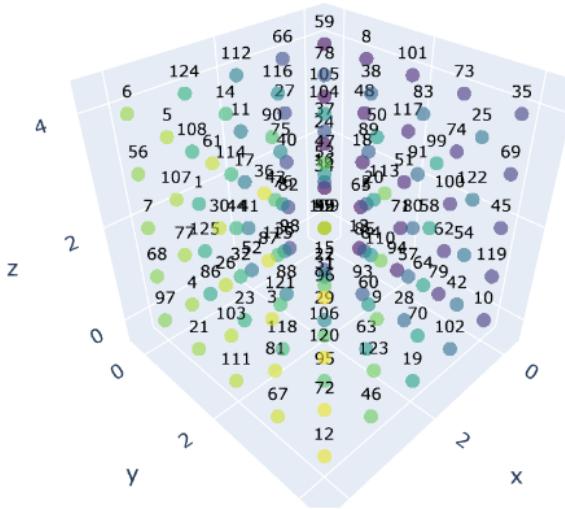


Gambar 38. Plot Probabilitas

Percobaan 3

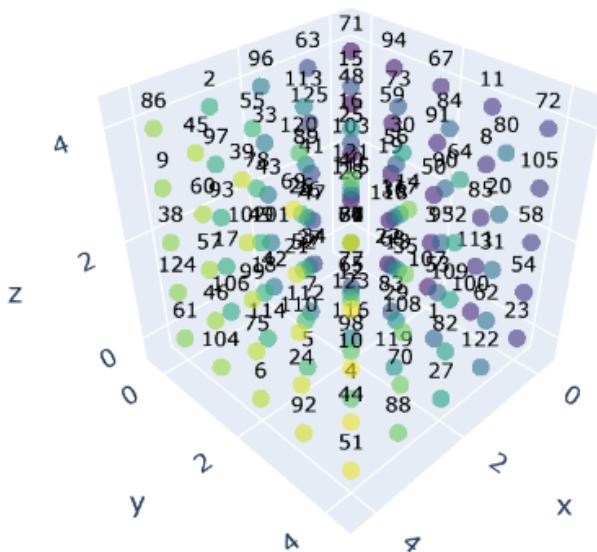
Temperatur awal	: 10000
Cooling rate	: 0.95
Jumlah iterasi	: 20000
Durasi proses pencarian	: 22.455 s
Nilai Objective Function akhir yang dicapai	: 446
Frekuensi escape di local optima	: 114

State awal kubus:



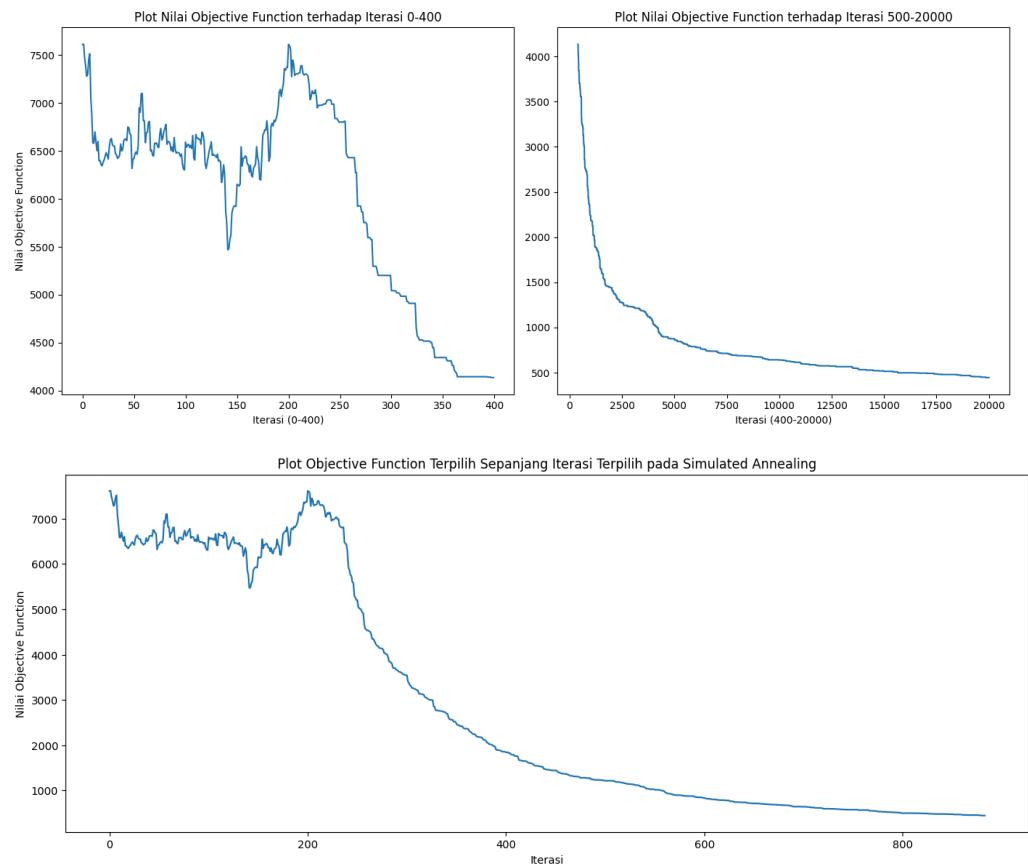
Gambar 39. Initial State

State akhir kubus:



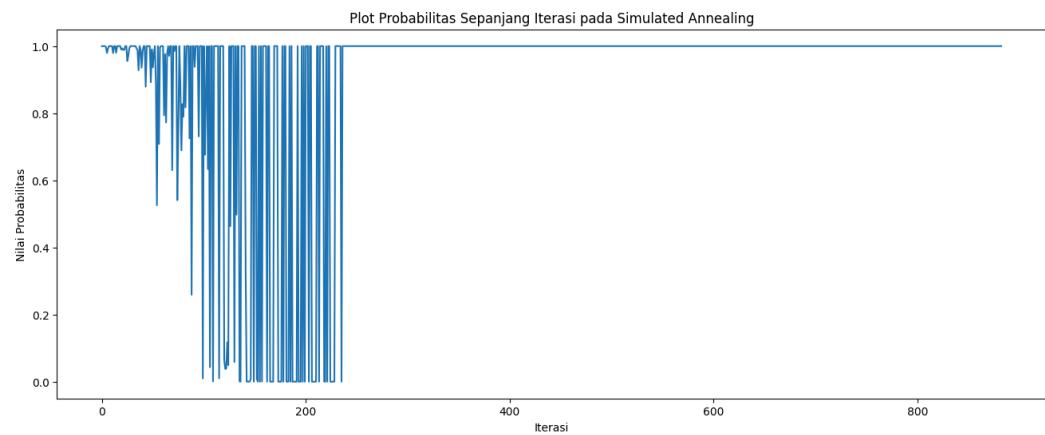
Gambar 40. Final State

Plot nilai objective function terhadap banyak iterasi yang telah dilewati:



Gambar 41. Plot Objective Function

Plot nilai probabilitas $P = e^{\Delta E/T}$ terhadap banyak iterasi yang telah dilewati:



Gambar 42. Plot Probabilitas

Pembahasan

Terdapat beberapa variabel yang dapat disesuaikan pada algoritma Simulated Annealing dan berkorelasi pada hasil yang didapat. Temperatur awal dan cooling rate berpengaruh pada probabilitas yang berdasar pada Metropolis acceptance criterion: $P = e^{\Delta E/T}$, yang berimplikasi pada banyaknya eksplorasi (melalui penerimaan bad moves). Dapat dilihat melalui banyaknya frekuensi escape dari local optima, dengan temperatur awal yang sama pada percobaan 1 dan 3 tetapi cooling rate yang berbeda, frekuensi escape yang didapat berbeda jauh. Begitu pula percobaan 1 dan 2 yang memiliki cooling rate yang sama, namun frekuensi escape dari local optima berbeda akibat dari temperatur awal yang berbeda.

Durasi pencarian berkorelasi positif dengan banyaknya iterasi, karena pada algoritma Simulated Annealing kompleksitas algoritmanya hanya bergantung pada jumlah iterasinya: $O(\text{iterations})$. Banyaknya iterasi juga berpengaruh terhadap seberapa baik hasil yang didapatkan. Dengan kompleksitas dan waktu yang lebih baik daripada algoritma Hill Climbing dan Genetic Algorithm, algoritma Simulated Annealing dapat mendapat hasil yang lebih baik dalam waktu yang sama.

Pada algoritma yang dibuat, Nilai ambang (threshold) didapat dari batas bawah rentang nilai $\Delta E/T$ yaitu -700 . e^{-700} bernilai $9.86e-305$. Diambil nilai threshold sebesar $7e-300$ karena dirasa cukup mengakomodasi bad moves.

Hasil akhir yang paling mendekati global optima adalah percobaan 1. Hasil ini berkorelasi positif dengan temperatur awal yang tinggi serta cooling rate yang lambat, serta banyaknya iterasi sehingga memungkinkan hasil yang lebih baik.

- **Genetic Algorithm**

- Jumlah Populasi sebagai Kontrol**

- Jumlah Populasi = 500

- a. Variasi Iterasi 1

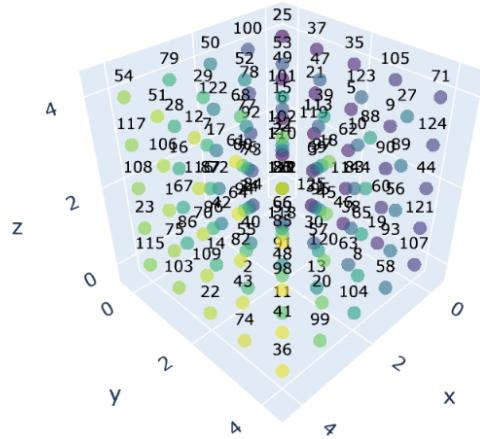
- Jumlah Iterasi = 100

Percobaan 1

Waktu yang dibutuhkan = 93.045912 second(s)

State Awal

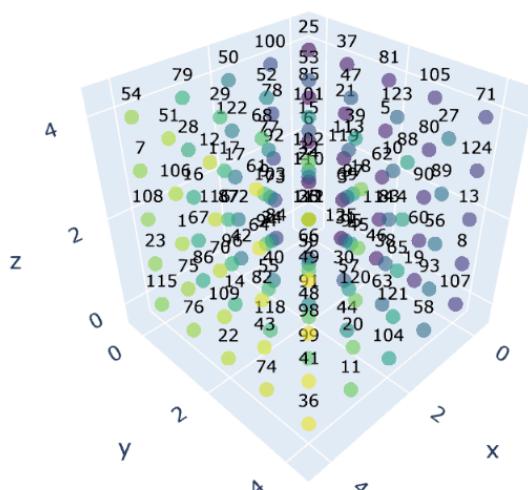
best of initialize iteration from 500 population(s) with value: 5624



Gambar 43. Initial State

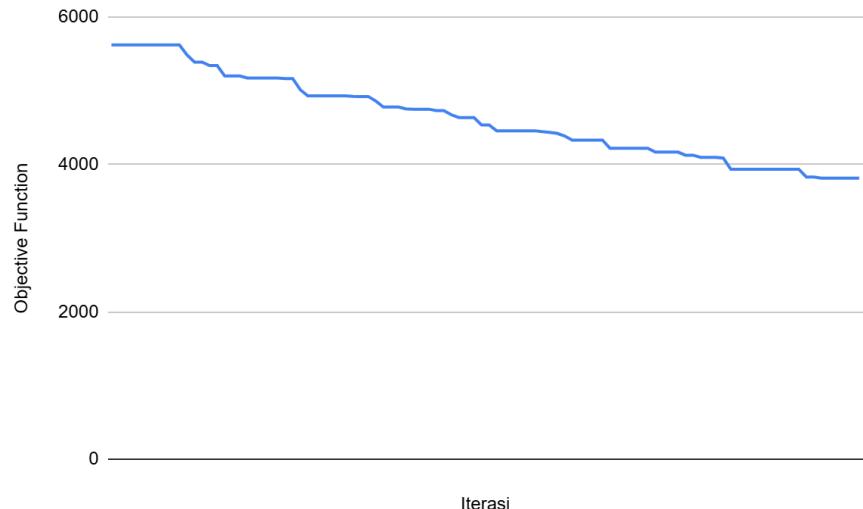
State Akhir

best last iteration with value: 3815



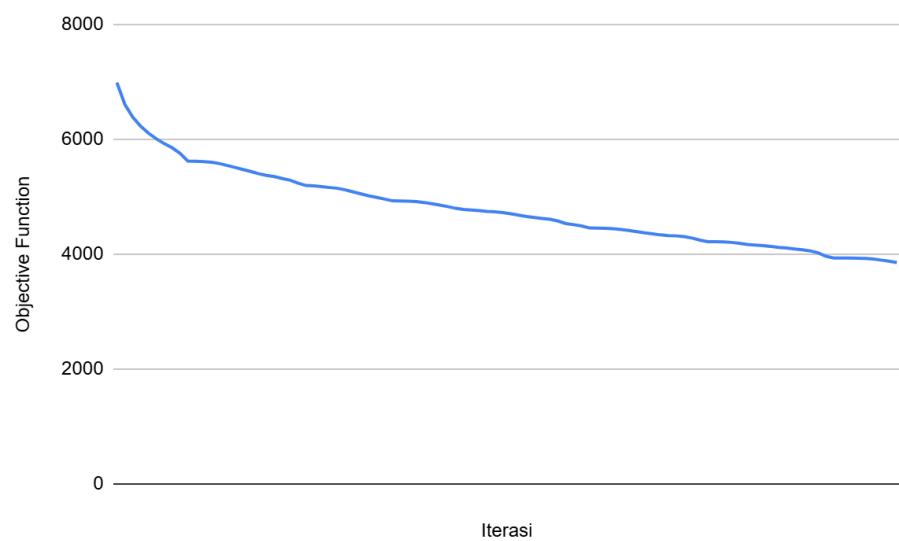
Gambar 44. Final State

Plot Objective Function Maximum



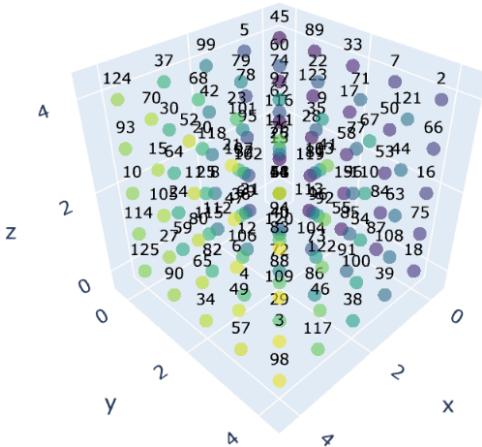
Gambar 45. Plot Objective Function Max

Plot Objective Function Average



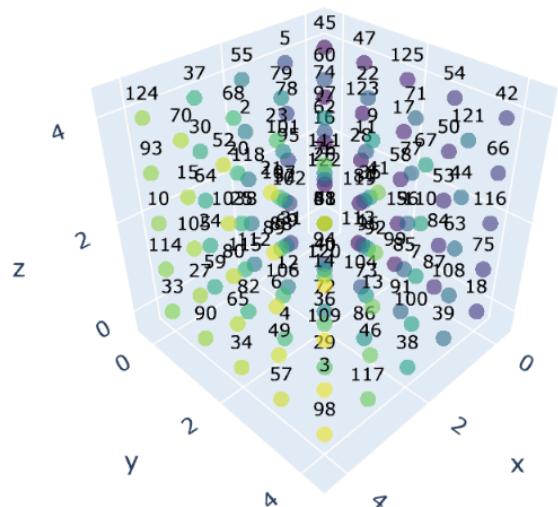
Gambar 46. Plot Objective Function Ag
Percobaan 2
Waktu yang dibutuhkan = 97.239923 second(s)
State Awal

