TUGAS BESAR 1

IF3170 Inteligensi Artifisial

Pencarian Solusi Diagonal Magic Cube dengan Local Search

Disusun Oleh:

13522061	**Maximilian Sulistiyo**
13522075	**Marvel Pangondian**
13522083	**Evelyn Yosiana**
13522103	**Steven Tjhia**
13521154	**Naufal Baldemar Ardanni**

Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

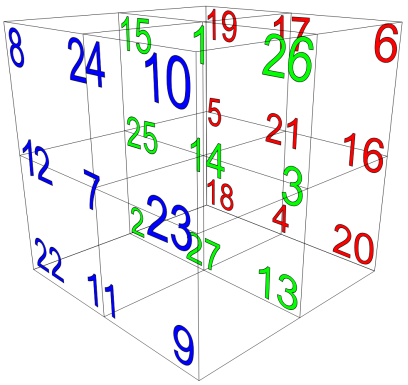
DAFTAR ISI

BAB 1

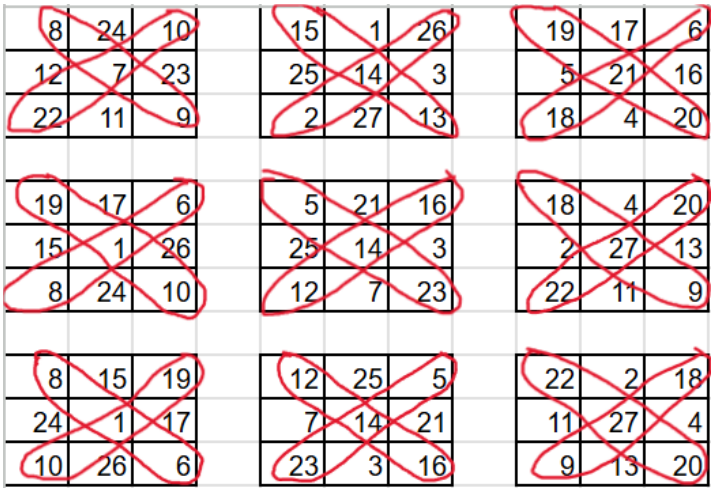
Deskripsi Persoalan

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

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



    - Terdapat 9 potongan bidang, yaitu:



    - Diagonal yang dimaksud adalah yang dilingkari warna merah saja
- Ilustrasi dan penjelasan lebih detail bisa anda lihat di link berikut: [Features of the magic cube - Magisch vierkant]

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

Anda akan diminta untuk mengimplementasikan rencana yang telah Anda buat pada Tugas Kecil 1. Berikut merupakan hal-hal yang perlu dilakukan oleh setiap kelompok:

- Implementasikan 3 algoritma local search dengan rincian sebagai berikut:
    - Salah satu algoritma hill-climbing
    - Simulated Annealing
    - Genetic Algorithm
- Lakukan eksperimen dengan skema sebagai berikut:
    - Jalankan setiap algoritma sebanyak **3 kali**, kemudian catat beberapa hal berikut:
        - Berlaku untuk semua algoritma
            - State awal dan akhir dari kubus
            - Nilai objective function akhir yang dicapai
            - Plot nilai objective function terhadap banyak iterasi yang telah dilewati
            - Durasi proses pencarian
        - Berlaku hanya untuk Steepest Ascent Hill-Climbing dan Stochastic Hill-Climbing
            - Banyak iterasi hingga proses pencarian berhenti
        - Berlaku hanya untuk Hill-Climbing with Sideways Move
            - Banyak iterasi hingga proses pencarian berhenti
            - Note: Tambahkan parameter maximum sideways move, dimana ketika banyak sideways move yang dilakukan sudah mencapai maksimum, pencarian dihentikan
        - Berlaku hanya untuk Random Restart Hill-Climbing
            - Banyak restart
            - Banyak iterasi per restart
            - Note: Tambahkan parameter maximum restart, dimana ketika banyak restart sudah mencapai maksimum, pencarian dihentikan.
        - Berlaku hanya untuk Simulated Annealing
            - Plot $e^{\frac{\Delta E}{T}}$ terhadap banyak iterasi yang telah dilewati
            - Frekuensi ‘stuck’ di local optima
    - Khusus untuk Genetic Algorithm, lakukan beberapa hal berikut:

        - Terdapat 2 parameter yang dapat diubah, yaitu **jumlah populasi** dan **banyak iterasi.**
        - Jadikan jumlah populasi sebagai kontrol, kemudian pilih **3 variasi** banyak iterasi yang berbeda. Jalankan program sebanyak masing-masing **3 kali** untuk setiap konfigurasi parameter.
        - Jadikan banyak iterasi sebagai kontrol, kemudian pilih **3 variasi** jumlah populasi yang berbeda. Jalankan program sebanyak masing-masing **3 kali** untuk setiap konfigurasi parameter.
        - Untuk setiap eksperimen, catat beberapa hal berikut:
            - State awal dan akhir dari kubus
            - Nilai objective function akhir yang dicapai
            - Plot nilai objective function terhadap banyak iterasi yang telah dilewati (Cukup plot nilai objective function maksimum dan rata-rata dari populasi terhadap banyak iterasi yang telah dilewati. Jika sudah terlanjur membuat plot untuk tiap individu pada populasi tidak menjadi masalah, silahkan diberikan keterangan saja maksud plotnya apa)
            - Jumlah populasi
            - Banyak iterasi
            - Durasi proses pencarian
- Lakukan analisis terhadap hasil eksperimen, berikut merupakan beberapa pertanyaan yang dapat menjadi acuan untuk analisis yang Anda lakukan (Anda boleh menambahkan beberapa pertanyaan tambahan jika dirasa perlu untuk dijelaskan):
    - Seberapa dekat tiap-tiap algoritma bisa mendekati global optima dan mengapa hasilnya demikian?
    - Bagaimana perbandingan hasil pencarian tiap-tiap algoritma dengan algoritma local search yang lain?
    - Bagaimana perbandingan durasi proses pencarian tiap algoritma relatif terhadap algoritma lainnya?
    - Seberapa konsisten hasil akhir yang didapatkan dari tiap-tiap eksperimen yang dilakukan?
    - Bagaimana pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada Genetic Algorithm?
    - dst…
- Program yang dibuat harus bisa **memvisualisasikan** state awal kubus, state akhir kubus, dan juga hasil eksperimennya (sesuaikan informasi hasil eksperimen yang ditampilkan dengan ketentuan yang telah dijelaskan di poin sebelum ini).
- Cara visualisasi dibebaskan kepada kelompok masing-masing selama seluruh angka pada kubus dan juga hasil eksperimen terlihat dengan jelas.
- Dibebaskan menggunakan bahasa pemrograman apapun.
- Diperbolehkan untuk menggunakan heuristik yang Anda buat sendiri atau dari referensi lain untuk optimasi pencarian solusi, asalkan masih dalam lingkup local search. Jangan lupa jelaskan heuristik yang Anda pakai di laporan.

BAB 2

Pembahasan

2.1. Pemilihan Objective Function

Objective Function yang ditetapkan oleh penulis pada setiap *state* adalah fungsi yang menghitung nilai dari setiap *state* berdasarkan jumlah angka pada setiap baris, kolom, tiang, diagonal ruang, dan diagonal pada suatu potongan bidang yang memiliki nilai 315. Nilai 315 ini didasarkan pada rumus magic number (S), yaitu:

$$S = m.(m^3 + 1)/2$$

Objective Function dimulai dengan mencari jumlah dari setiap baris, kolom, tiang, diagonal ruang, dan diagonal irisan bidang. Kemudian, untuk setiap jumlah tersebut akan dihitung banyaknya yang bernilai 315. Jumlah dari semua nilai yang bernilai 315 akan menjadi nilai dari *state* tersebut. Alasan pnulis memilih *objective function* adalah untuk mencerminkan tujuan dari *magic cube*, yaitu memastikan bahwa setiap baris, kolom, tiang, diagonal ruang, dan diagonal pada suatu potongan bidang memiliki jumlah yang sama dengan *magic number*.

2.2. Implementasi Class Cube

Class ini merupakan representasi dari Cube. Cube disini merupakan matrix 3 dimensi yang direpresentasikan menggunakan array 1D. Pemilihan ini digunakan untuk mempermudah proses komputasi karena komputasi array 1D lebih mudah dibandingkan proses array of array of array. Representasi ini pun digunakan untuk mempermudah *Genetic Algorithm* yang menggunakan representasi ini untuk melakukan crossover.

Pada class ini pun terdapat beberapa fungsi penting seperti

- toIndex : Menghitung index dari nilai posisi 3D
- Insert_value : Memasukkan suatu nilai ke dalam posisi 3D
- Calculate_state_value : Menghitung nilai state dari suatu cube dan memasukkannya ke dalam atribut cube
- get_value : Mendapatkan nilai dari suatu posisi
- Delete_value : Menghapus nilai dari suatu posisi
- print_cube : Mencetak cube
- objective _cube : Menghitung nilai state dari suatu cube
- generate_successor : Membangkitkan semua successor dari cube

Source Code

```
import random
import copy
import time

class Cube:
  def __init__(self, x_size, y_size, z_size, random=False, array=None):
    self.x_size = x_size
    self.y_size = y_size
    self.z_size = z_size
    self.array = [None] * (self.x_size * self.y_size * self.z_size)
    self.magic_number = (
      x_size * (x_size**3 + 1) // 2
      if x_size == y_size and y_size == z_size
      else None
    )
    if random:
      self.random_cube()
      self.state_value = objective_function(self)
    if array:
      self.array = array
      self.state_value = objective_function(self)

  def random_cube(self):
    numbers = list(range(1, self.x_size**3 + 1))
    random.shuffle(numbers)
    self.array = numbers

  def to_index(self, x, y, z):
    return x * (self.y_size * self.z_size) + y * (self.z_size) + z

  def insert_value(self, x, y, z, value):
    self.array[self.to_index(x, y, z)] = value

  def calculate_state_value(self):
    self.state_value = objective_function(self)

  def get_value(self, x, y, z):
    return self.array[self.to_index(x, y, z)]

  def delete_value(self, x, y, z):
    index = self.to_index(x, y, z)
    value = self.array[index]
```

```
        self.array[index] = None
        return value

    def print_cube(self):
        for x in range(self.x_size):
            print(f"Layer {x}:")
            for y in range(self.y_size):
                row = [self.array[self.to_index(x, y, z)] for z in range(self.z_size)]
                print(row)
            print()

    def copy(self):
        new_array = copy.deepcopy(self.array)
        new_cube = Cube(self.x_size, self.y_size, self.z_size, array=new_array)
        new_cube.state_value = self.state_value
        return new_cube


def check_rows(cube):
    size = cube.x_size
    arr = []
    for x in range(size):
        for y in range(size):
            row_sum = sum(cube.get_value(x, y, z) for z in range(size))
            arr.append(row_sum)
    return arr


def check_columns(cube):
    size = cube.x_size
    arr = []
    for x in range(size):
        for z in range(size):
            col_sum = sum(cube.get_value(x, y, z) for y in range(size))
            arr.append(col_sum)
    return arr


def check_pillars(cube):
    size = cube.x_size
    arr = []
    for y in range(size):
        for z in range(size):
            pillar_sum = sum(cube.get_value(x, y, z) for x in range(size))
```

```
            arr.append(pillar_sum)
    return arr


def check_space_diagonals(cube):
    size = cube.x_size
    diag1 = sum(cube.get_value(i, i, i) for i in range(size))
    diag2 = sum(cube.get_value(i, i, size - i - 1) for i in range(size))
    diag3 = sum(cube.get_value(i, size - i - 1, i) for i in range(size))
    diag4 = sum(cube.get_value(i, size - i - 1, size - i - 1) for i in range(size))
    return [diag1, diag2, diag3, diag4]


def check_plane_diagonals(cube):
    arr = []
    size = cube.x_size
    for x in range(size):
        diag1 = sum(cube.get_value(x, i, i) for i in range(size))
        diag2 = sum(cube.get_value(x, i, size - i - 1) for i in range(size))
        arr.append(diag1)
        arr.append(diag2)

    for y in range(size):
        diag1 = sum(cube.get_value(i, y, i) for i in range(size))
        diag2 = sum(cube.get_value(size - i - 1, y, i) for i in range(size))
        arr.append(diag1)
        arr.append(diag2)

    for z in range(size):
        diag1 = sum(cube.get_value(i, i, z) for i in range(size))
        diag2 = sum(cube.get_value(size - i - 1, i, z) for i in range(size))
        arr.append(diag1)
        arr.append(diag2)

    return arr


def objective_function_view(cube):
    arr = []
    arr += (
        check_rows(cube)
        + check_columns(cube)
        + check_pillars(cube)
        + check_plane_diagonals(cube)
```

```
    + check_space_diagonals(cube)
  )
  print(arr)



def generate_successor(curr_cube_arr):
  successors = []
  n = len(curr_cube_arr)
  for i in range(n):
    for j in range(i + 1, n):
      new_arr = curr_cube_arr[:]
      new_arr[i], new_arr[j] = new_arr[j], new_arr[i]
      successors.append(Cube(5, 5, 5, False, new_arr))
  return successors

def random_1d_array(n):
 numbers = list(range(1, n**3 + 1))
 random.shuffle(numbers)
 return numbers

def objective_function(cube):
  arr = []
  arr += (
    check_rows(cube)
    + check_columns(cube)
    + check_pillars(cube)
    + check_plane_diagonals(cube)
    + check_space_diagonals(cube)
  )
  return sum(1 for el in arr if el == cube.magic_number)
```

2.3. Implementasi Algoritma Local Search

2.2.1. Steepest Ascent Hill-climbing

Langkah-langkah pada algoritma ini :

1. Membuat sebuah cube random dan dihitung state valuenya, setiap cube dan state valuenya disimpan pada array cubes dan values. Dihitung juga banyak iterasi yang dilakukan
2. Melakukan loop dimulai dengan pembangkitan seluruh successor dari current state

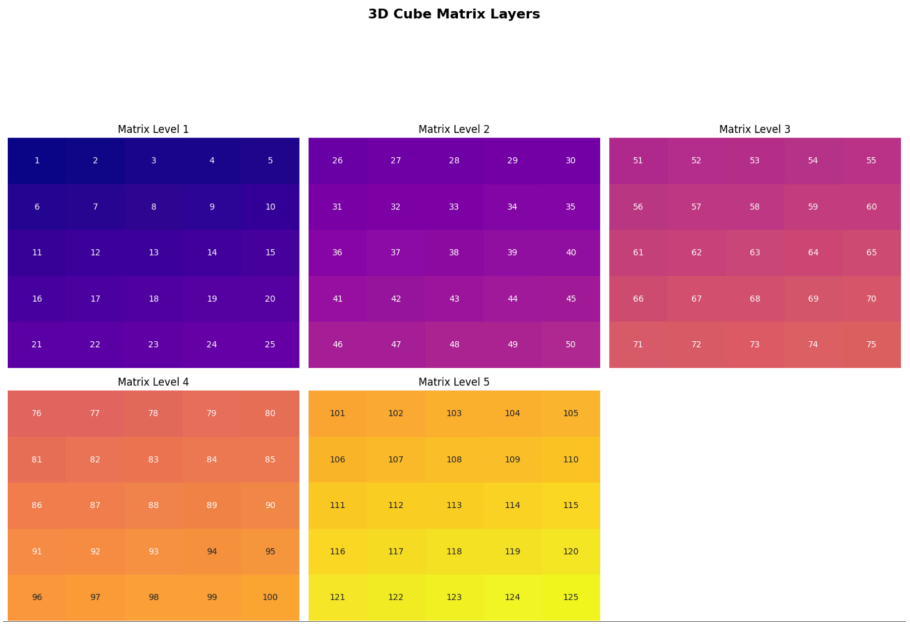
3. Menghitung seluruh state value dari successor dan mencari yang terbaik
4. Melakukan komparasi dari nilai neighbour terbaik saat ini dan nilai state sekarang. Jika lebih besar maka simpan neighbour dan valuenya ke current state, Jika tidak maka return current state

Source Code

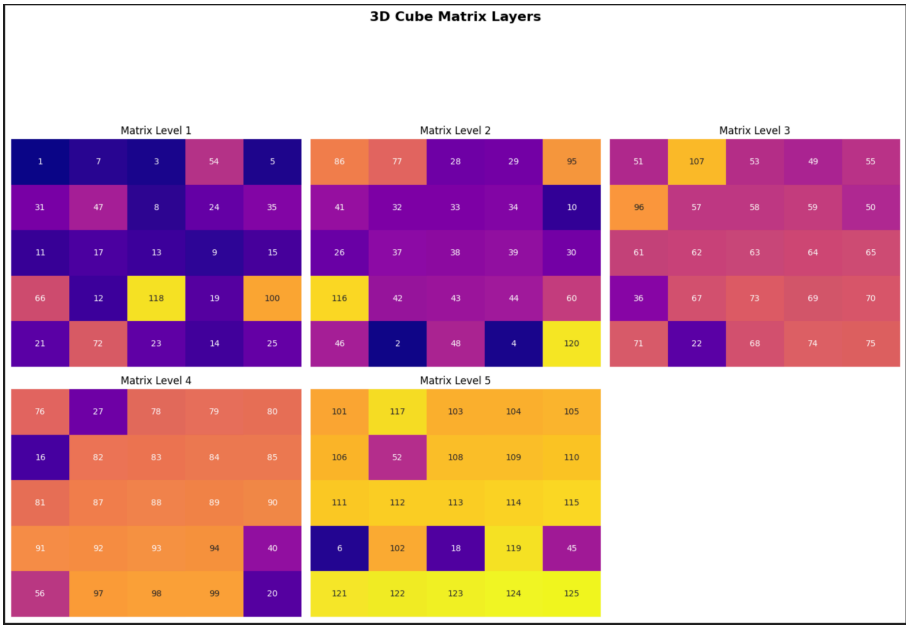
```
def SteepestAscentHillClimbingCube(init_cube):
    current_cube = Cube(5, 5, 5, False, init_cube)
    current_value = current_cube.state_value
    cubes = [current_cube]
    values = [current_value]
    count_iter = 0

    while True:
        print("Current val: ", current_value)
        count_iter += 1
        neighbors = generate_successor(current_cube.array)
        neighbors_value = [cube.state_value for cube in neighbors]
        best_neighbor = neighbors[np.argmax(neighbors_value)]
        best_value = max(neighbors_value)

        if best_value > current_value:
            current_cube = best_neighbor
            current_value = best_value
        else:
            return cubes, values, count_iter

        cubes.append(current_cube)
        values.append(current_value)
```