best of initialize iteration from 500 population(s) with value: 5686



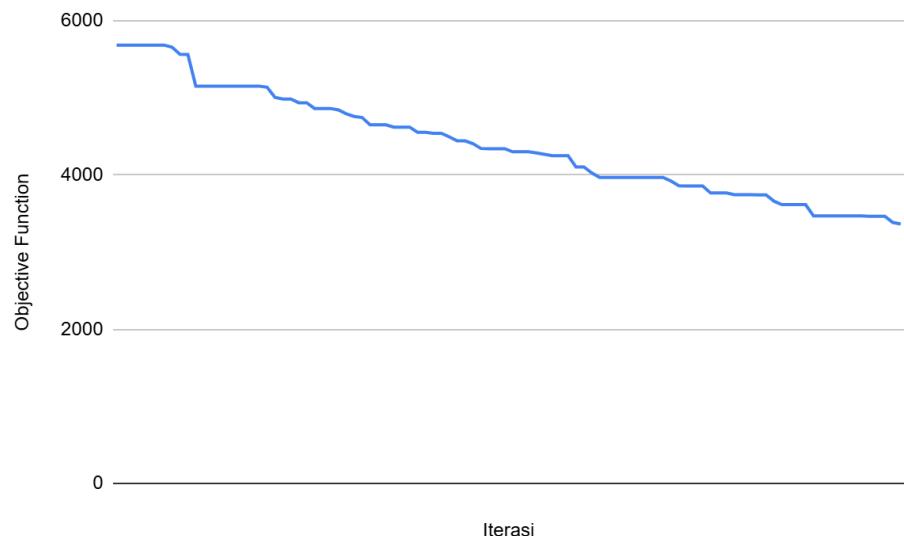
Gambar 47. Initial State
State Akhir

best last iteration with value: 3365

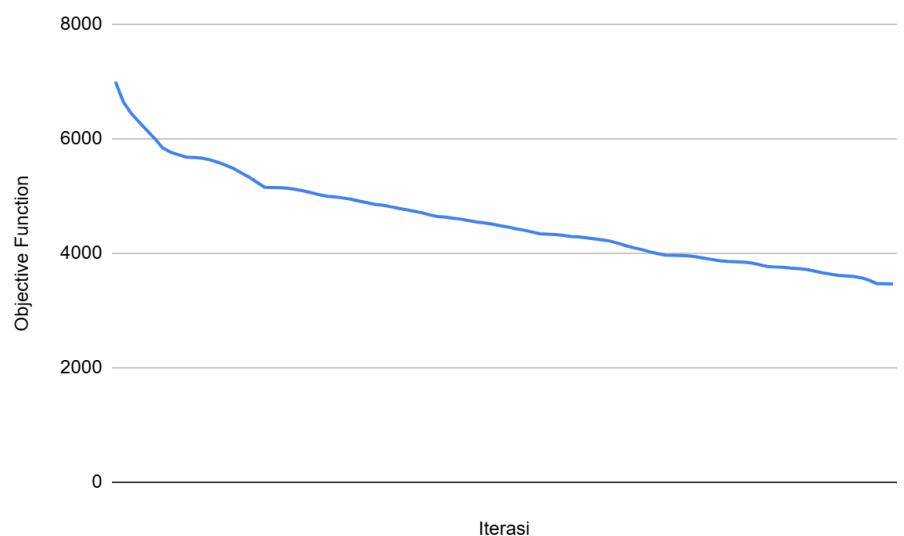


Gambar 48. Final State

Plot Objective Function Maximum



Gambar 49. Plot Objective Function Maximum
Plot Objective Function Average



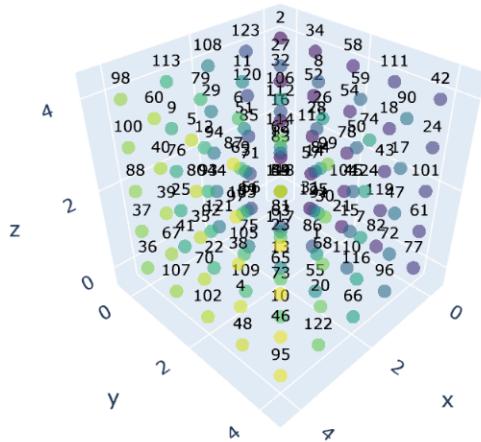
Gambar 50. Plot Objective Function Average

Percobaan 3

Waktu yang dibutuhkan = 93.144186 second(s)Z

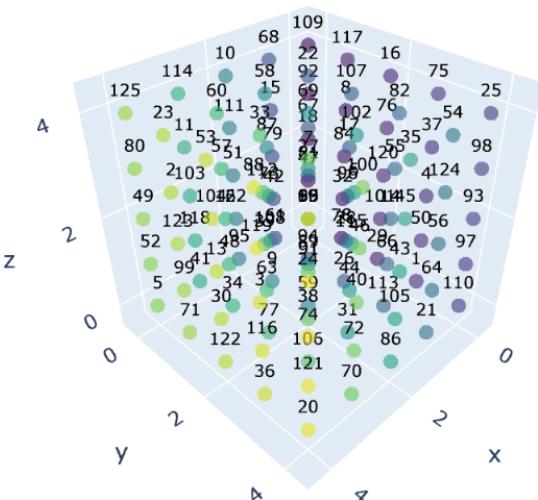
State Awal

best of initialize iteration from 500 population(s) with value: 5821



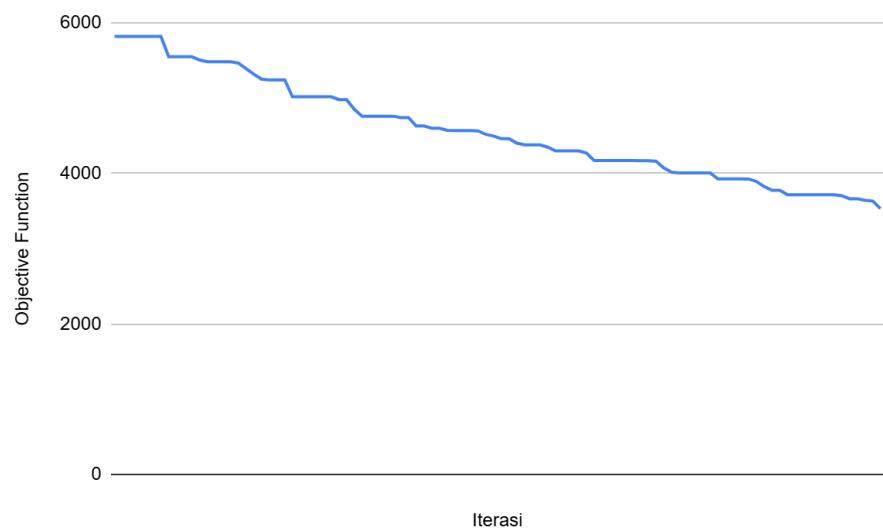
Gambar 51. Initial State
State Akhir

best last iteration with value: 3533

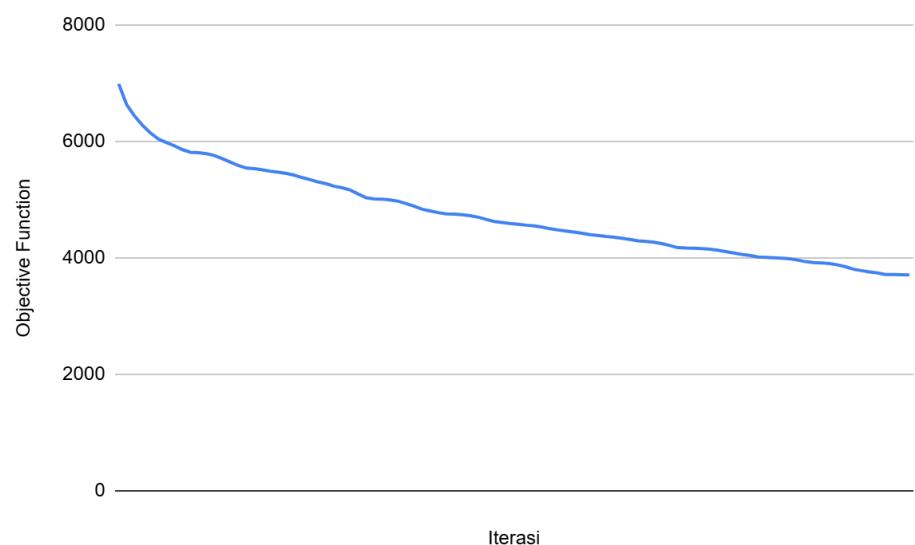


Gambar 52. Final State

Plot Objective Function Maximum



Gambar 53. Plot Objective Function Maximum
Plot Objective Function Average

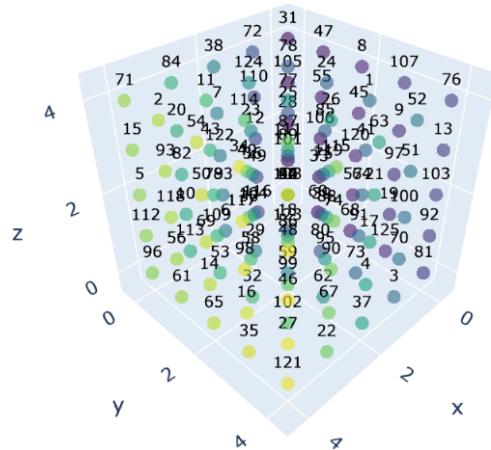


Gambar 54. Plot Objective Function Average

- b. Variasi Iterasi 2
Jumlah Iterasi = 500

Percobaan 1
Waktu yang dibutuhkan = 484.023242 second(s)
State Awal

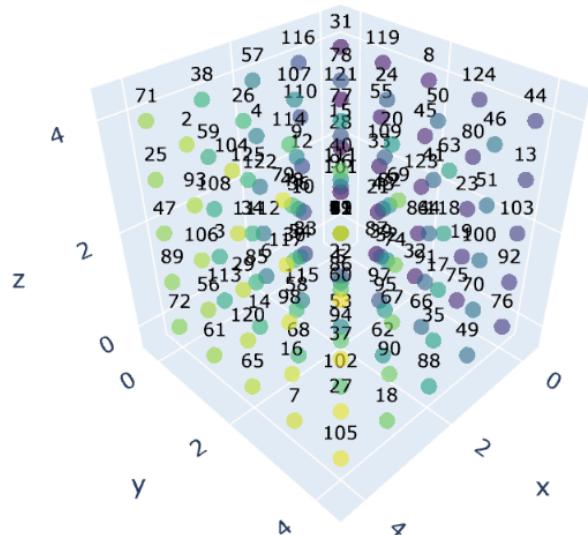
best of initialize iteration from 500 population(s) with value: 5777



Gambar 55. Initial State

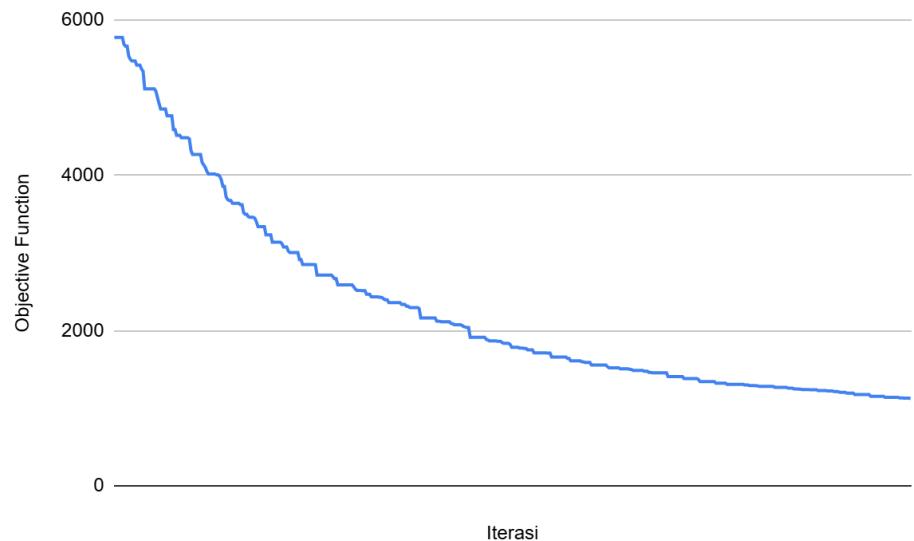
State Akhir

best last iteration with value: 1124



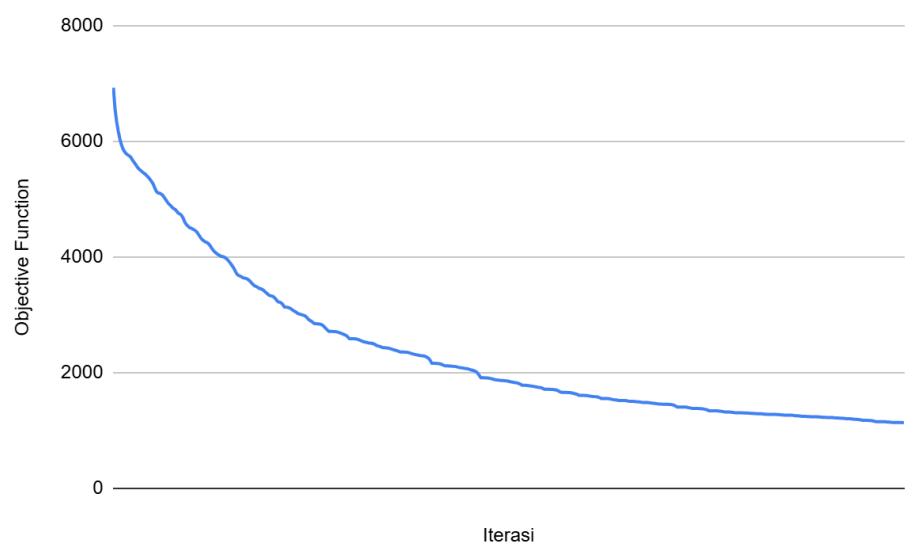
Gambar 56. Final State

Plot Objective Function Maximum



Gambar 57. Plot Objective Function Maximum

Plot Objective Function Average



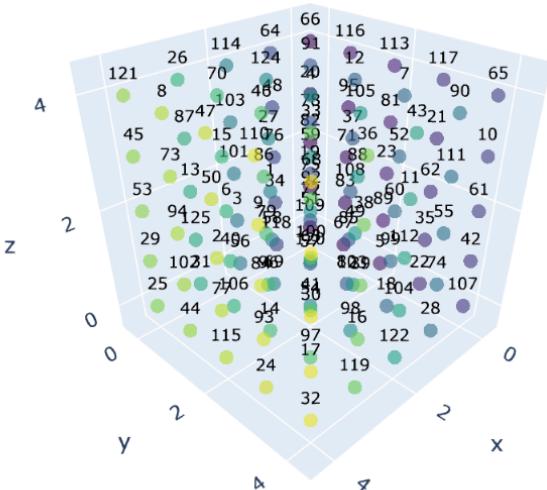
Gambar 58. Plot Objective Function Average

Percobaan 2

Waktu yang dibutuhkan =326.314432 second(s)

State Awal

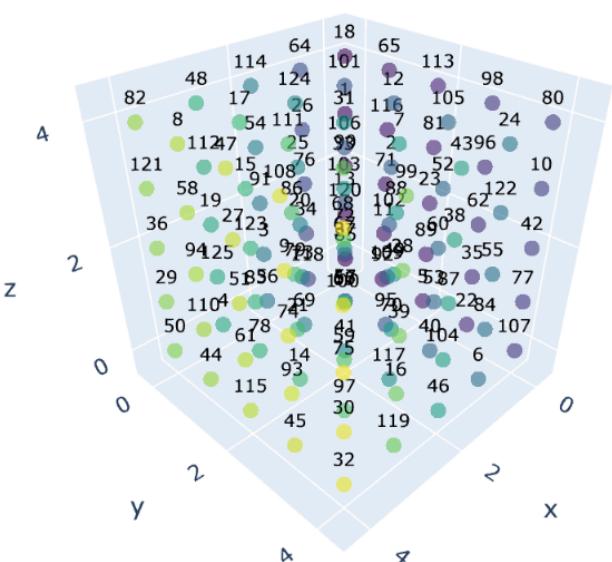
best of initialize iteration from 500 population(s) with value: 5486



Gambar 59. Initial State

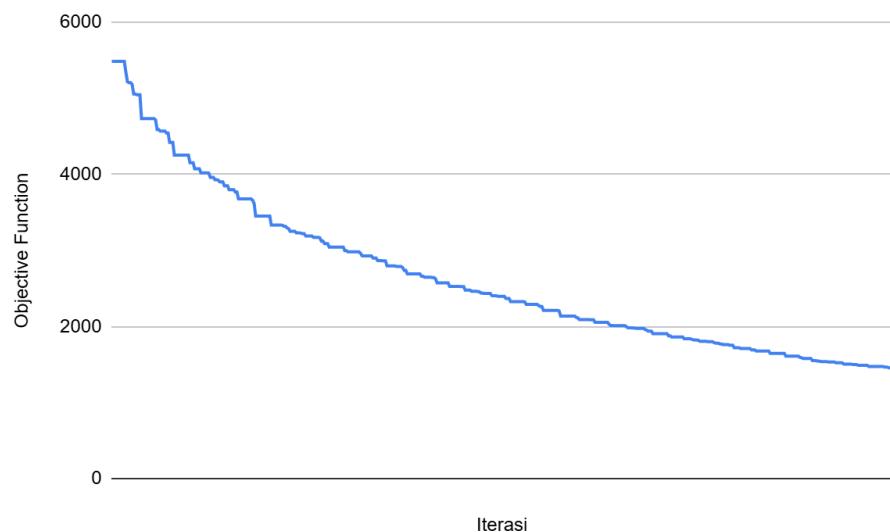
State Akhir

best last iteration with value: 1456



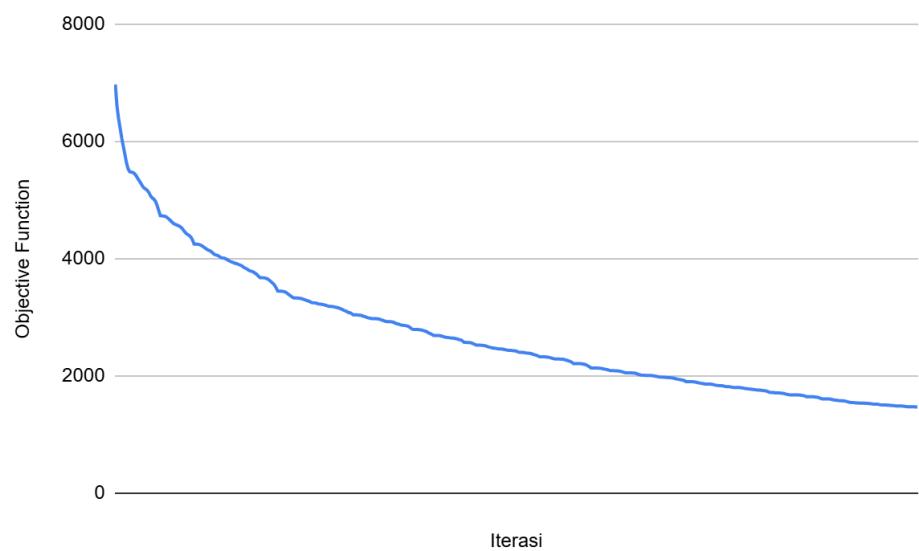
Gambar 60. Final State

Plot Objective Function Maximum



Gambar 61. Plot Objective Function Maximum

Plot Objective Function Average



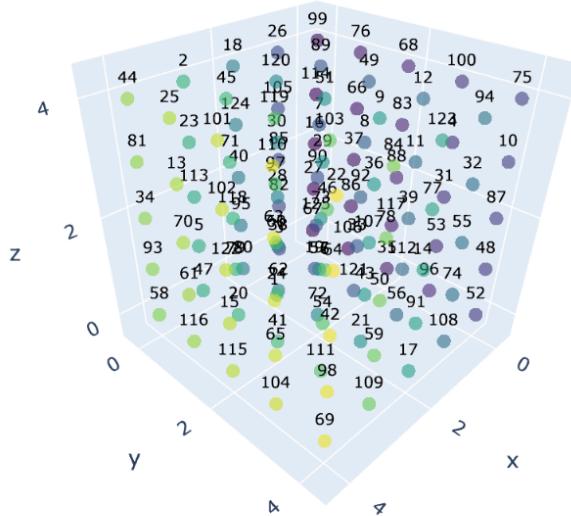
Gambar 62. Plot Objective Function Average

Percobaan 3

Waktu yang dibutuhkan = 477.959154 second (s)

State Awal

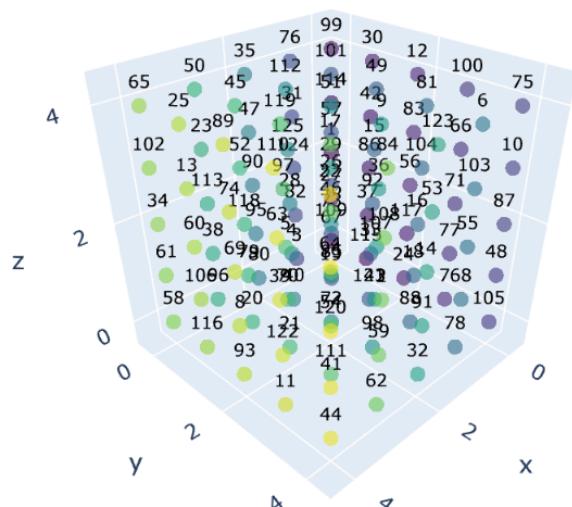
best of initialize iteration from 500 population(s) with value: 5433



Gambar 63. Initial State

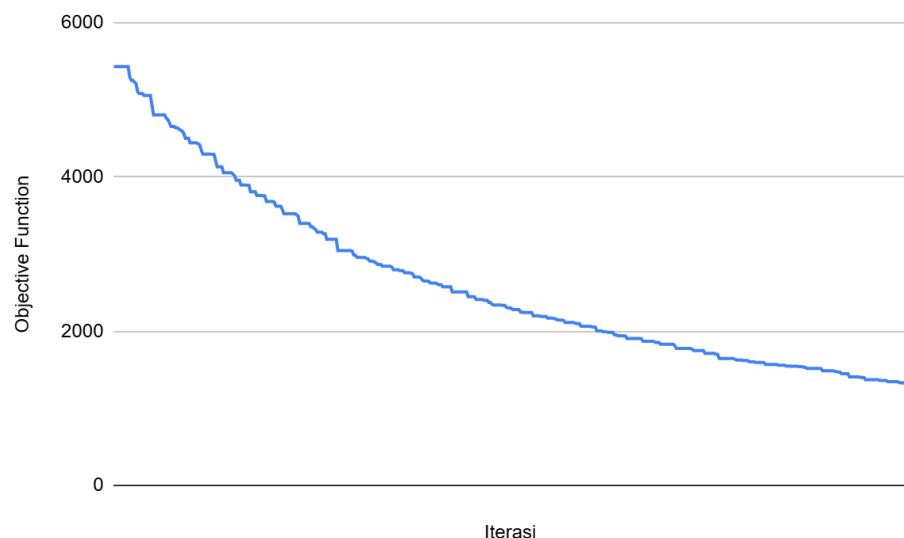
State Akhir

best last iteration with value: 1319



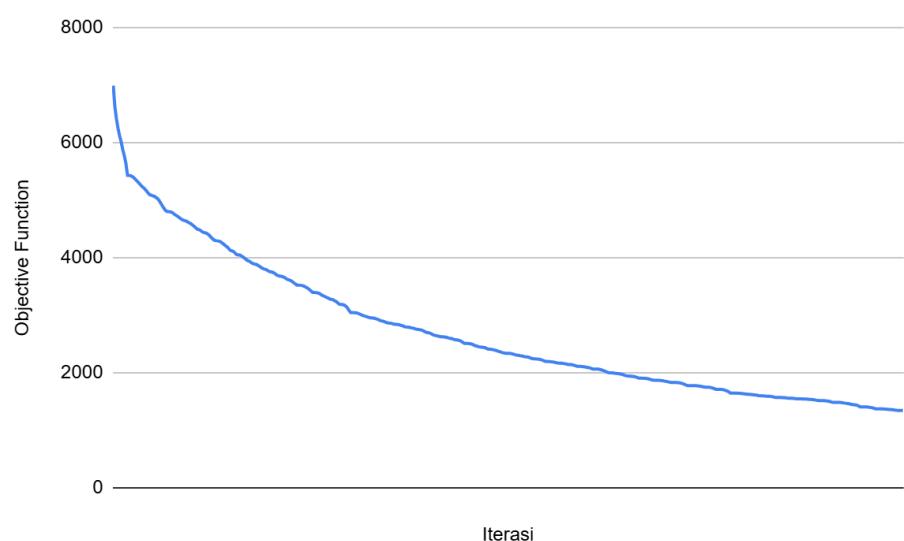
Gambar 64. Final State

Plot Objective Function Maximum



Gambar 65. Plot Objective Function Maximum

Plot Objective Function Average



Gambar 66. Plot Objective Function Average

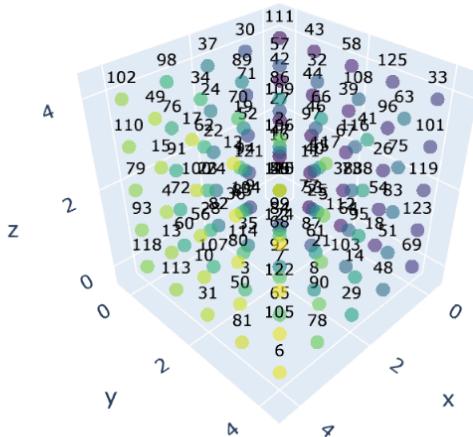
- c. Variasi Iterasi 3
Jumlah Iterasi = 1000

Percobaan 1

Waktu yang dibutuhkan = 644.900125 second(s)

State Awal

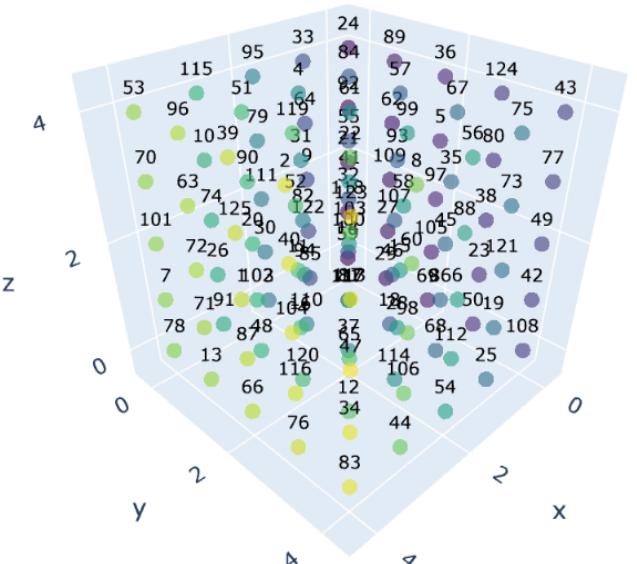
best of initialize iteration from 500 population(s) with value: 5692



Gambar 67. Initial State

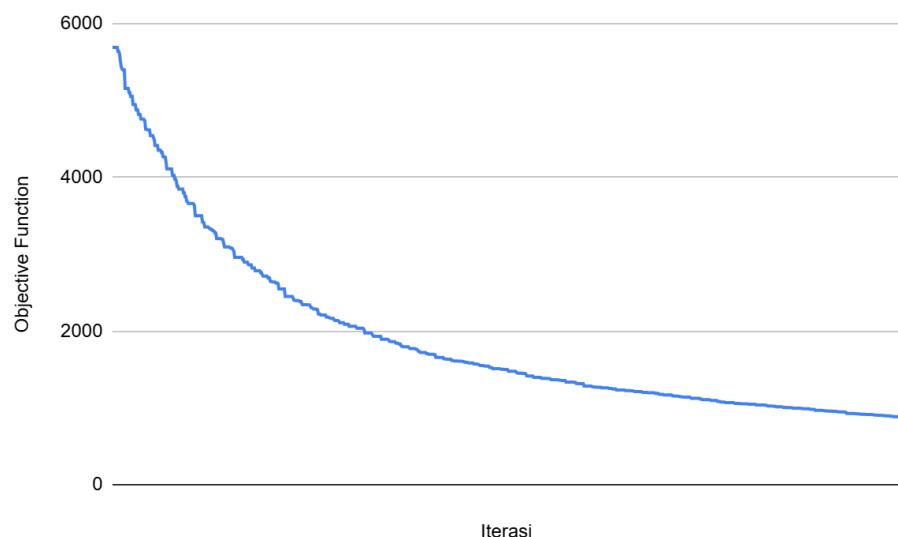
State Akhir

best last iteration with value: 883



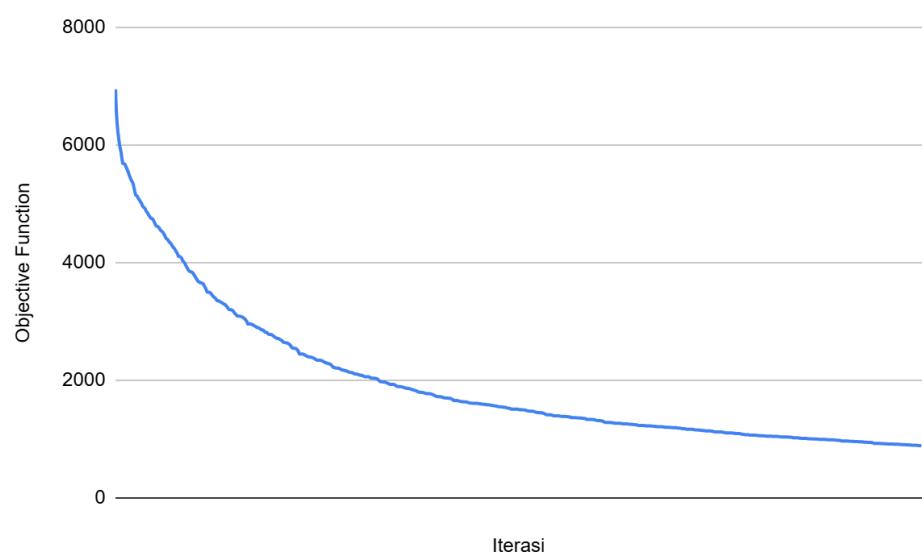
Gambar 68. Final State

Plot Objective Function Maximum



Gambar 69. Plot Objective Function Maximum

Plot Objective Function Average



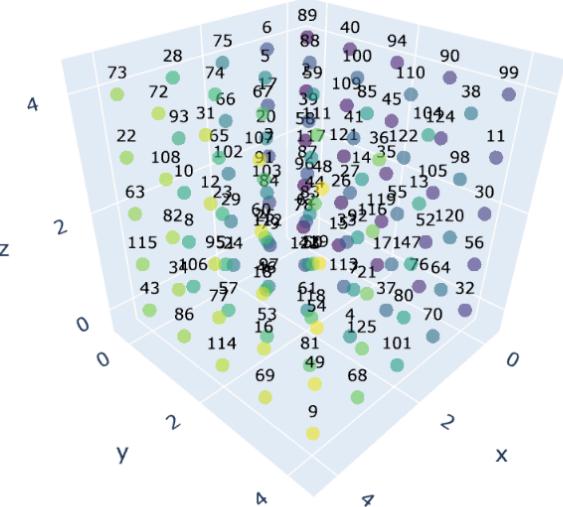
Gambar 70. Plot Objective Function Average