2.2.1.1. Percobaan 1

State Awal

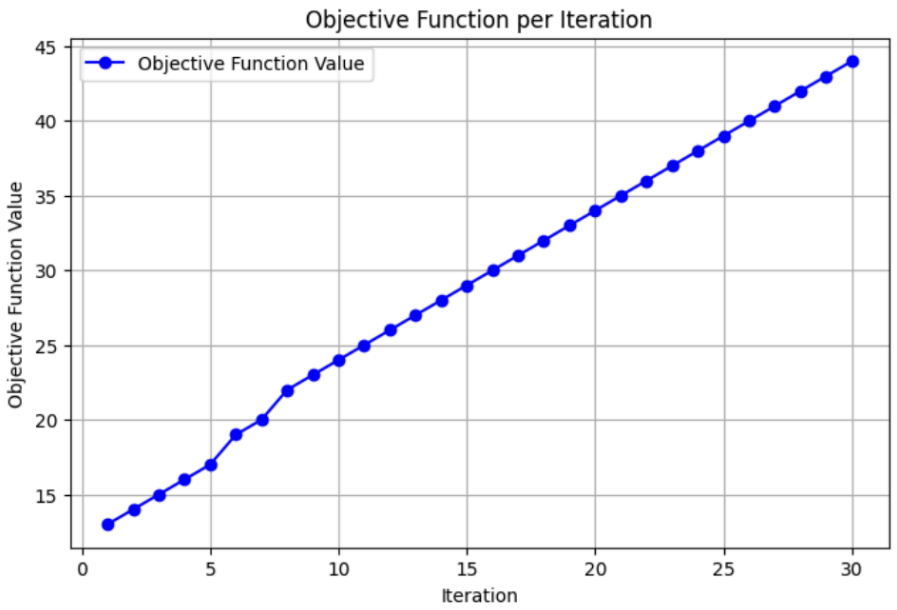


State Akhir



Nilai Objective Function: 44

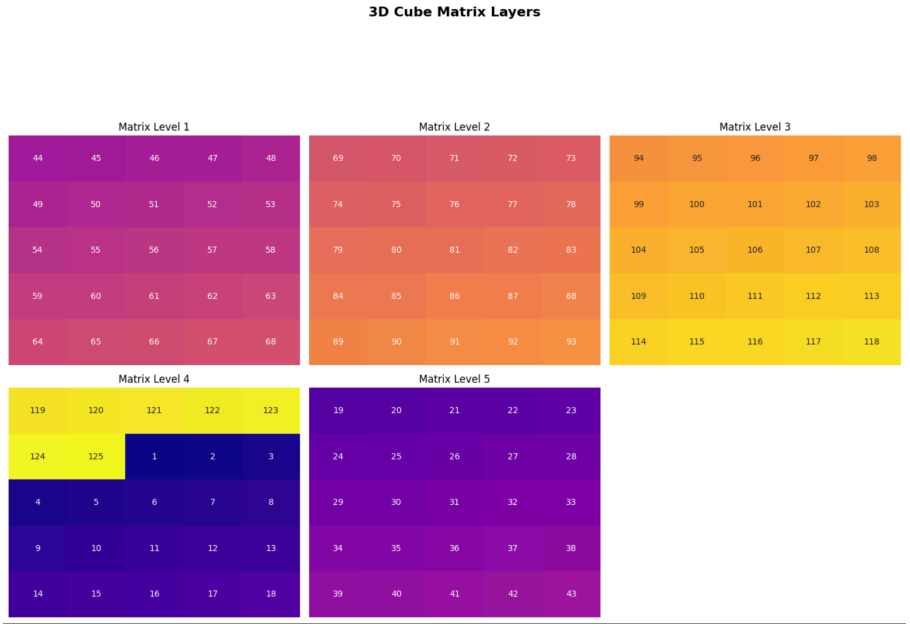
Plot Nilai Objective Function:



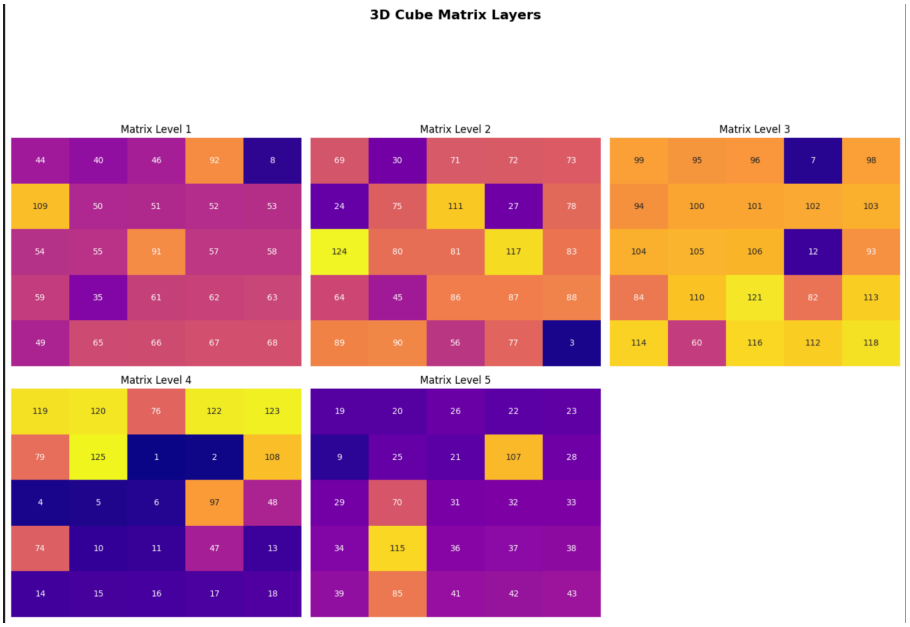
Banyak Iterasi: 30

2.2.1.2. Percobaan 2

State Awal

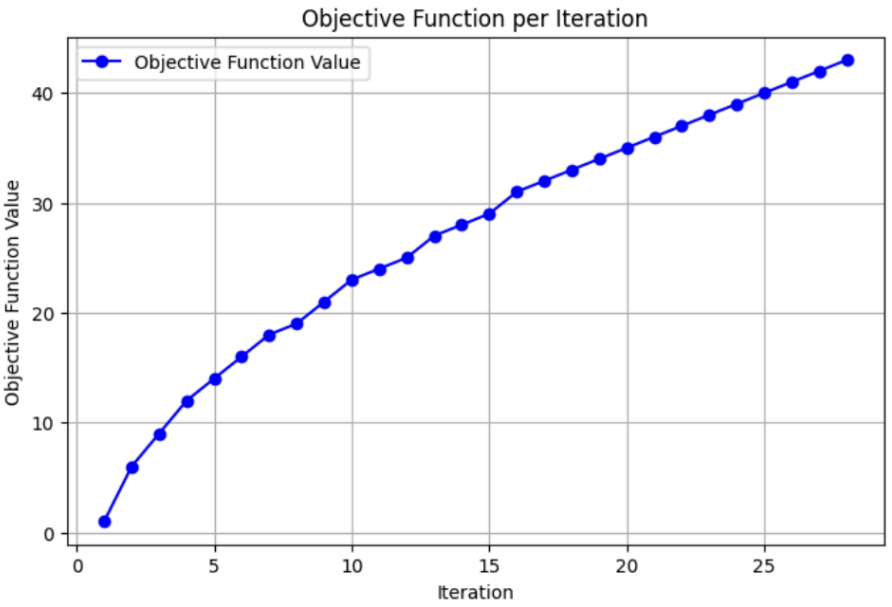


State Akhir



Nilai Objective Function: 43

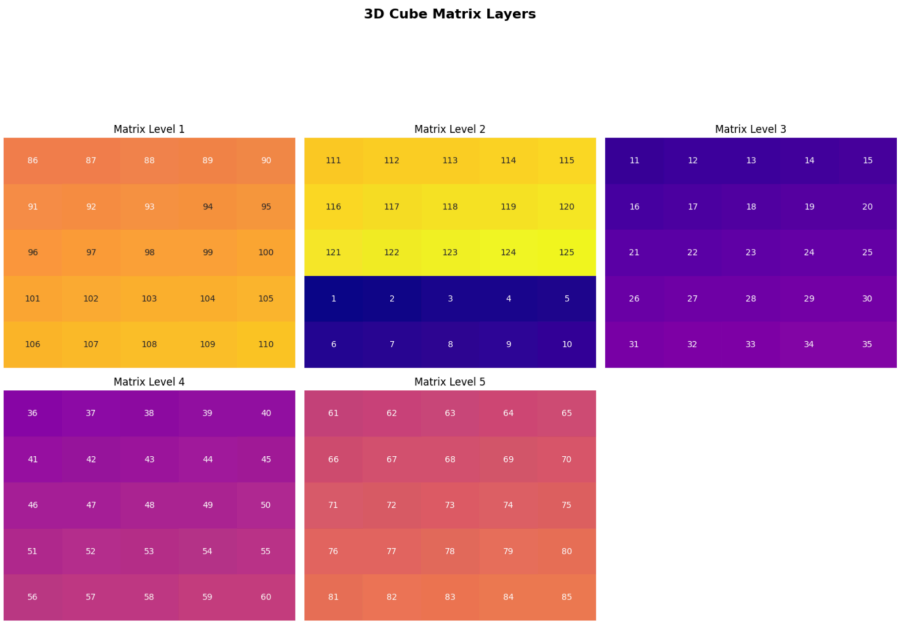
Plot Nilai Objective Function:



Banyak Iterasi: 28

2.2.1.3. Percobaan 3

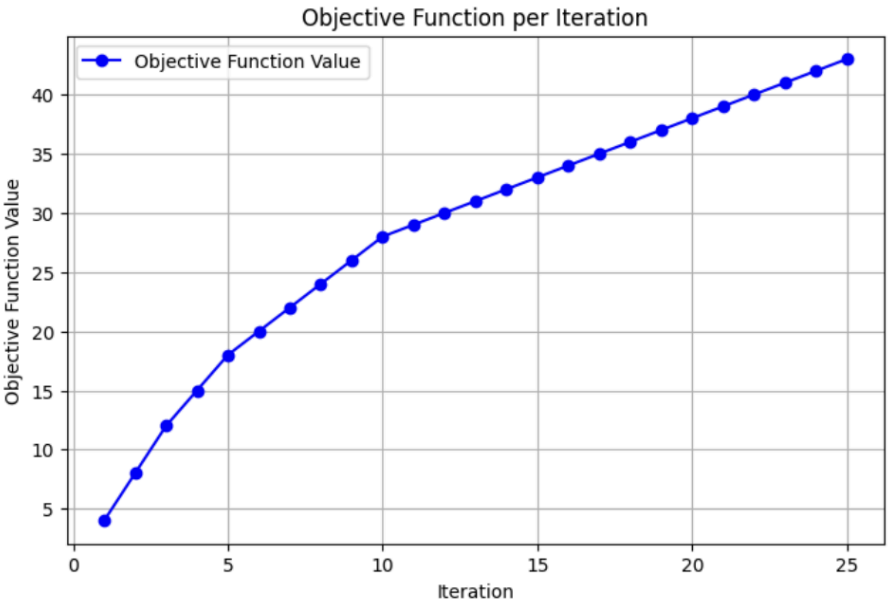
State Awal



State Akhir

Nilai Objective Function: 43

Plot Nilai Objective Function:



Banyak Iterasi: 25

2.2.2. Hill-climbing with Sideways Move

Langkah-langkah pada algoritma ini adalah

1. Membuat sebuah cube random dan dihitung state valuenya, setiap cube dan state valuenya disimpan pada array cubes dan values. Dihitung juga banyak iterasi yang dilakukan dan juga inisiasi dari sideways count
2. Melakukan loop dimulai dengan pembangkitan seluruh successor dari current state
3. Menghitung seluruh state value dari successor dan mencari yang terbaik
4. Melakukan komparasi dari nilai neighbour terbaik saat ini dan nilai state sekarang. Jika lebih besar sama dengan maka simpan neighbour dan valuenya ke current state. Jika nilai sama dengan sebelumnya maka pindah ke state tersebut dan increment sideway count. Jika state value neighbour lebih kecil atau sideways count sudah melebihi maksimum maka return current state

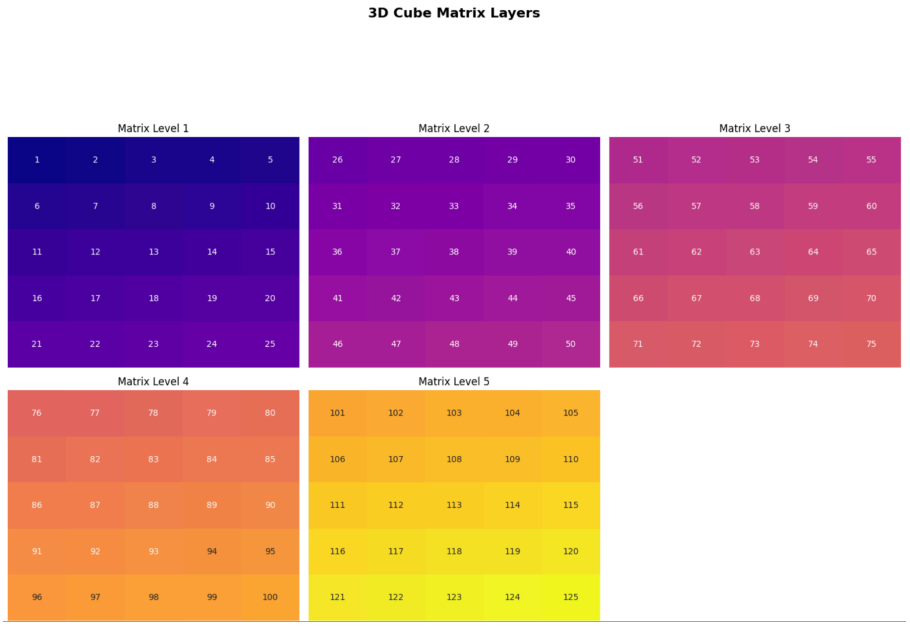
Source Code

```
def HillClimbingWithSidewaysMoveCube(init_cube, max_sideways):
    current_cube = Cube(5, 5, 5, False, init_cube)
    current_value = current_cube.state_value
    neighbor_value = current_value

    count_iter = 0
    sideways_count = 0
```

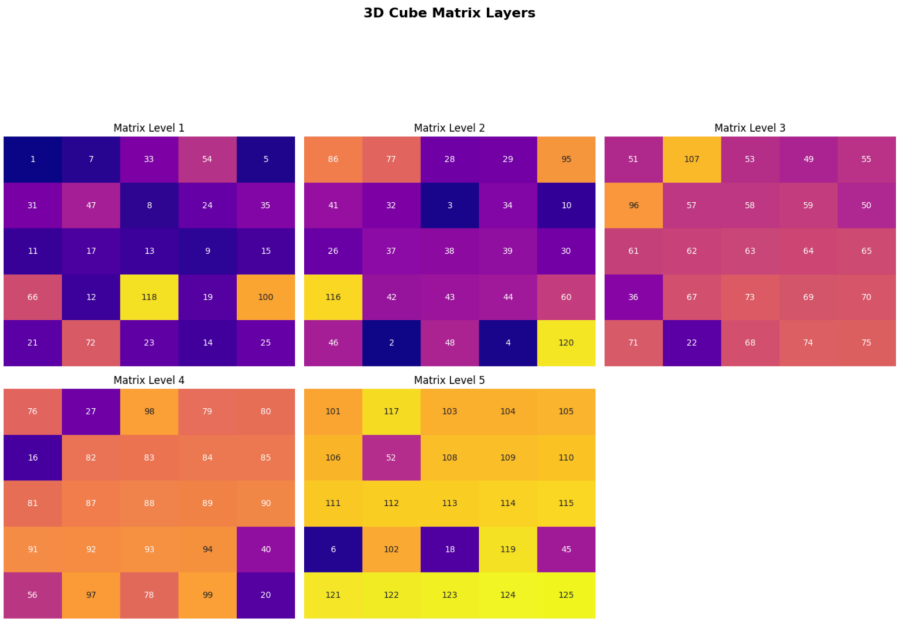
```
    cubes = [current_cube]
    values = [current_value]

    while neighbor_value >= current_value and sideways_count <
max_sideways:
        count_iter += 1
        neighbors = generate_successor(current_cube.array)
        neighbors_value = [cube.state_value for cube in neighbors]
        best_neighbor = neighbors[np.argmax(neighbors_value)]
        neighbor_value = max(neighbors_value)

        if neighbor_value >= current_value:
            if neighbor_value > current_value:
                sideways_count = 0
            elif neighbor_value == current_value:
                sideways_count += 1
            current_cube = best_neighbor
            current_value = neighbor_value
        print("Current val:", current_value, "| Neighbor Val:",
neighbor_value, "| Sideways Count:", sideways_count)

    return cubes, values, count_iter
```

2.2.2.1. Percobaan 1

State Awal

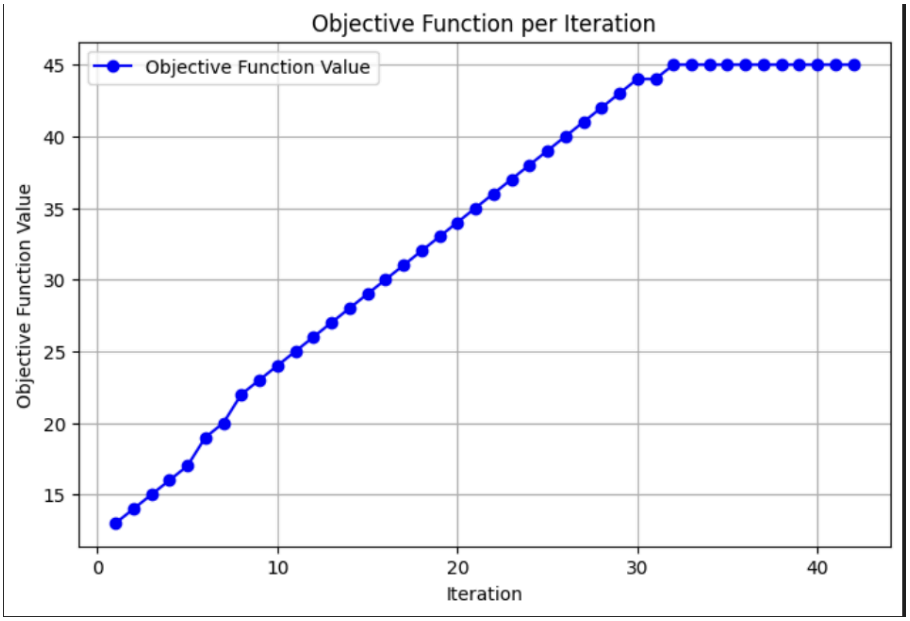


State Akhir



Nilai Objective Function: 45

Plot Nilai Objective Function:

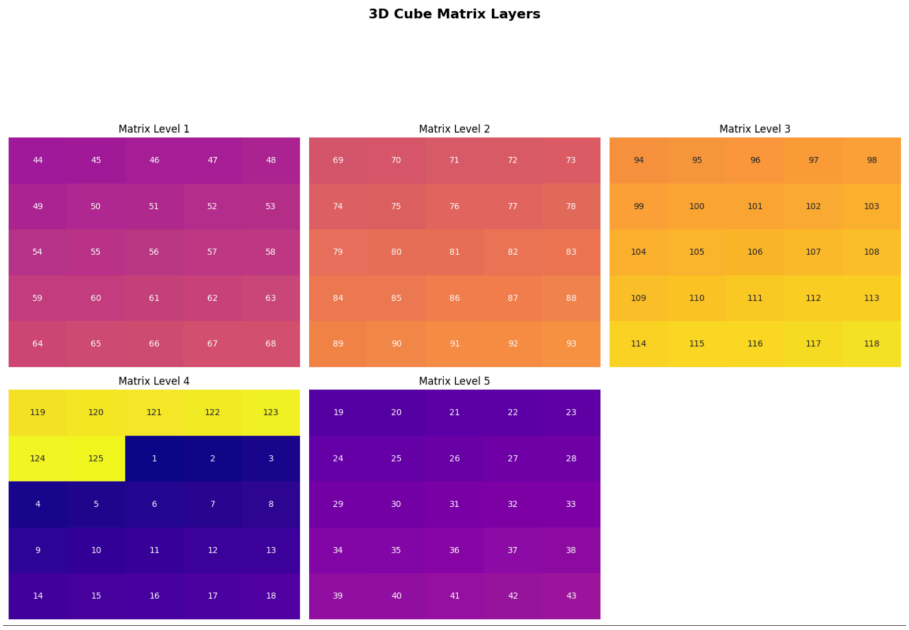


Banyak Iterasi: 41

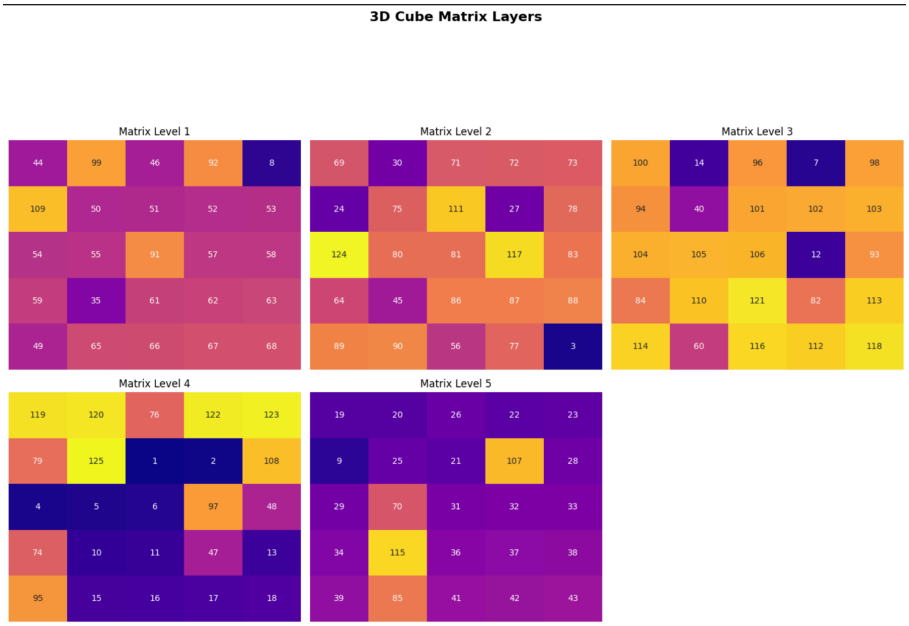
Max Sideways : 10

2.2.2.2. Percobaan 2

State Awal

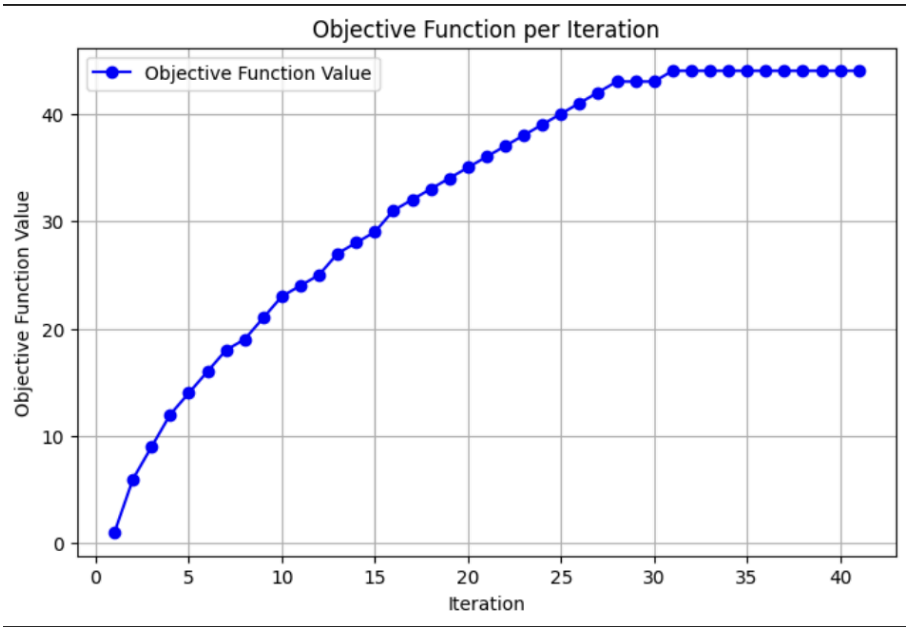


State Akhir



Nilai Objective Function: 44

Plot Nilai Objective Function

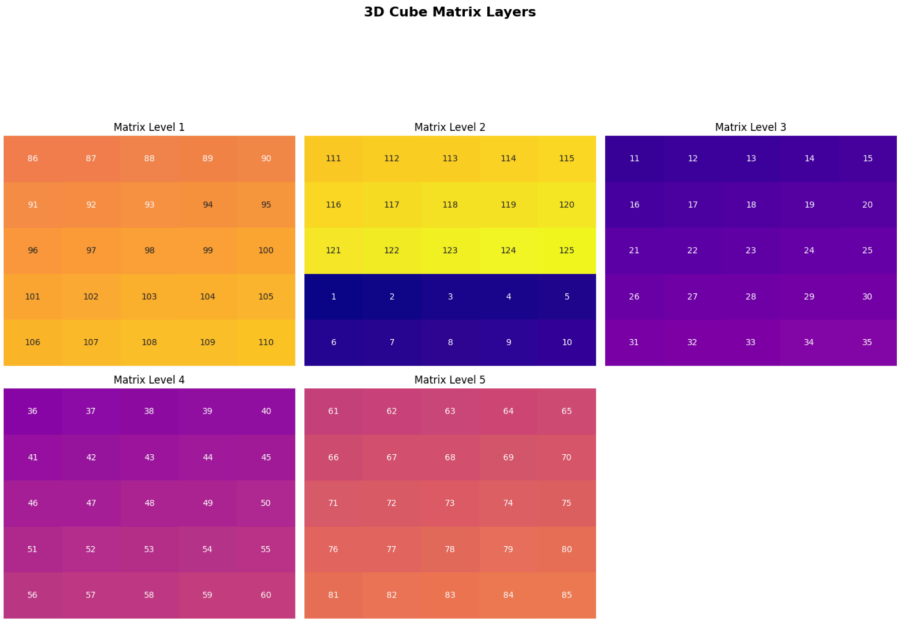


Banyak Iterasi: 40

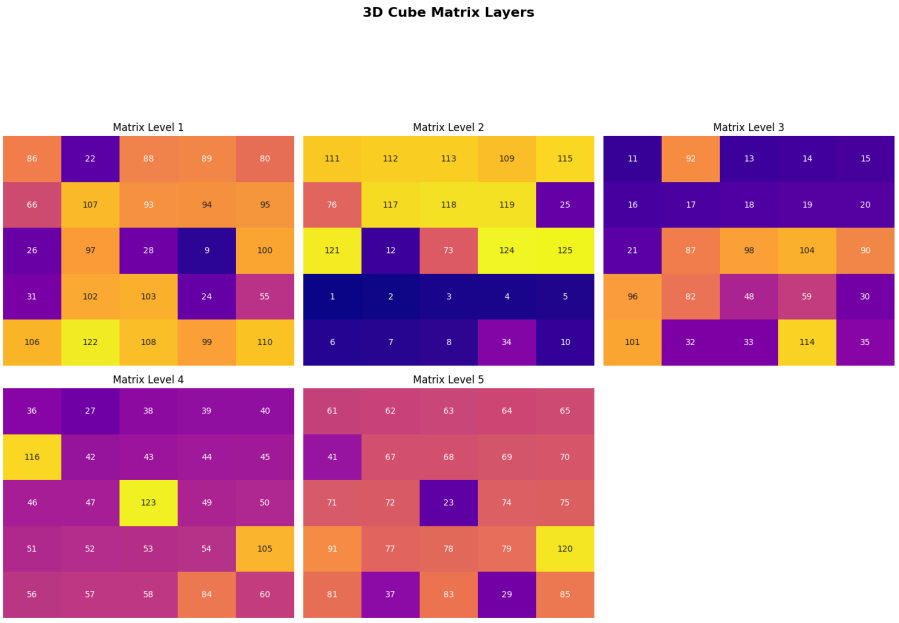
Max Sideways : 10

2.2.2.3. Percobaan 3

State Awal

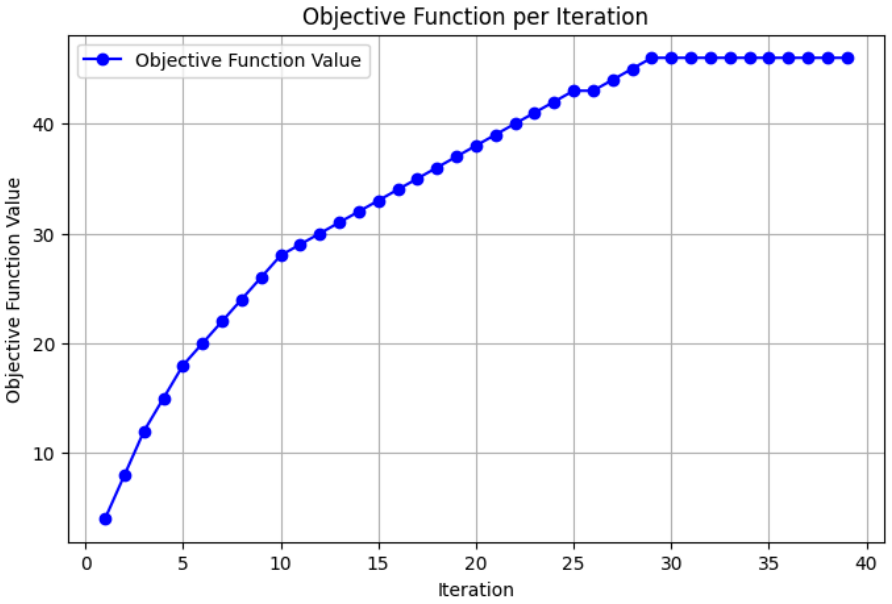


State Akhir



Nilai Objective Function: 46

Plot Nilai Objective Function:



Banyak Iterasi: 38

Max Sideways : 10

2.2.3. Random Restart Hill-climbing

Langkah-langkah pada algoritma ini:

1. Melakukan local search inisial menggunakan steepest ascent hill climbing untuk menentukan nilai awal
2. Kemudian lakukan restart sebanyak max restart dimana pada setiap iterasi membangkitkan cube baru untuk dilakukan steepest ascent hill climbing
3. Setelah max restart fungsi ini mengembalikan hasil local search per restart, value akhir setiap restart, iterasi per restart, dan jumlah restart yang dilakukan

Source Code

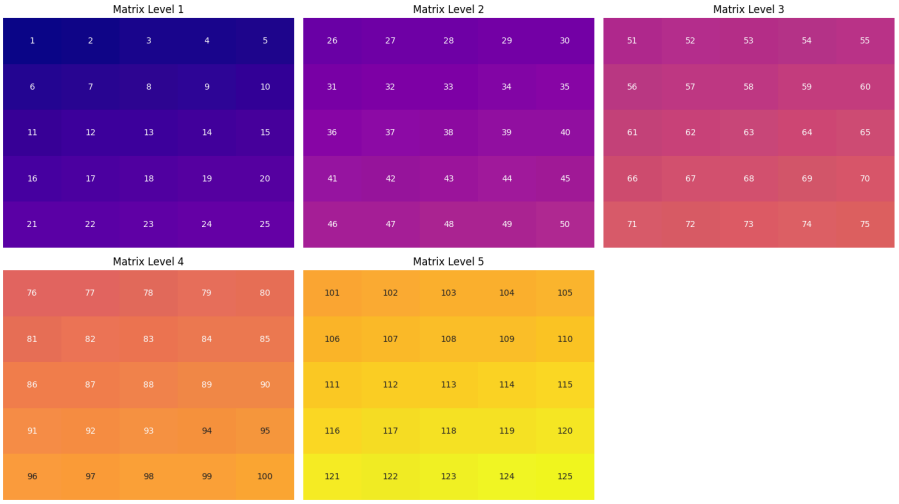
```
def RandomRestartHillClimbingCube(init_cube, max_restart):

    restart = 0
    cubes, values, count_iter =
SteepestAscentHillClimbingCube(init_cube)

    current_cube = cubes[0]
    current_value = current_cube.state_value

    cubes_per_restart = [cubes]
    values_per_restart = [values]
    iteration_per_restart = [count_iter]

    print(f"Best value at restart {restart} = {values[-1]}")
    if values[-1] < current_value:
        current_value = values[-1]
        current_cube = cubes[-1]

    while restart < max_restart:
        restart += 1
        new_cube = random_1d_array(5) # Generate a new random Cube
instance
        cubes, values, count_iter =
SteepestAscentHillClimbingCube(new_cube)

        cubes_per_restart.append(cubes)
        values_per_restart.append(values)
        iteration_per_restart.append(count_iter)

        print(f"Best value at restart {restart} = {values[-1]}")
        if values[-1] < current_value:
            current_value = values[-1]
            current_cube = cubes[-1]
    return cubes_per_restart, values_per_restart,
iteration_per_restart, restart
```