Percobaan 2

Waktu yang dibutuhkan = 1012.240776 second(s)

State Awal

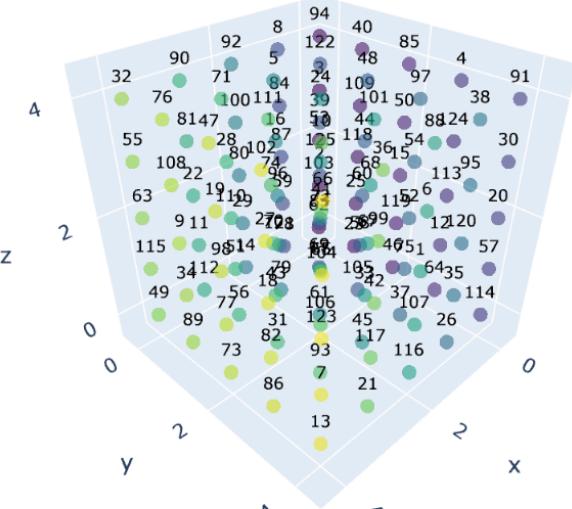
best of initialize iteration from 500 population(s) with value: 5497



Gambar 71. Initial State

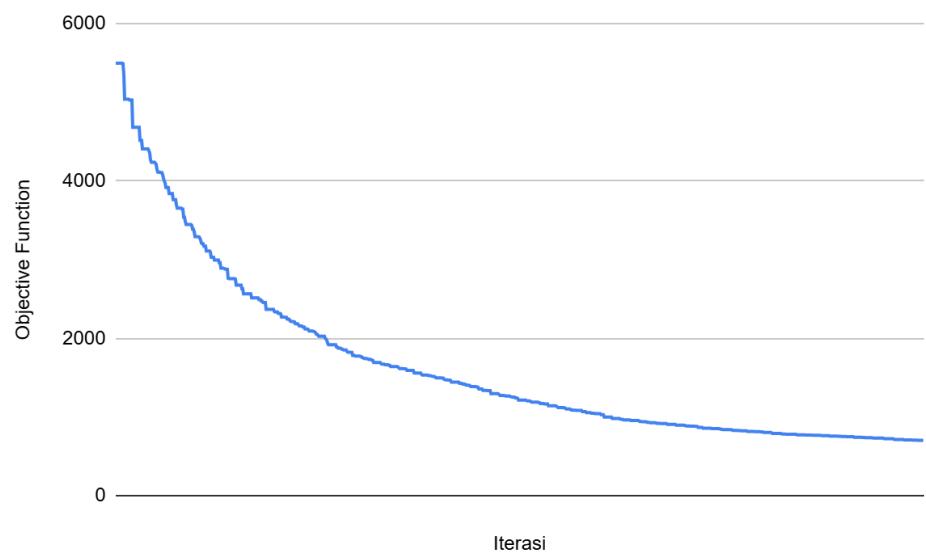
State Akhir

best last iteration with value: 702



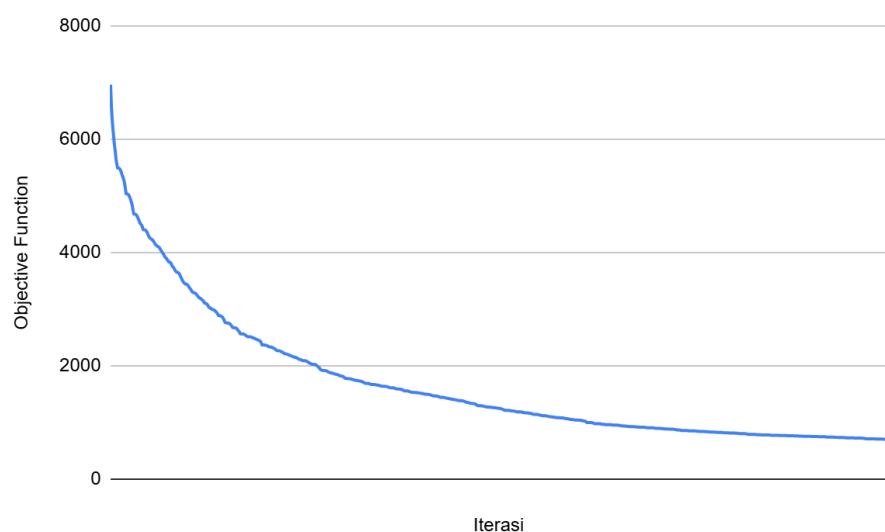
Gambar 72. Final State

Plot Objective Function Maximum



Gambar 73. Plot Objective Function Maximum

Plot Objective Function Average



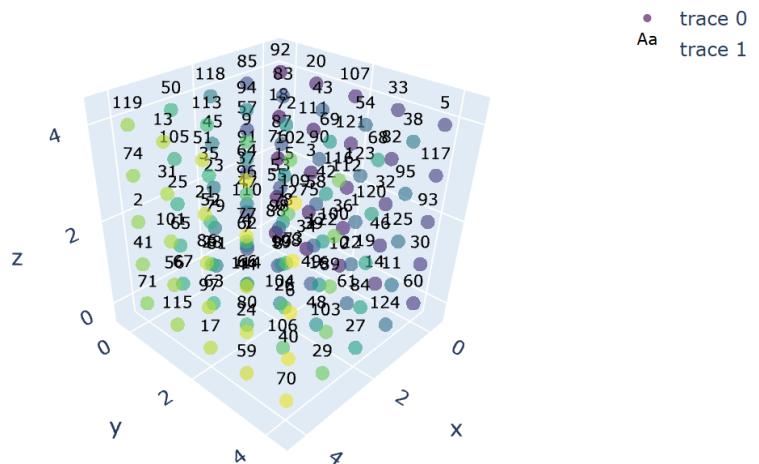
Gambar 74. Plot Objective Function Average

Percobaan 3

Waktu yang dibutuhkan = 997.759081 second(s)

State Awal

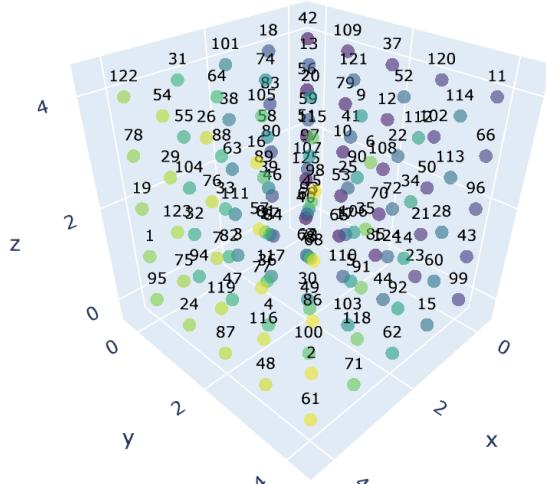
best of initialize iteration from 500 population(s) with value: 5327



Gambar 75. Initial State

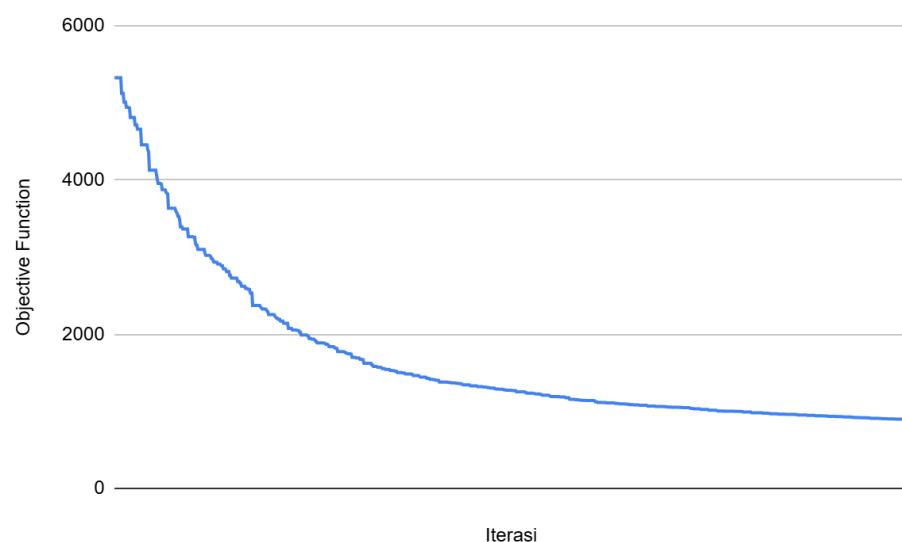
State Akhir

best last iteration with value: 893



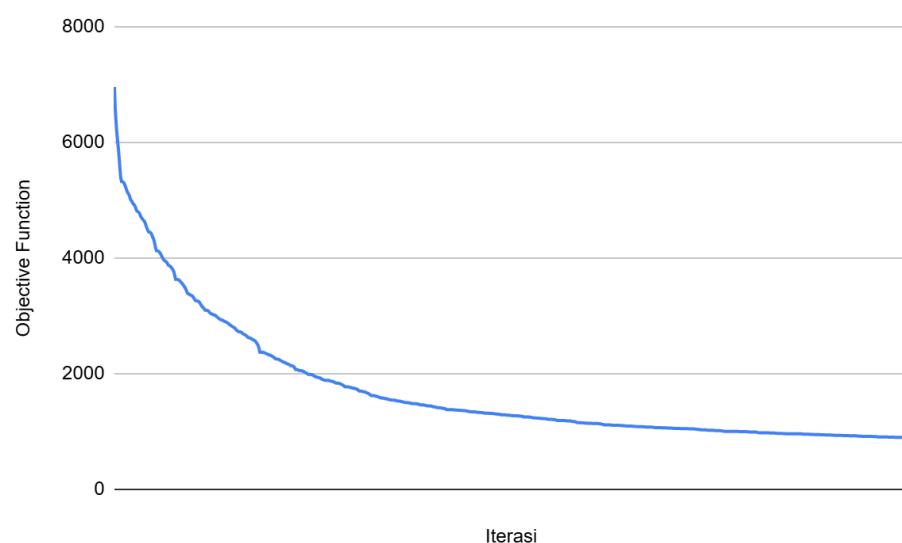
Gambar 76. Final State

Plot Objective Function Maximum



Gambar 77. Plot Objective Function Maximum

Plot Objective Function Average



Gambar 78. Plot Objective Function Average

Jumlah Iterasi sebagai Kontrol

Jumlah iterasi= 100

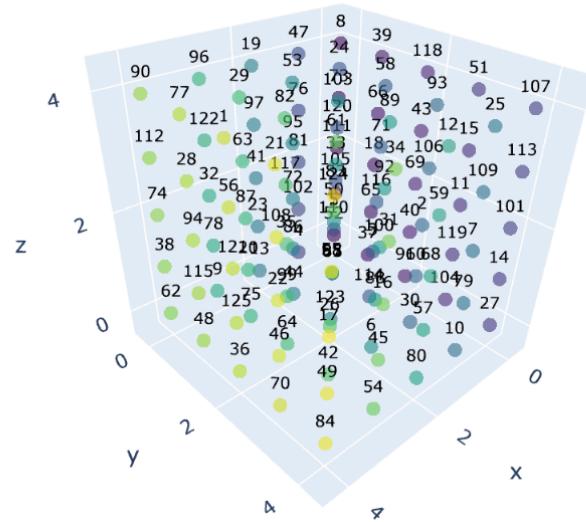
- d. Variasi Populasi 1
- Jumlah populasi= 100

Percobaan 1

Waktu yang dibutuhkan = 17.126181 second(s)

State Awal

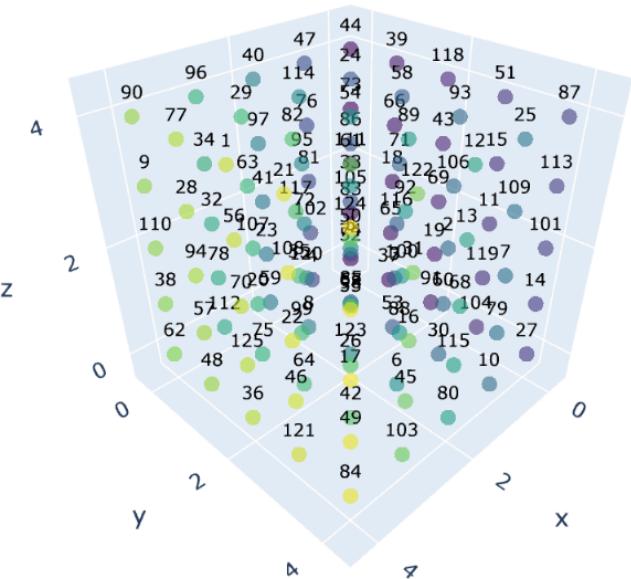
best of initialize iteration from 100 population(s) with value: 5624



Gambar 79. Initial State

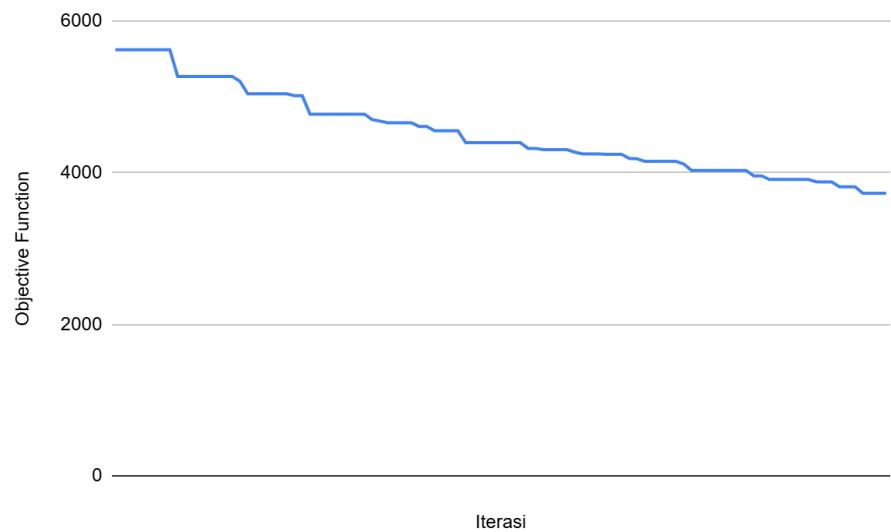
State Akhir

best last iteration with value: 3729



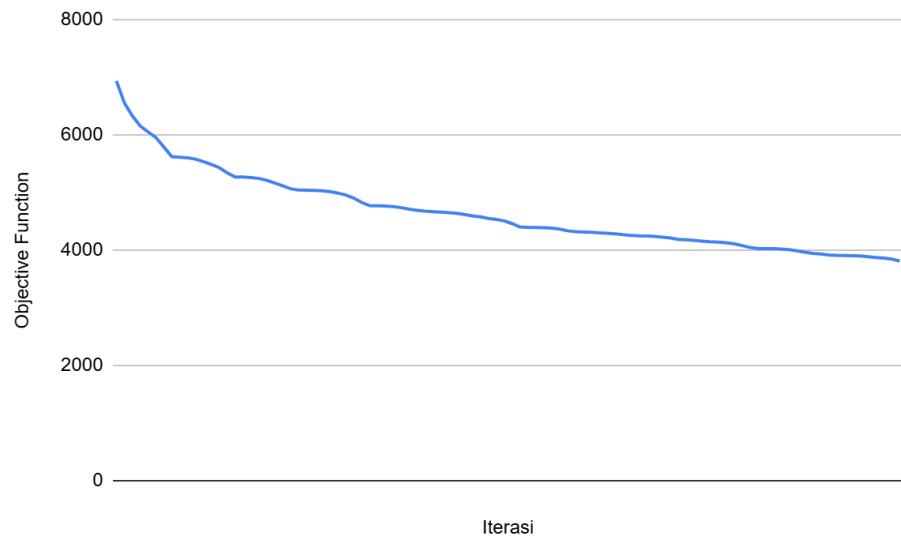
Gambar 80. Final State

Plot Objective Function Maximum



Gambar 81. Plot Objective Function Maximum

Plot Objective Function Average



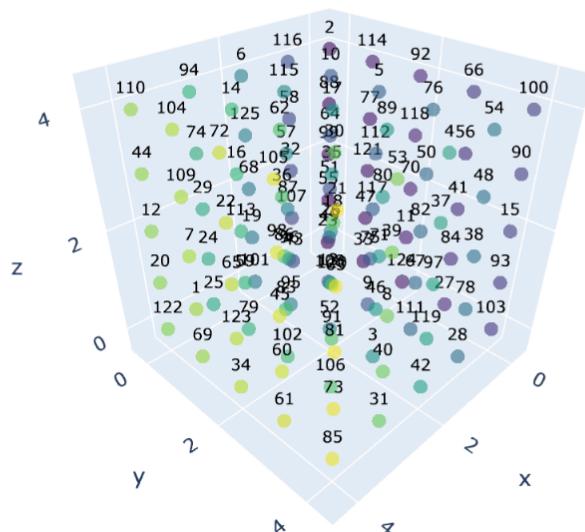
Gambar 82. Plot Objective Function Average

Percobaan 2

Waktu yang dibutuhkan = 11.325574 second(s)

State Awal

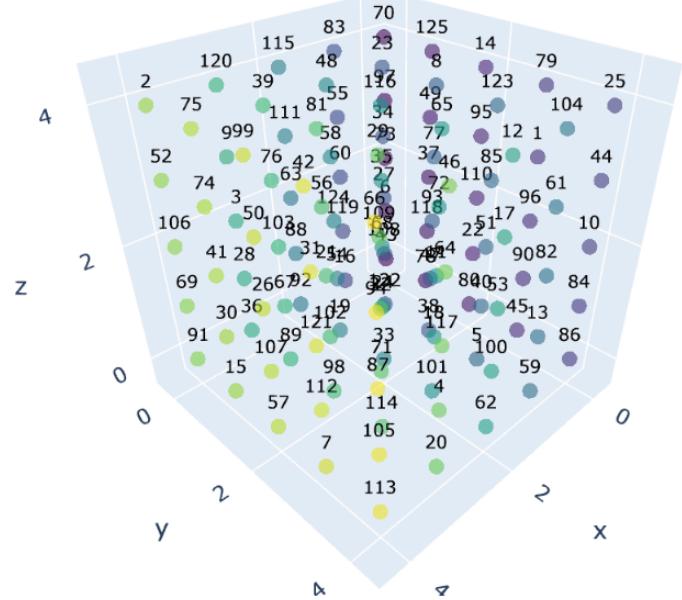
best of initialize iteration from 100 population(s) with value: 5713



Gambar 83. Initial State

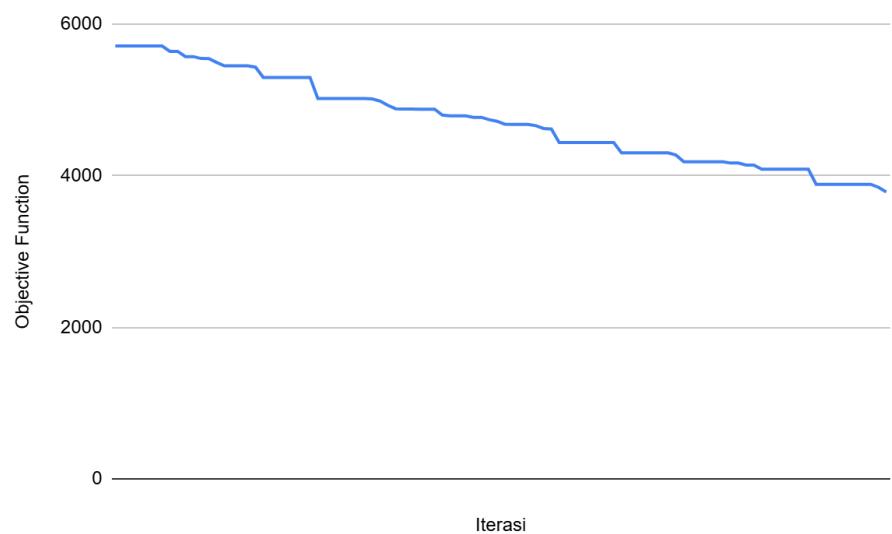
State Akhir

best last iteration with value: 3787



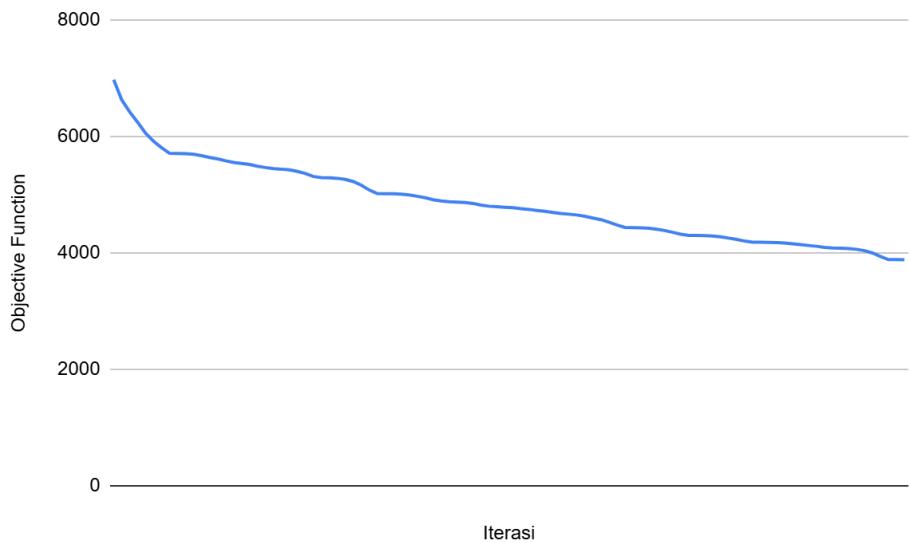
Gambar 84. Final State

Plot Objective Function Maximum



Gambar 85. Plot Objective Function Maximum

Plot Objective Function Average



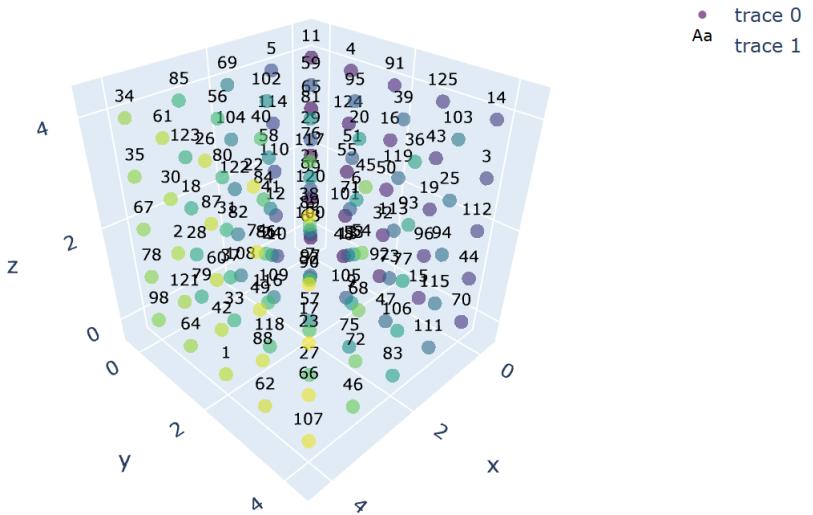
Gambar 86. Plot Objective Function Average

Percobaan 3

Waktu yang dibutuhkan = 18.164963 second(s)

State Awal

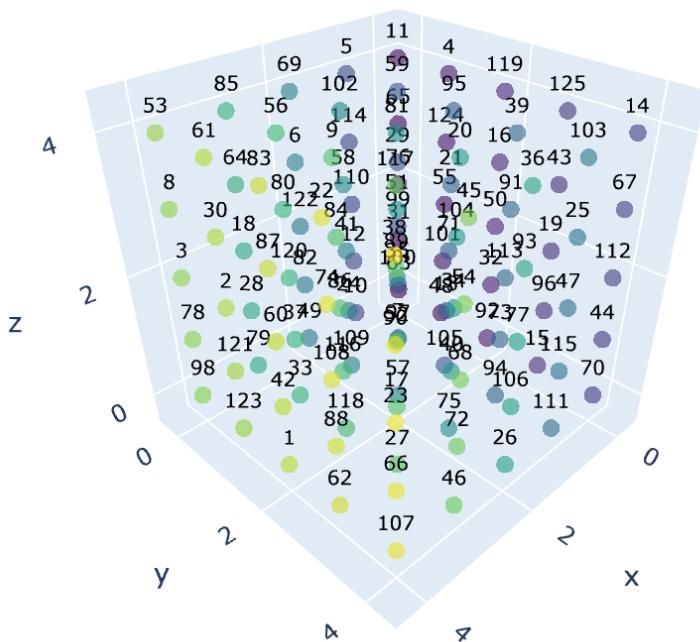
best of initialize iteration from 100 population(s) with value: 5755



Gambar 87. Initial State

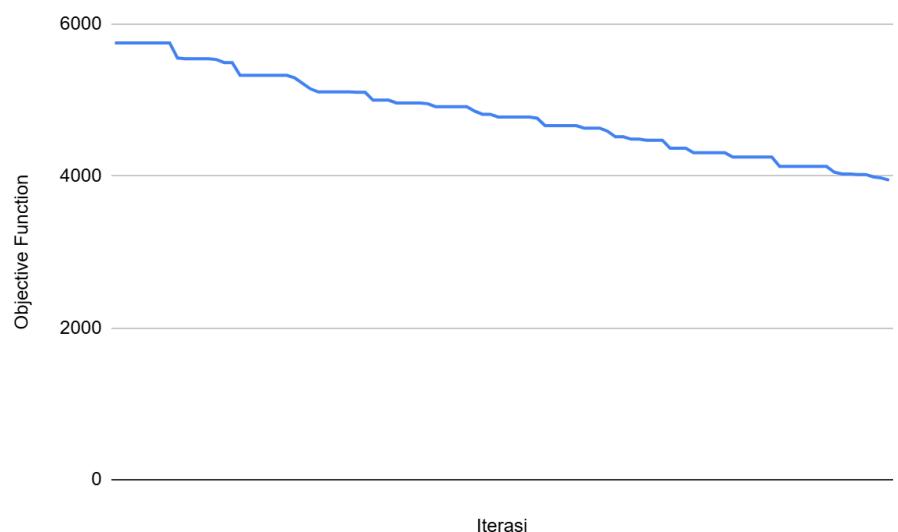
State Akhir

best last iteration with value: 3947



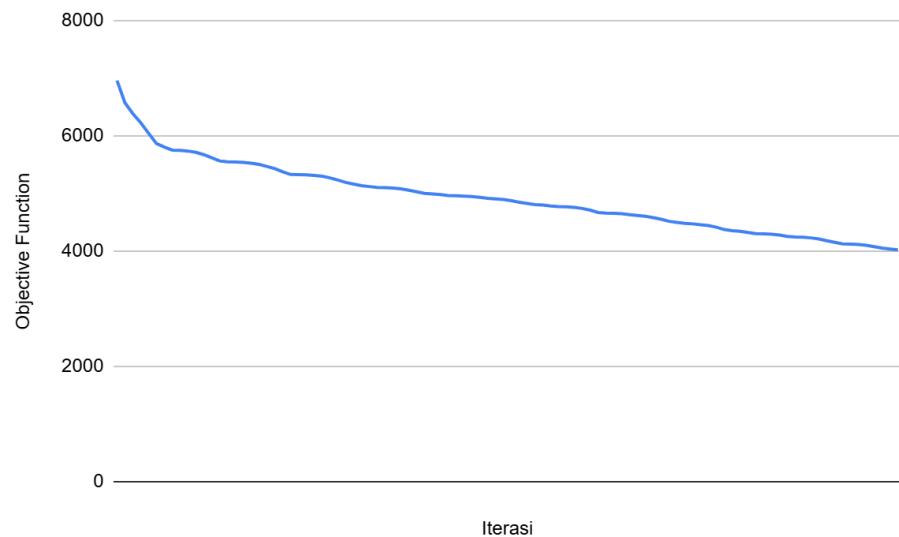
Gambar 88. Final State

Plot Objective Function Maximum



Gambar 89. Plot Objective Function Maximum

Plot Objective Function Average

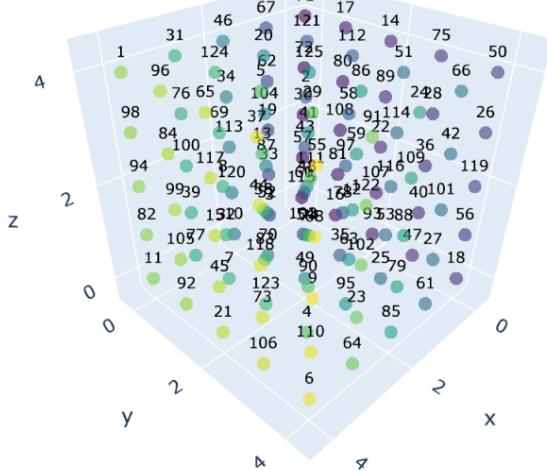


Gambar 90. Plot Objective Function Average

- e. Variasi Populasi 2
Jumlah populasi= 1000

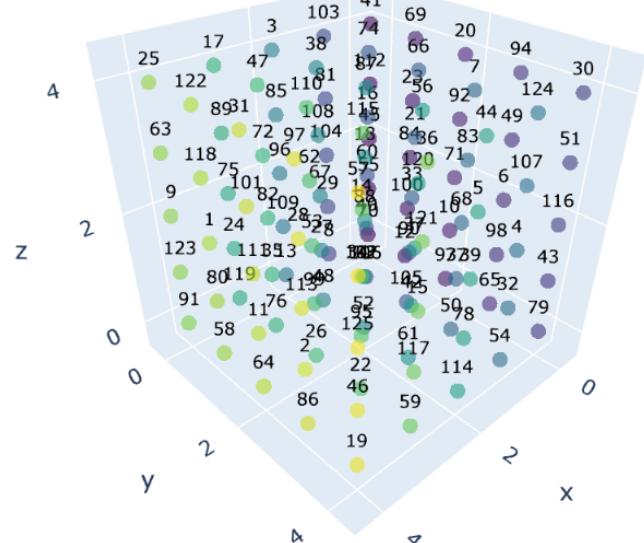
Percobaan 1
Waktu yang dibutuhkan =208.308606 second(s)
State Awal