2.2.3.1. Percobaan 1

State Awal

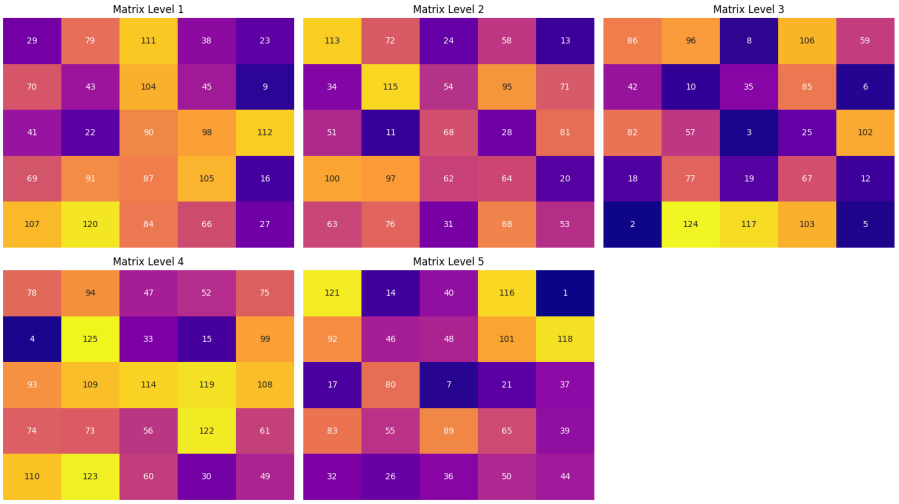
Initial state restart - 0

Fitness Value: 13



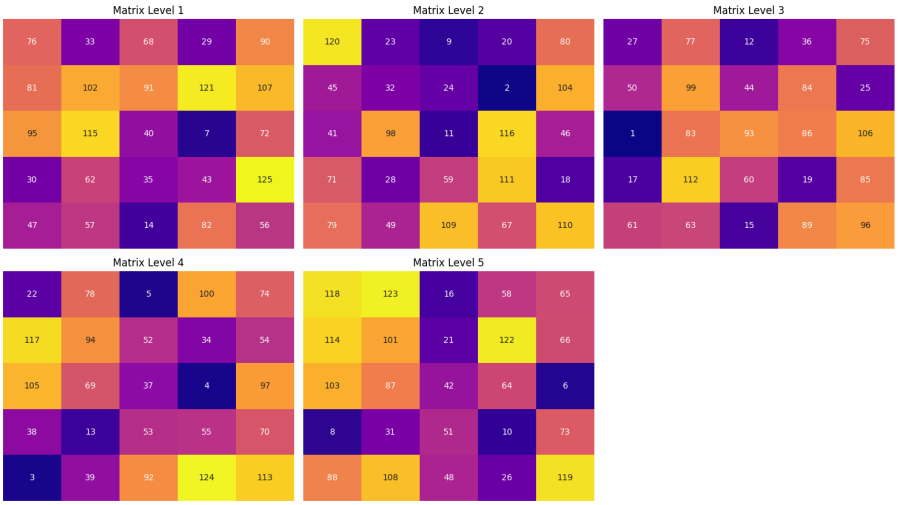
Initial state restart - 1

Fitness Value: 0



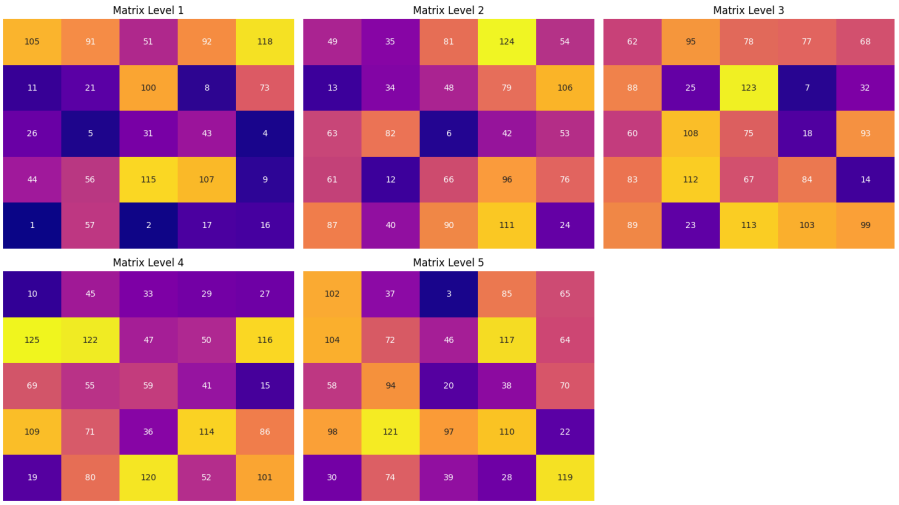
Initial state restart - 2

Fitness Value: 0



Initial state restart - 3

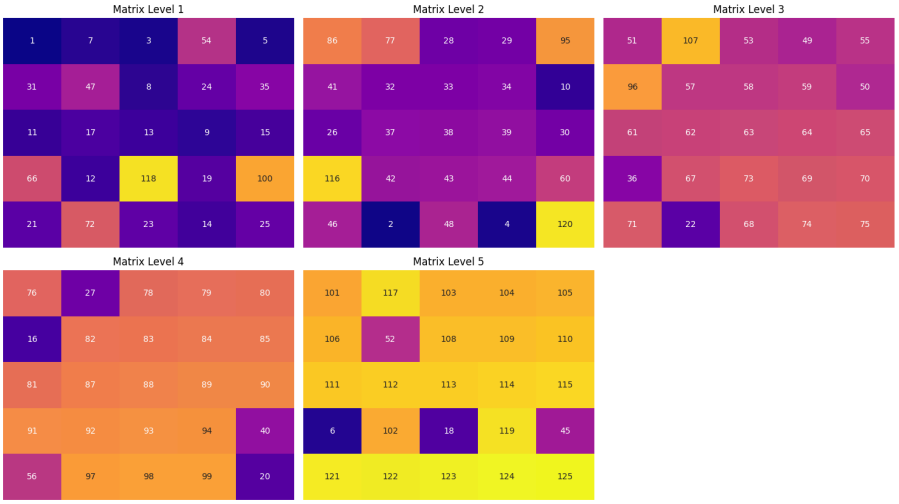
Fitness Value: 0



State Akhir

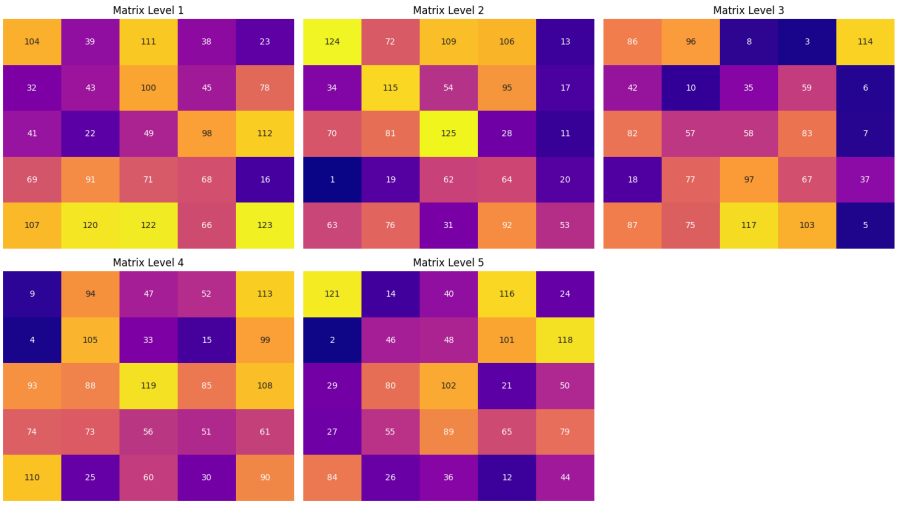
Final state restart - 0

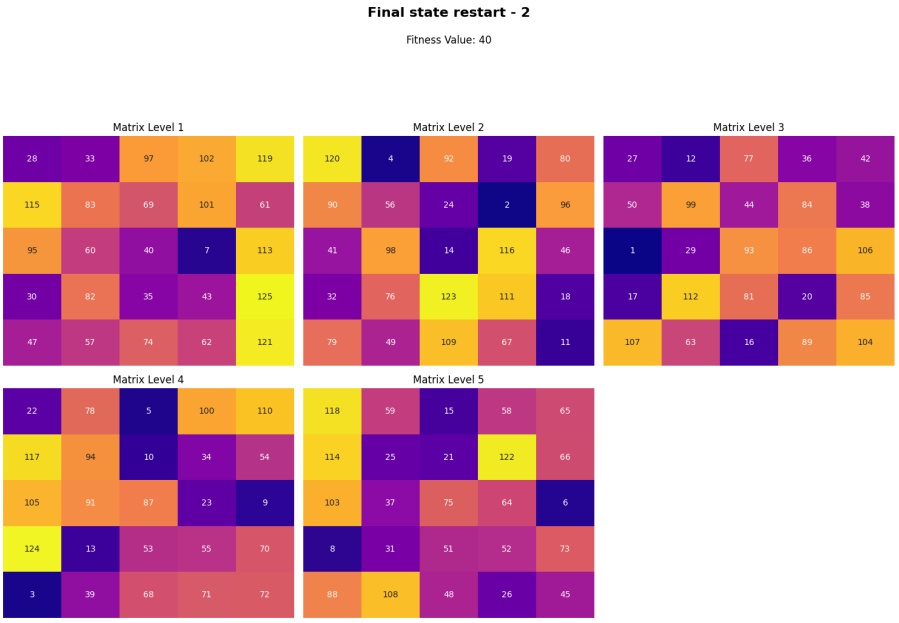
Fitness Value: 44

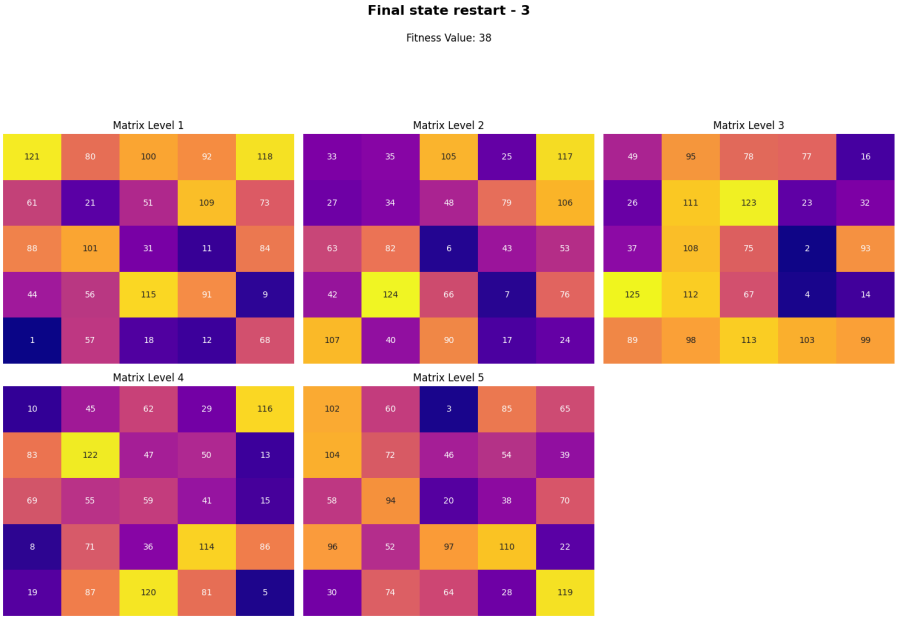


Final state restart - 1

Fitness Value: 41

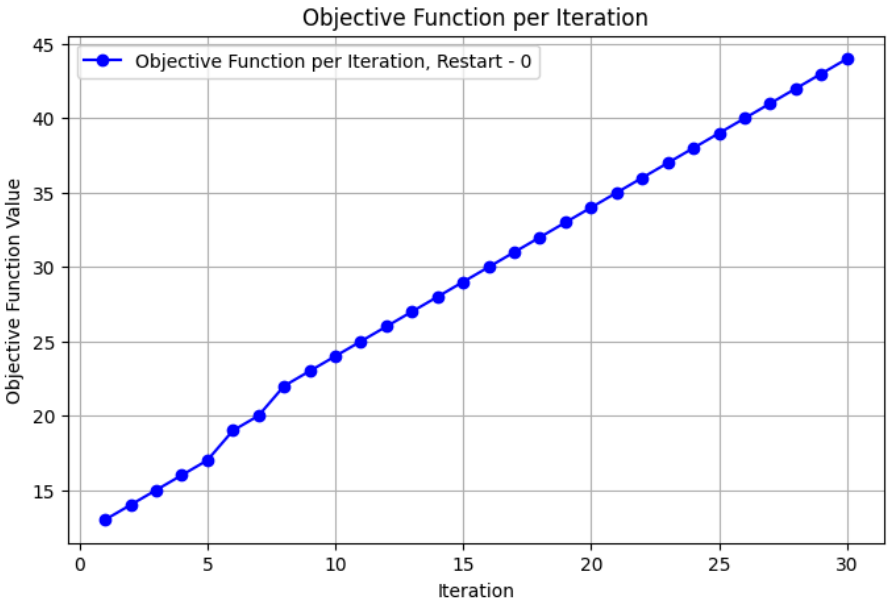


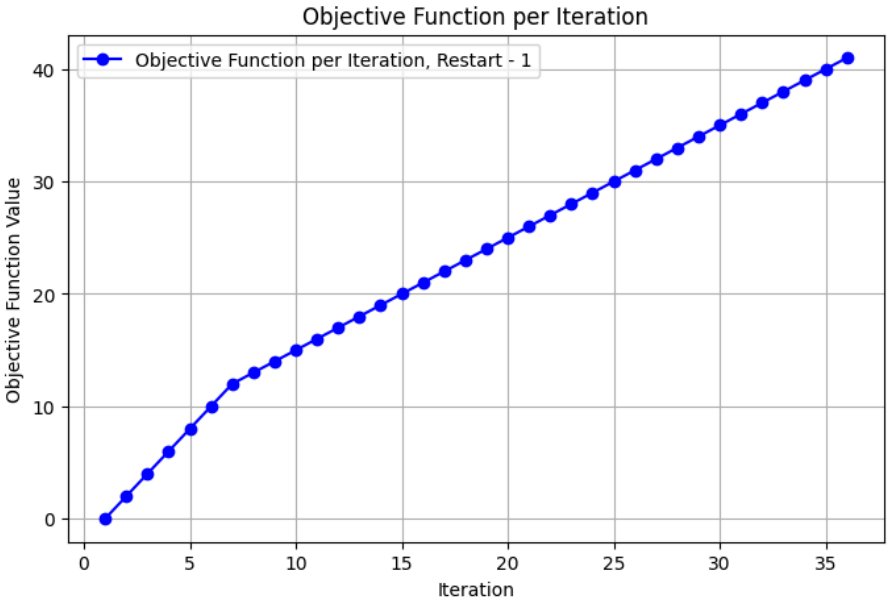


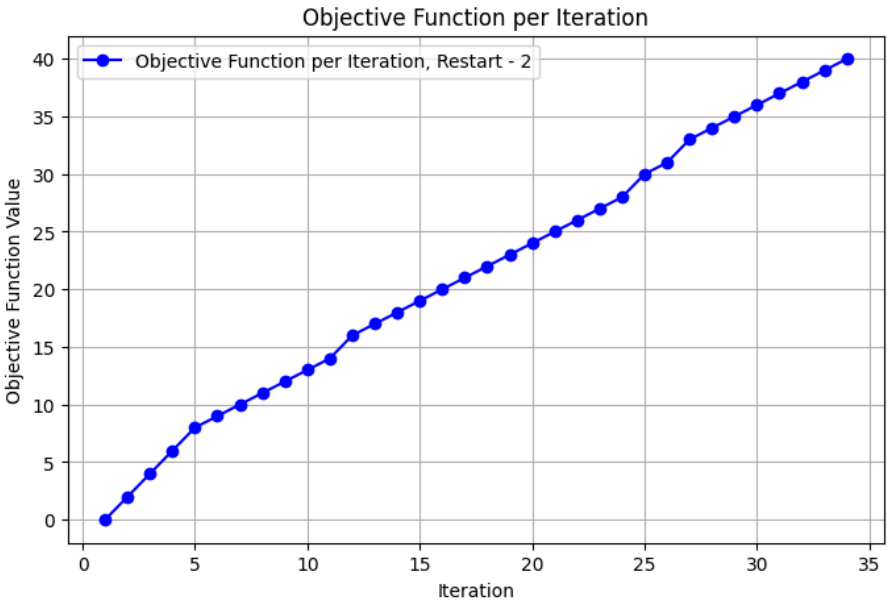


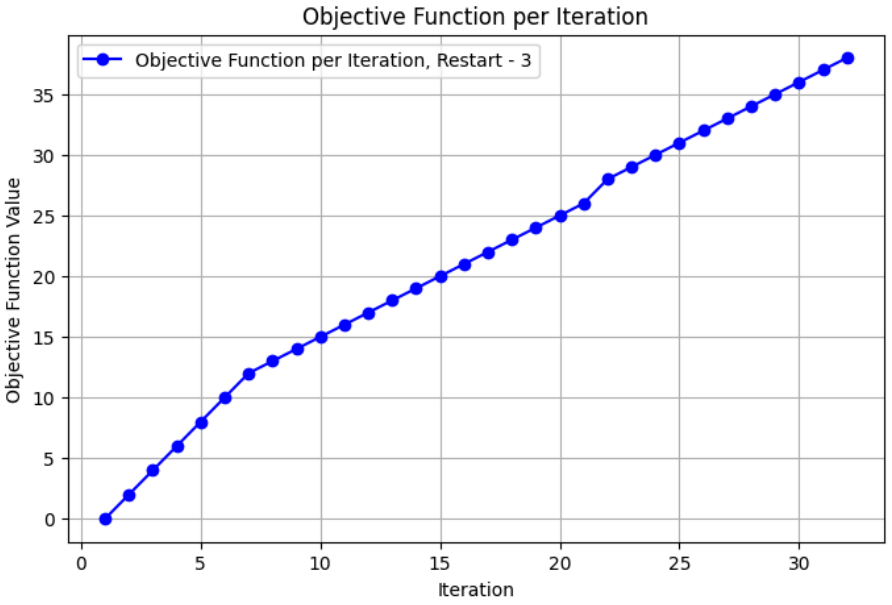
Nilai Objective Function Terbesar: 44

Plot Nilai Objective Function:


Objective Function per Iteration
Objective Function per Iteration, Restart - 0
Objective Function Value
Iteration
45
40
35
30
25
20
15
0
5
10
15
20
25
30

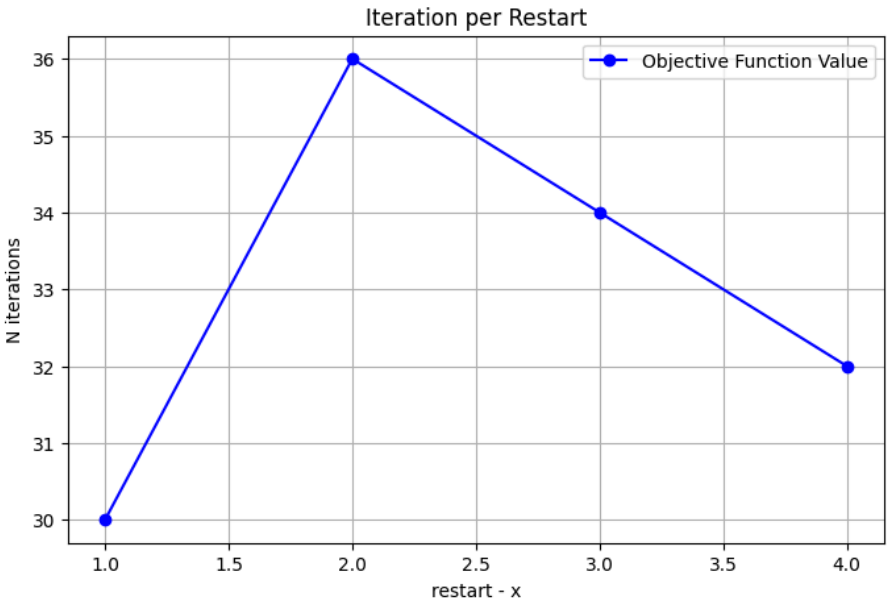

Objective Function per Iteration
Objective Function per Iteration, Restart - 1
Objective Function Value
Iteration
40
30
20
10
0
0
5
10
15
20
25
30
35





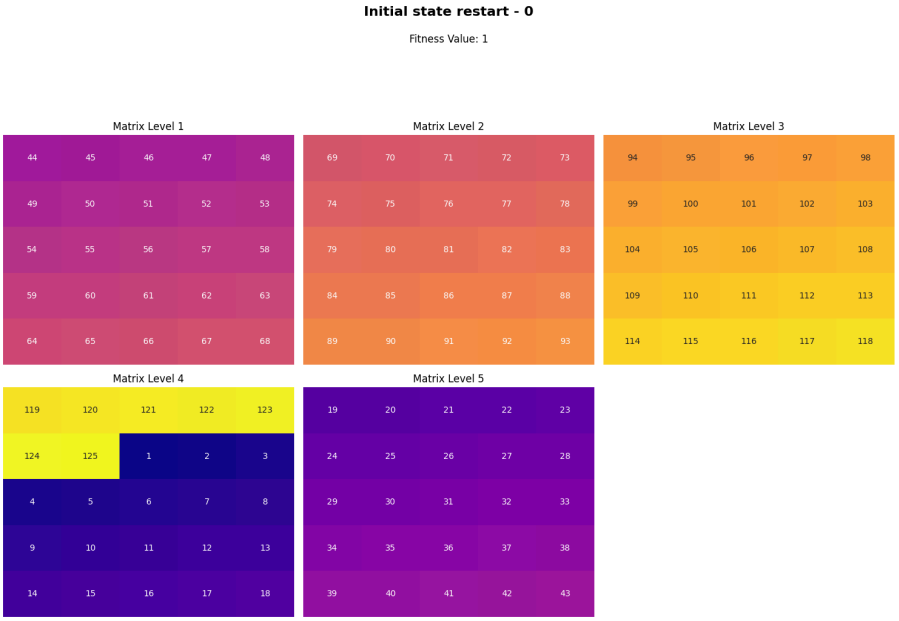
Banyak Restart: 3

Banyak Iterasi per Restart:



2.2.3.2. Percobaan 2

State Awal



Initial state restart - 1

Fitness Value: 0

Matrix Level 1

63	22	91	47	87
32	54	118	109	59
123	55	82	74	70
110	52	56	81	90
94	86	9	19	69

Matrix Level 2

77	120	103	8	41
28	20	102	57	107
40	15	106	34	2
7	23	125	111	119