best of initialize iteration from 1000 population(s) with value: 5565



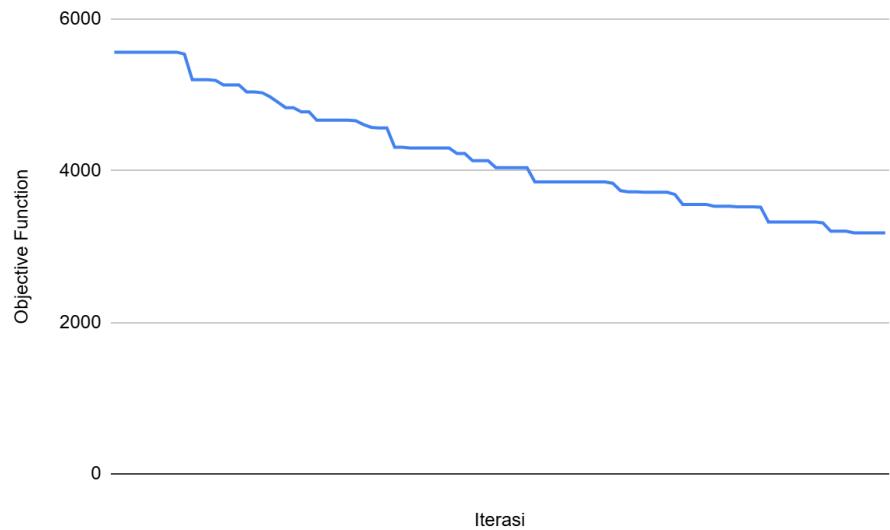
Gambar 91. Initial State
State Akhir

best last iteration with value: 3181



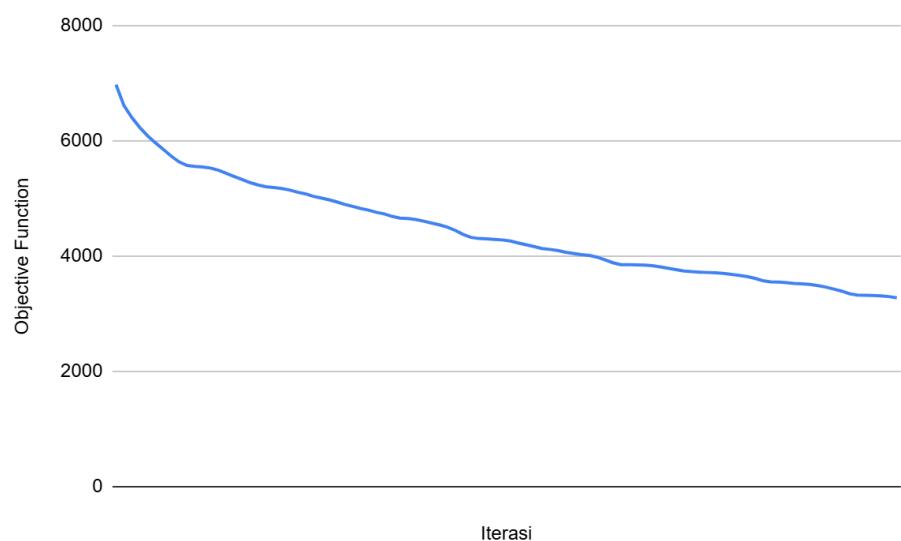
Gambar 92. Final State

Plot Objective Function Maximum



Gambar 93. Plot Objective Function Maximum

Plot Objective Function Average



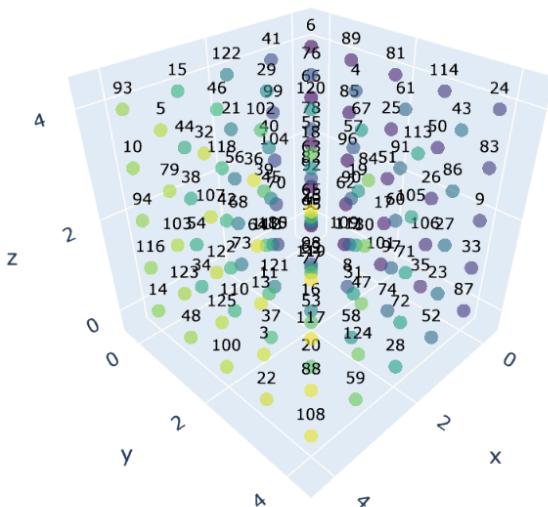
Gambar 94. Plot Objective Function Average

Percobaan 2

Waktu yang dibutuhkan = 131.207083 second(s)

State Awal

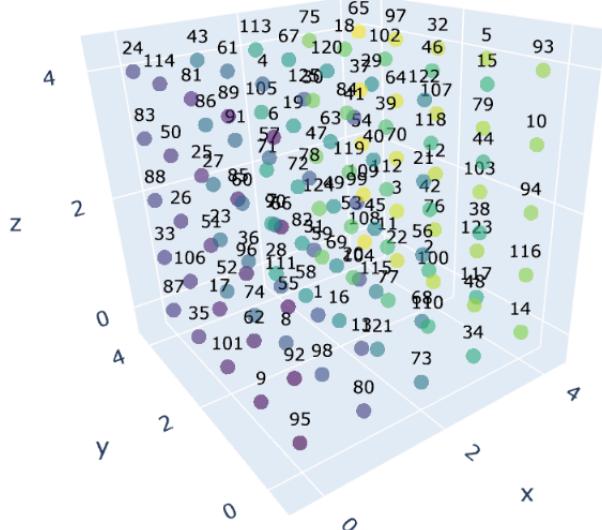
best of initialize iteration from 1000 population(s) with value: 5453



Gambar 95. Initial State

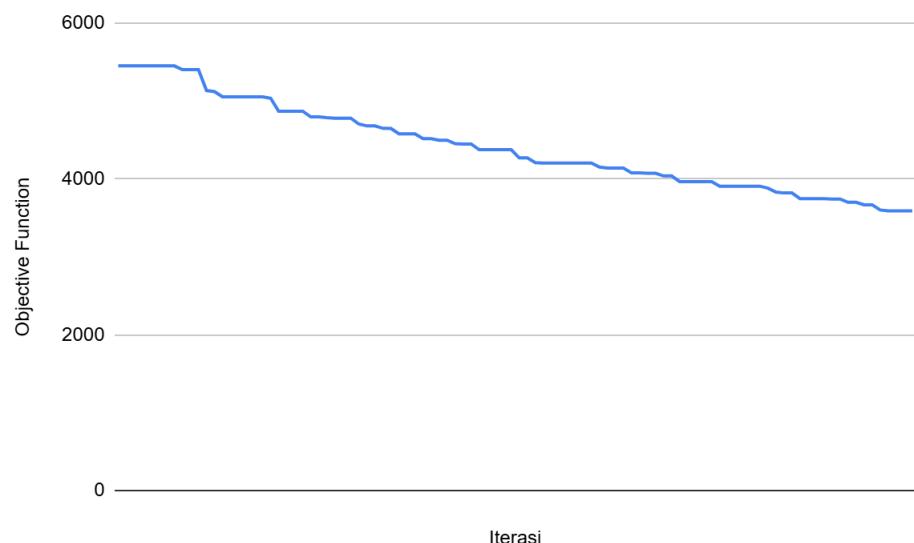
State Akhir

best last iteration with value: 3591



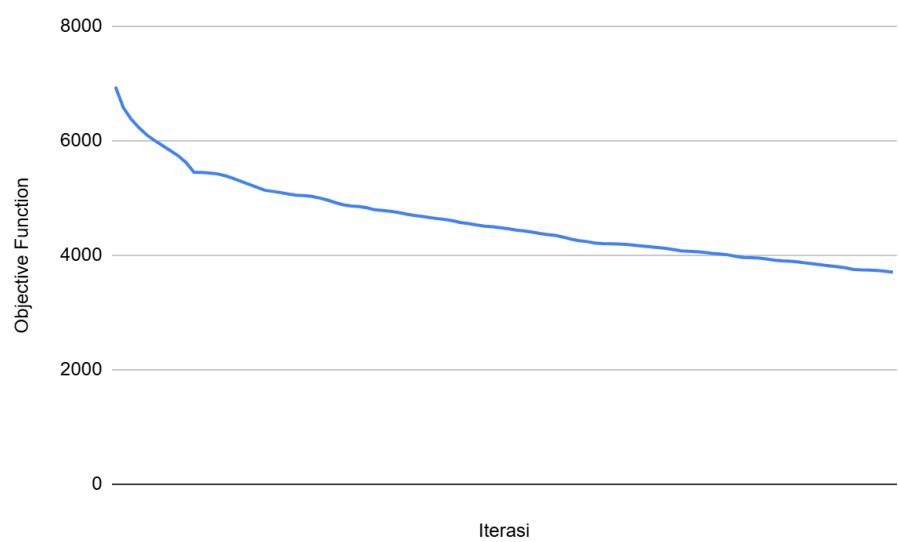
Gambar 96. Final State

Plot Objective Function Maximum



Gambar 97. Plot Objective Function Maximum

Plot Objective Function Average



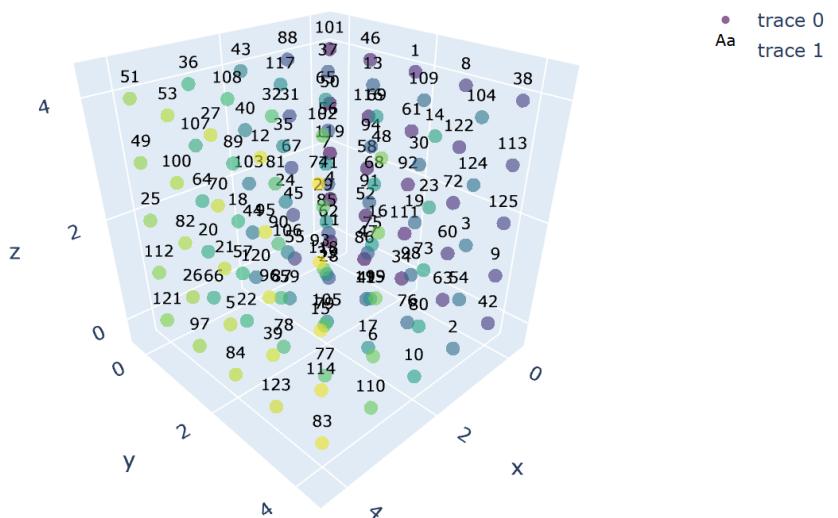
Gambar 98. Plot Objective Function Maximum

Percobaan 3

Waktu yang dibutuhkan = 200.120063 second(s)

State Awal

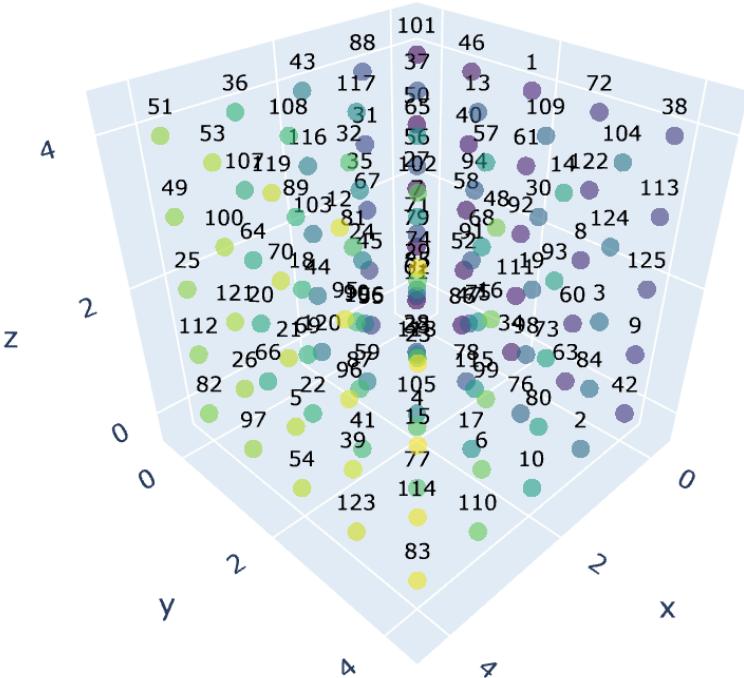
best of initialize iteration from 1000 population(s) with value: 5347



Gambar 99. Initial State

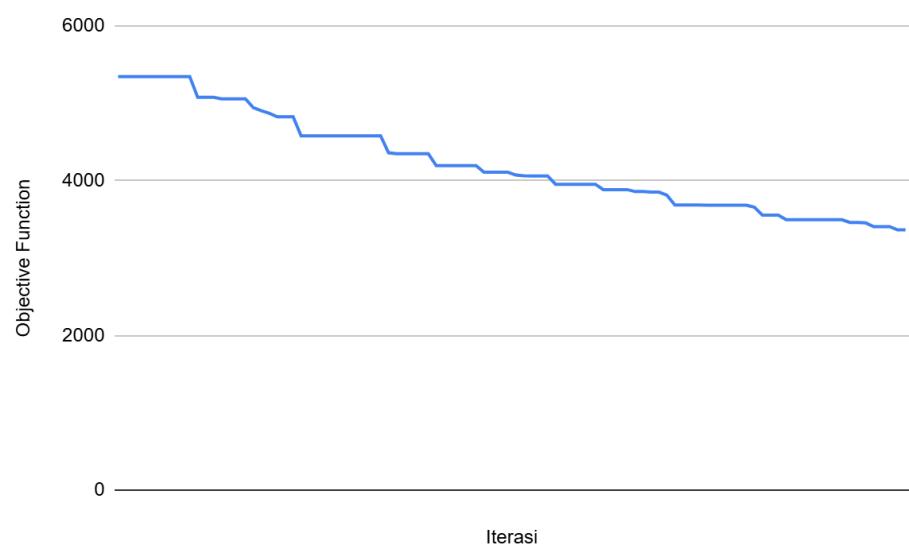
State Akhir

best last iteration with value: 3364



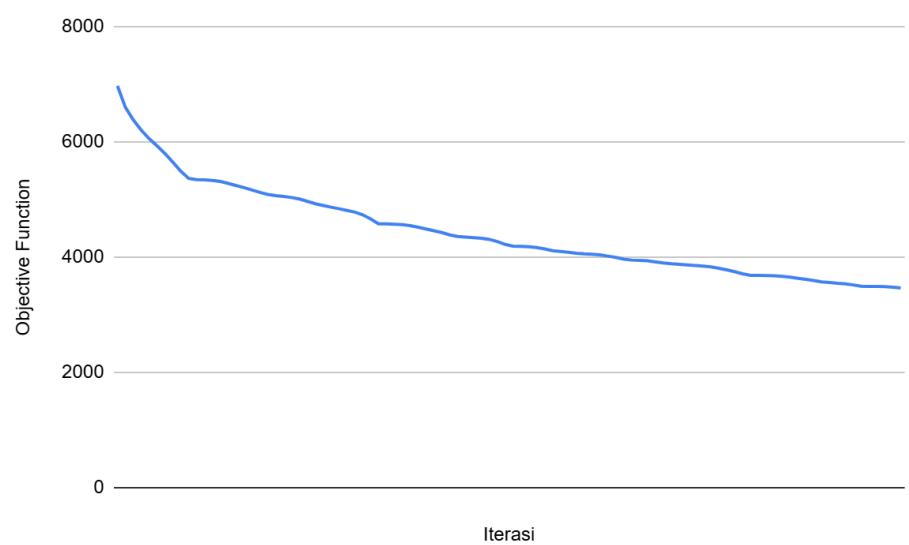
Gambar 100. Final State

Plot Objective Function Maximum



Gambar 101. Plot Objective Function Maximum

Plot Objective Function Average



Gambar 102. Plot Objective Function Average

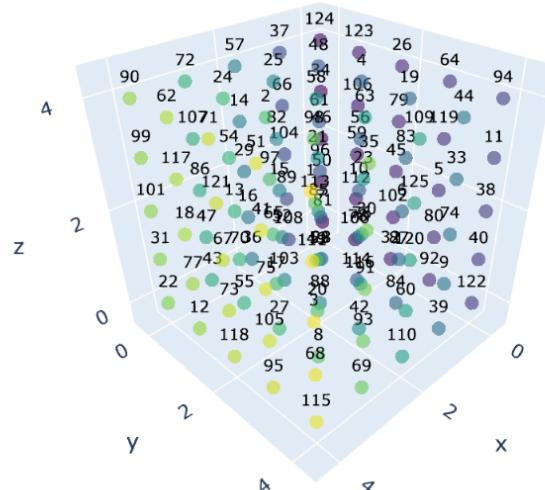
- f. Variasi Populasi 3
Jumlah populasi= 4

Percobaan 1

Waktu yang dibutuhkan = 1.902713 second(s)

State Awal

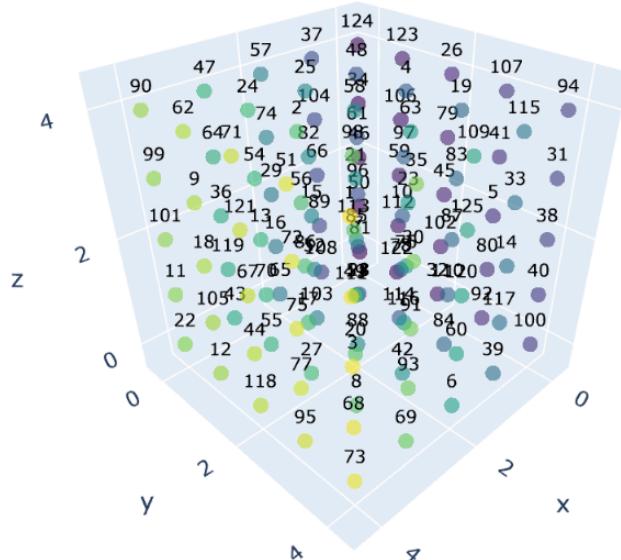
best of initialize iteration from 4 population(s) with value: 6803



Gambar 103. Initial State

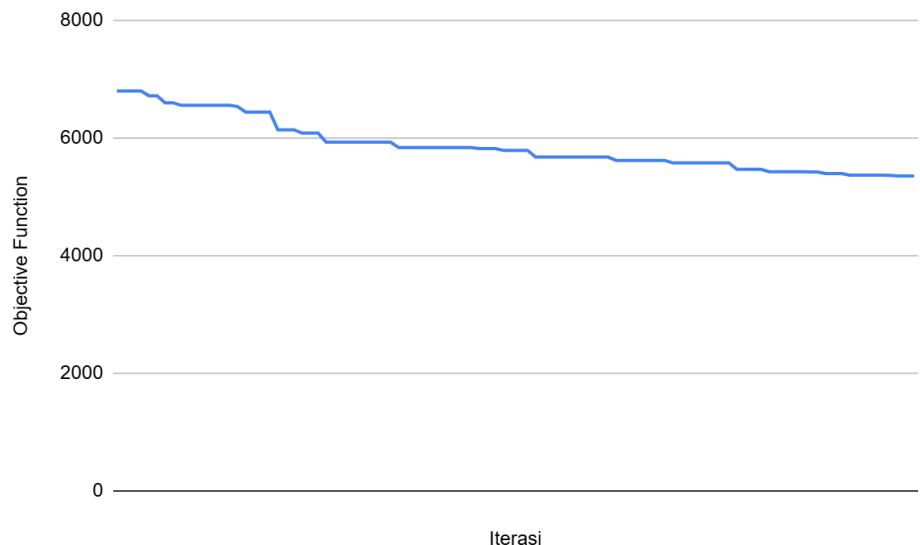
State Akhir

best last iteration with value: 5358



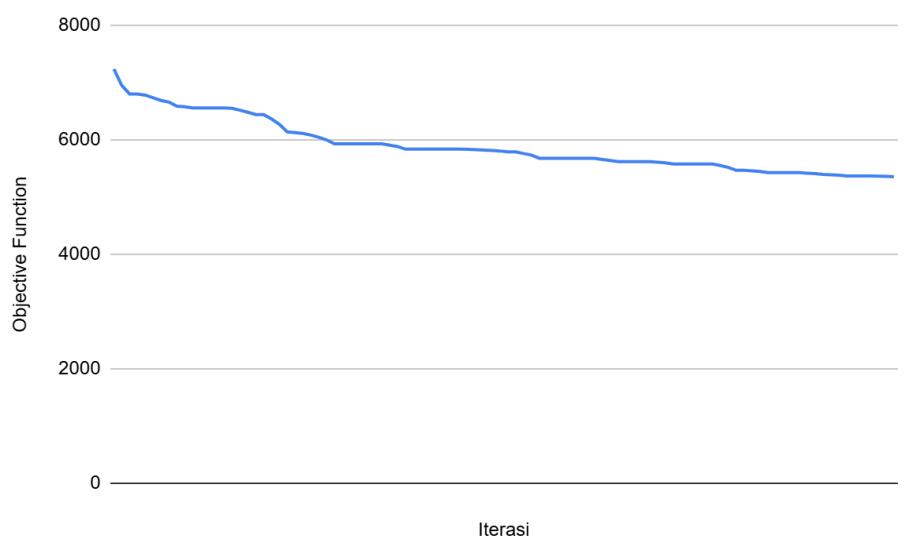
Gambar 104. Final State

Plot Objective Function Maximum



Gambar 105. Plot Objective Function Maximum

Plot Objective Function Average



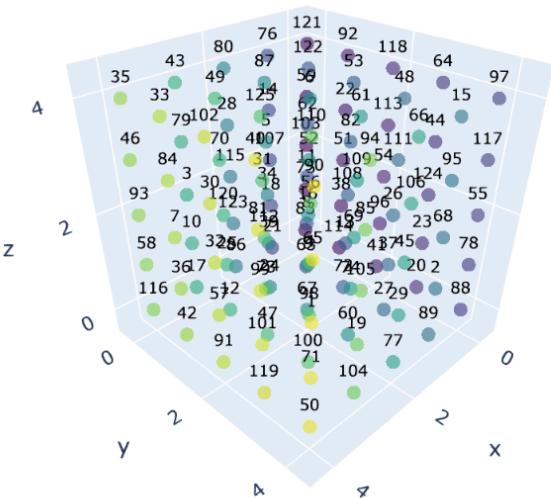
Gambar 106. Plot Objective Function Average

Percobaan 2

Waktu yang dibutuhkan = 1.125098 second(s)

State Awal

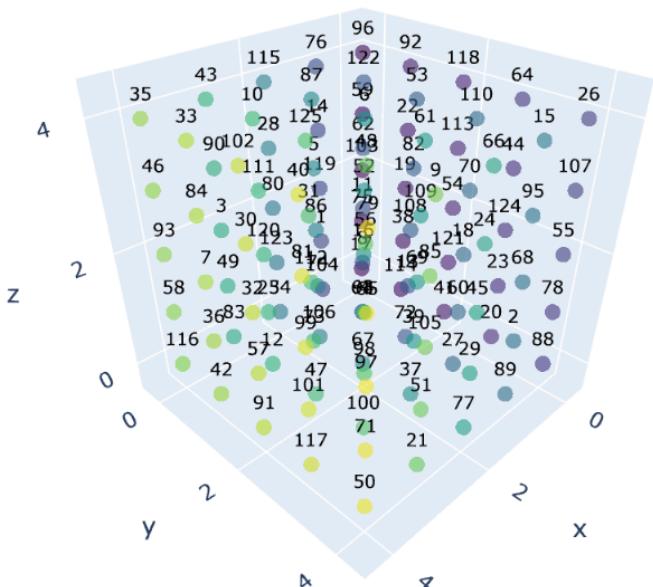
best of initialize iteration from 4 population(s) with value: 6502



Gambar 107. Initial State

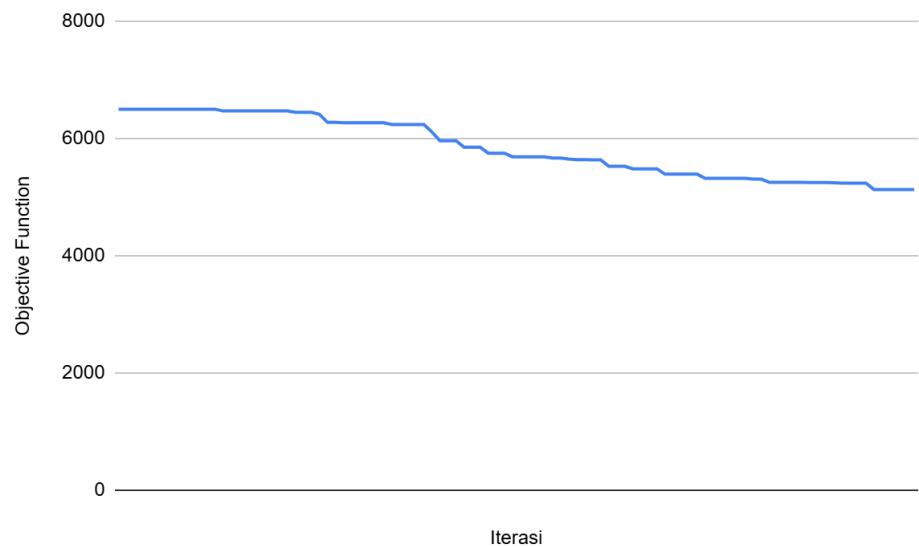
State Akhir

best last iteration with value: 5132



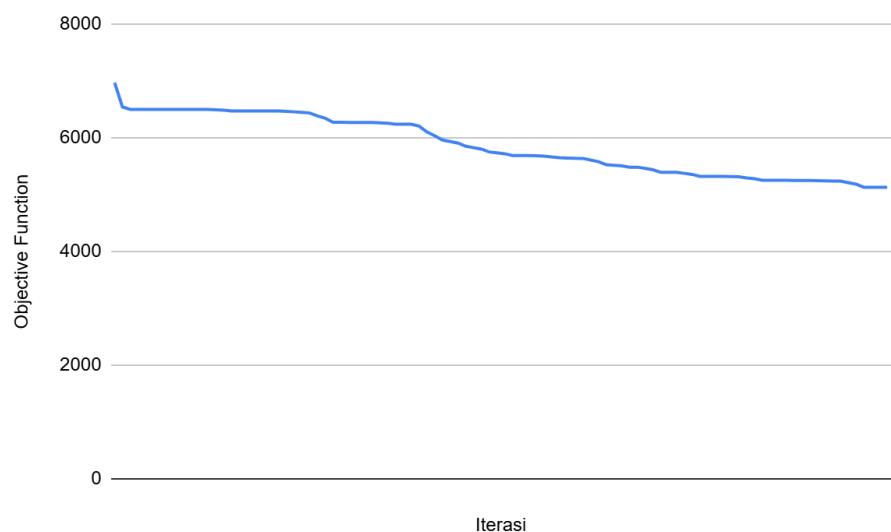
Gambar 108. Final State

Plot Objective Function Maximum



Gambar 109. Plot Objective Function Maximum

Plot Objective Function Average



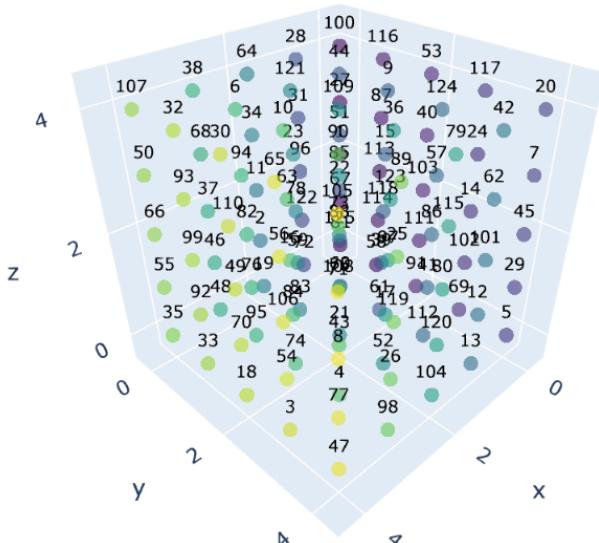
Gambar 110. Plot Objective Function Average

Percobaan 3

Waktu yang dibutuhkan = 0.623082 second(s)

State Awal

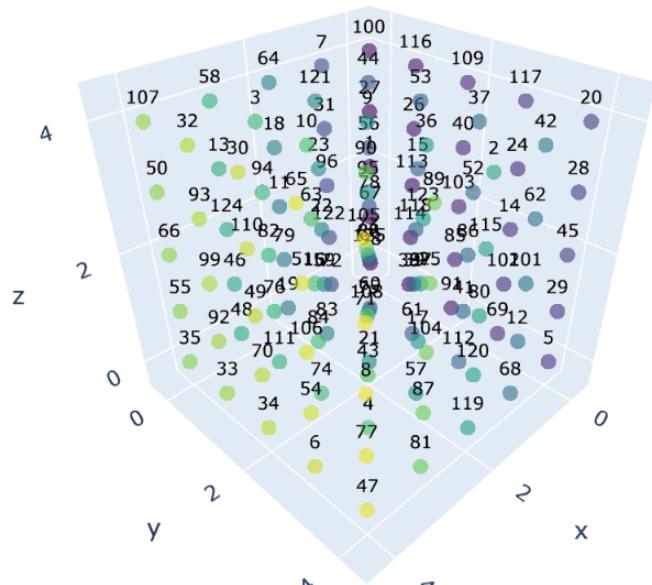
best of initialize iteration from 4 population(s) with value: 6360



Gambar 111. Initial State

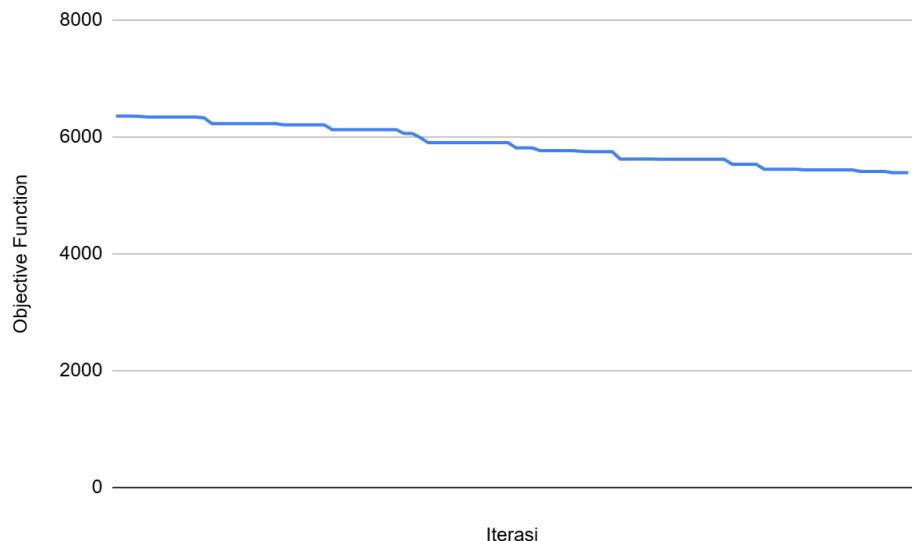
State Akhir

best last iteration with value: 5391



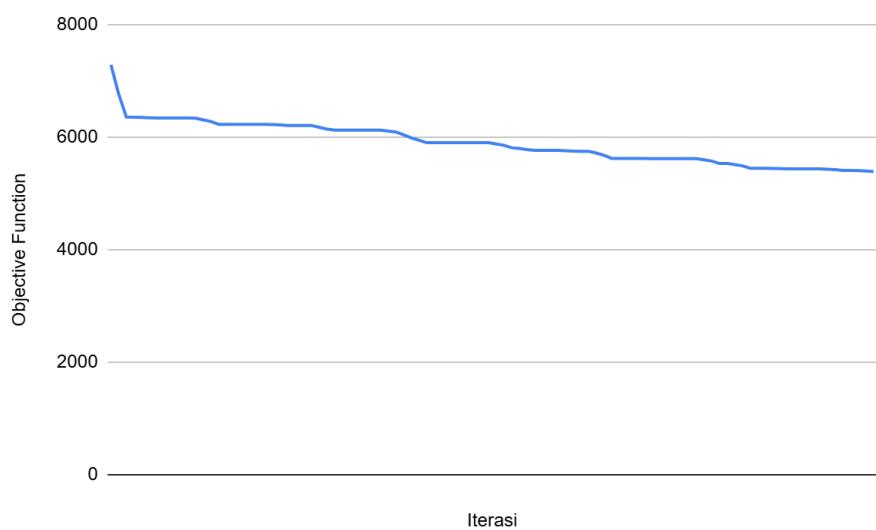
Gambar 112. Final State

Plot Objective Function Maximum



Gambar 113. Plot Objective Function Maximum

Plot Objective Function Average



Gambar 114. Plot Objective Function Average

Berdasarkan hasil data yang diberikan menggunakan variasi kontrol jumlah populasi dan banyak iterasi dengan

masing-masing perbedaan parameter dan banyaknya percobaan didapatkan kesimpulan data sebagai berikut.

Tabel 5. Hasil Genetic Algorithm

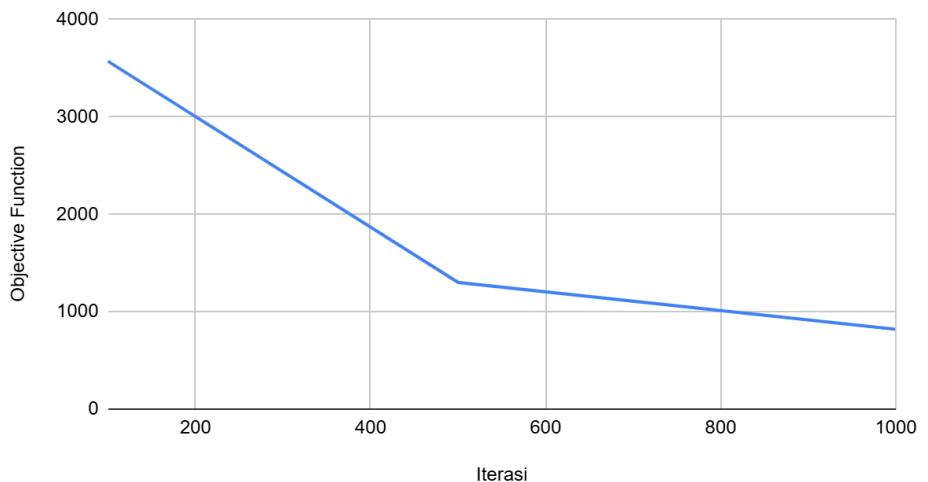
Kontrol	Variasi	Jumlah	Percobaan	Waktu (s)	Rata-Rata Waktu (s)	Objective Function	Rata-rata Objective Function	Standar Deviasi Objective Function
Jumlah Populasi = 500	Banyak Iterasi	100	1	93.045912	94.47667367	3815	3571	227.3939313
			2	97.239923		3365		
			3	93.144186		3533		
		500	1	484.023242	429.432276	1124	1299.666667	166.8422409
			2	326.314432		1456		
			3	477.959154		1319		
		1000	1	644.900125	884.9666607	863	819.333333	102.7148155
			2	1012.240776		702		
			3	997.759081		893		
Banyak Iterasi = 100	Jumlah Populasi	4	1	1.902713	1.216964333	5358	5293.666667	140.9763574
			2	1.125098		5132		
			3	0.623082		5391		
		100	1	17.126181	15.538906	3729	3821	112.9070414
			2	11.325574		3787		
			3	18.164963		3947		
		1000	1	208.308606	179.878584	3181	3380.333333	205.2348249
			2	131.207083		3591		

			3	200.1200 63		3369		
--	--	--	---	----------------	--	------	--	--

Berdasarkan tabel diatas dapat disimpulkan pengaruh antara masing-masing parameter seperti berikut.

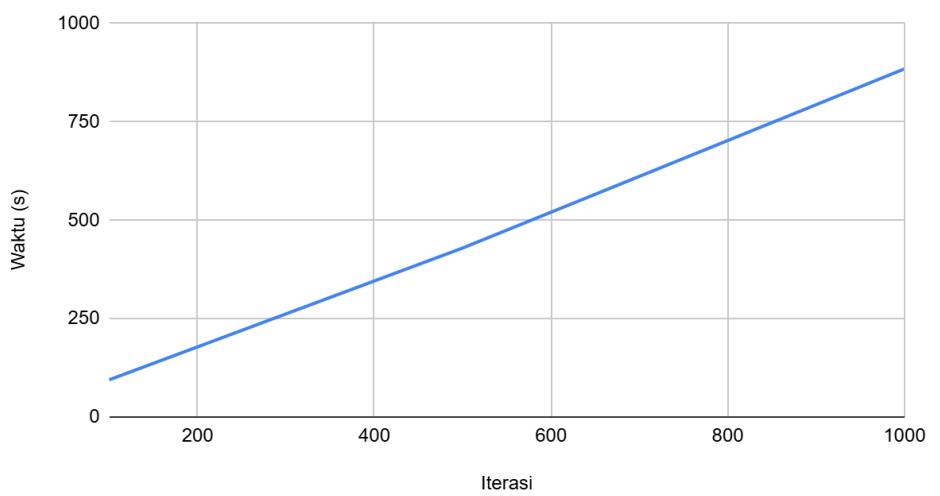
Jumlah Populasi sebagai Kontrol

Objective Function vs Iterasi



Gambar 115. Objective Function vs Iterasi

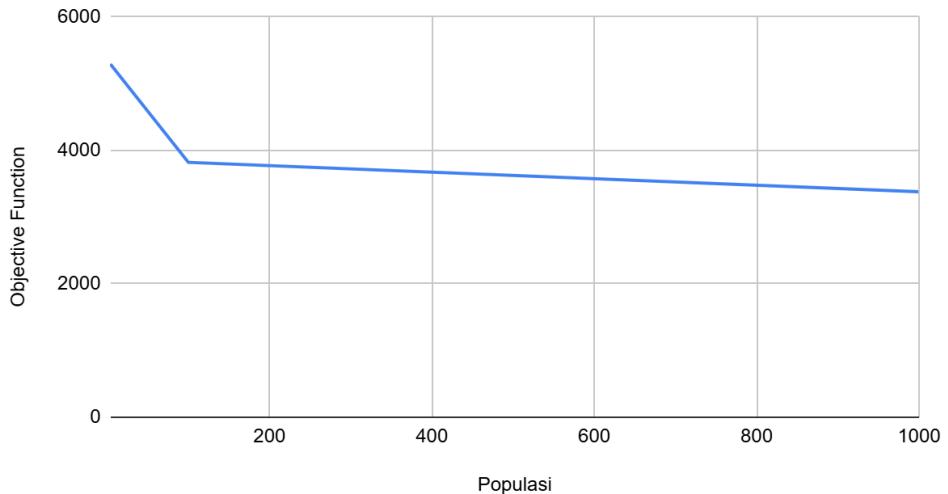
Waktu (s) vs Iterasi



Gambar 116. Waktu vs Iterasi

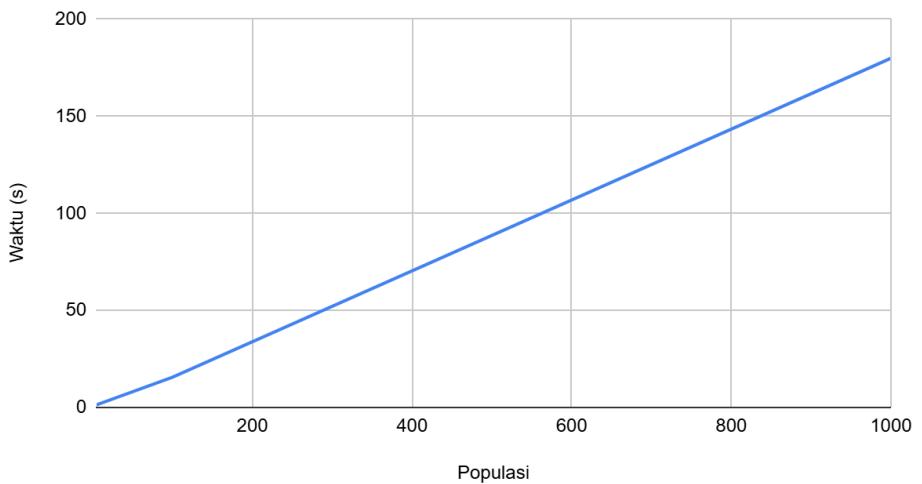
Banyak Iterasi sebagai Kontrol

Objective Function vs Populasi



Gambar 117. Objective Function vs Populasi

Waktu (s) vs Populasi



Gambar 118. Waktu vs Populasi

Untuk mengetahui seberapa dekat genetic algorithm untuk mendekati global optima dapat dilihat dari nilai objective function yang didapatkan dari masing-masing percobaan. Semakin kecil nilai objective function yang didapatkan menunjukkan semakin dekat hasil pencarian dengan global Optima. Untuk genetic algorithm semakin banyak iterasi yang yang diberikan dan semakin besar jumlah populasi yang

diberikan di algoritma akan menghasilkan nilai objective function yang semakin rendah yang menunjukkan bahwa genetic algorithm mampu mendekati global optima dengan kombinasi parameter yang lebih tinggi. Tetapi untuk menentukan solusi yang optimal peningkatan ukuran populasi dan penambahan banyaknya iterasi tidak bisa selalu dilakukan, karena terdapat batas nilai dimana peningkatan nilai melewati batas tersebut akan tidak memberikan peningkatan nilai yang signifikan dalam kualitas solusi yaitu menurunnya nilai objective function.

Jika dibandingkan dengan algoritma lain, Genetic Algorithm membutuhkan waktu lebih banyak dalam mendekati nilai global optimanya. Dapat dilihat jika dibandingkan dengan algoritma simulated annealing yang dapat melakukan 20.000 iterasi dalam waktu 22,5 detik, genetic algorithm butuh waktu hingga 94 detik hanya untuk melakukan 100 iterasi. Hal tersebut dapat disebabkan karena genetic algoritma memerlukan lebih banyak waktu untuk melakukan proses crossover dan mutasi populasi yang membutuhkan kerja yang lebih kompleks.

Dari data diatas dapat diketahui saat menggunakan jumlah populasi sebagai kontrol maka semakin besar banyak iterasi yang dilakukan maka semakin lama waktu pencarinya. Begitu pula saat banyak iterasi sebagai kontrolnya, maka saat jumlah populasinya semakin banyak, pencarian yang dilakukan akan memakan waktu yang lebih lama. Sehingga dapat diketahui bahwa durasi untuk melakukan pencarian akan meningkat seiring dengan meningkatnya jumlah populasi dan banyaknya iterasi yang digunakan dalam genetic algorithm.

Untuk menentukan konsistensi dari genetic algorithm dilakukan perhitungan standar deviasi untuk tiap variasi. Untuk setiap kombinasi kontrol yaitu jumlah populasi dan banyak iterasi, penentuan konsistensi dilakukan dengan membandingkan nilai rata-rata objective function dengan standar deviasinya. Misalkan untuk jumlah populasi 500 dan banyak iterasi 100 diketahui nilai rata-rata objectivenya sebesar 3571 dan standar deviasi objective function nya

sebesar 227.39, dari nilai tersebut terdapat fluktuasi sebesar 6.4% dari nilai rata-ratanya yang masih dibilang cukup rendah.

Nilai fluktuasi yang rendah menunjukkan bahwa hasil pencarian menggunakan genetic algorithm cukup konsisten.

3. KESIMPULAN DAN SARAN

Kesimpulan dari penggerjaan tugas besar ini adalah untuk melakukan pencarian solusi diagonal magic cube algoritma local search terbaik yang dapat digunakan adalah Simulated Annealing. Hal tersebut dapat ditentukan karena pada waktu yang sama, algoritma simulated annealing dapat melakukan iterasi yang lebih banyak dibandingkan algoritma Hill-Climbing dan Genetic Algoritma dan menghasilkan nilai objective function yang lebih rendah yang menunjukkan semakin dekatnya solusi ke nilai global optimanya. Hal tersebut dapat terjadi karena Simulated Annealing melakukan pencarian lebih fleksibel dan adaptif. Algoritma Simulated Annealing dapat menghindari terjebak dalam local optima seperti yang dialami oleh algoritma Hill-Climbing dikarenakan dalam pencarinya solusinya Simulated Annealing menggunakan mekanisme probabilistik sehingga dapat menuju ke berbagai solusi dalam ruang solusi walaupun bisa jadi solusi yang dituju lebih buruk dibandingkan solusi sebelumnya. Algoritma Simulated Annealing dapat melakukan iterasi yang lebih cepat dalam menyelesaikan pencarian magic cube dibandingkan dengan genetic algorithm sehingga mampu menjelajah ruang solusi lebih optimal. Algoritma Simulated Annealing juga lebih konsisten dalam mengurangi nilai objective function ditunjukkan akan korelasi tiap parameter yang digunakan. Sehingga untuk melakukan pencarian solusi magic cube, algoritma Simulated Annealing merupakan pilihan yang tepat untuk menghasilkan solusi yang efektif dan optimal.

Saran dari penggerjaan tugas besar ini adalah perlu dilakukan optimasi lebih lanjut terhadap algoritma yang digunakan. Hal ini terutama sangat berpengaruh karena penggerjaan tugas besar dilakukan dengan bahasa Python yang merupakan *interpreted language* sehingga cenderung lambat. Apabila memungkinkan dan kinerja merupakan prioritas utama, maka penggunaan bahasa yang lebih *low-level* seperti C atau C++ dapat dipertimbangkan. Salah satu algoritma yang dapat dioptimasi misalnya algoritma fungsi objektif. Optimasi terhadap algoritma tersebut dapat dilakukan dengan memperbanyak penggunaan *vectorised function* dari Numpy yang berbasis C. Hal ini akan berdampak besar karena fungsi objektif digunakan di seluruh algoritma *local search* sebagai heuristik. Selain fungsi objektif, potensi optimasi terbesar ada pada algoritma *hill climbing steepest ascent* dan *hill climbing with sideways move* yang memiliki *time complexity* $O(n^6)$

sehingga sangat boros secara komputasi. Ada beberapa potensi optimasi, mulai dari penggunaan algoritma heuristik yang lebih "cerdas", mengurangi ruang pencarian, menggunakan *parallel processing* (misalnya mp dari Python), maupun teknik lain seperti Tabu search. Terakhir, ada isu terhadap konsensus penamaan variabel yang terkadang ambigu, rancu, dan membingungkan. Misalnya, terkait variabel untuk menyimpan *best state* dan *best candidate state*. Perlu dilakukan identifikasi lebih mendalam untuk menentukan skema penamaan yang lebih jelas.

4. PEMBAGIAN TUGAS

Pengerjaan tugas besar 1 IF3170 ini dikerjakan oleh lima orang dengan pembagian tugas sebagai berikut:

NIM	Nama	Kontribusi
13221011	Jazila Faza Aliyya Nurfauzi	Membuat algoritma visualisasi
		Mengerjakan laporan (major)
13221055	Ahmad Hafiz Aliim	Membuat algoritma <i>genetic</i>
13221065	Caitleen Devina	Membuat algoritma visualisasi
		Mengerjakan laporan (major)
18221130	Rayhan Maheswara Pramanda	Membuat seluruh algoritma <i>hill climbing</i>
		Membuat algoritma fungsi objektif
		Membuat fitur <i>video player</i>
		Mengerjakan laporan (minor)
18321008	Jasmine Callista Aurellie Irfan	Membuat algoritma <i>simulated annealing</i>
		Membuat algoritma fungsi objektif
		Mengerjakan laporan (minor)

5. REFERENSI

Russell, Stuart J., and Peter Norvig. "Artificial intelligence: a modern approach."

Choice Reviews Online 33, no. 03 (November 1, 1995): 33–1577.

<https://doi.org/10.5860/choice.33-1577>.

Liem, I. "Draft Diktat Kuliah Dasar Pemrograman (Bagian Pemrograman Prosedural)." April 2007.

Plotly. "scatter3d." Plotly Python Reference.

<https://plotly.com/python/reference/scatter3d/>.

Plotly. "3D Scatter Plots in Python." Plotly Documentation.

<https://plotly.com/python/3d-scatter-plots/>.