53	117	99	18	4

Matrix Level 3

122	108	5	27	49
98	10	62	88	101
6	84	97	104	31
13	76	115	26	17
80	58	51	38	44

Matrix Level 4

100	72	93	121	116
50	12	71	66	105
92	48	124	16	3
83	30	113	89	68
1	42	35	96	60

Matrix Level 5

78	25	65	24	45
37	112	67	64	21
75	85	79	61	33
29	11	73	95	43
39	114	14	46	36

Initial state restart - 2

Fitness Value: 3

Matrix Level 1

121	59	20	101	46
86	97	4	114	53
54	109	116	73	75
32	33	61	17	52
124	96	12	120	7

Matrix Level 2

25	108	47	24	29
70	39	22	44	14
125	72	41	119	99
106	85	28	23	117
56	11	88	64	92

Matrix Level 3

37	10	91	76	3
30	43	8	95	78
42	45	94	21	6
51	77	60	58	49
71	18	80	67	84

Matrix Level 4

62	15	27	50	98
118	13	90	74	55
89	40	2	68	35
66	31	36	1	102
69	93	122	16	87

Matrix Level 5

103	123	38	63	79
65	110	105	107	115
113	111	26	9	83
34	112	104	48	100
5	81	19	82	57

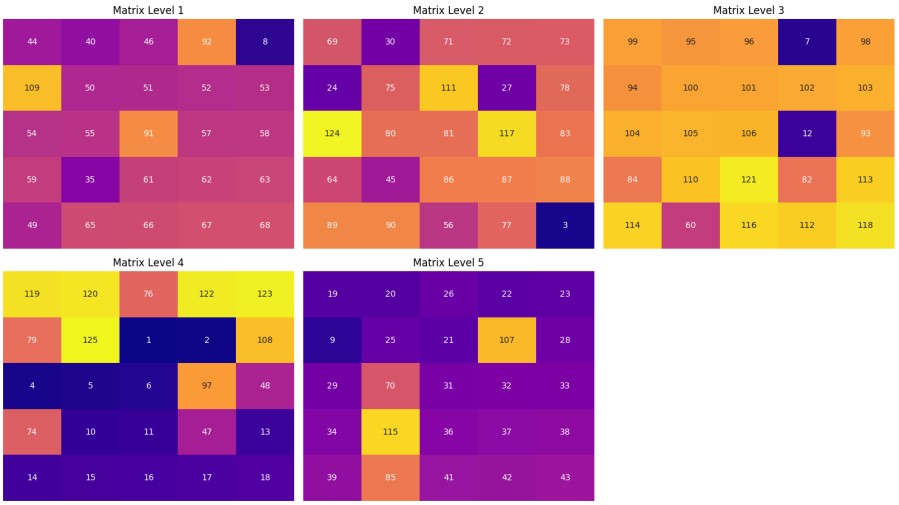
Initial state restart - 3

Fitness Value: 1

State Akhir

Final state restart - 0

Fitness Value: 43



Final state restart - 1

Fitness Value: 39

Matrix Level 1

63	22	91	47	92
50	20	77	109	59
107	55	82	100	5
11	119	56	81	90
84	99	9	10	69

Matrix Level 2

118	88	1	95	13
28	54	71	57	105
40	33	32	34	2
76	23	125	111	52
53	117	123	18	4

Matrix Level 3

46	108	35	27	49
98	19	17	120	101
6	94	97	104	14
41	7	115	26	62
124	58	51	38	44

Matrix Level 4

74	72	93	122	116
102	12	106	66	29
87	48	80	16	42
83	30	113	89	68
103	3	70	96	60

Matrix Level 5

78	25	65	24	45
37	114	67	64	21
75	85	79	61	15
86	110	73	8	43
39	112	31	121	36

Final state restart - 2

Fitness Value: 35

Matrix Level 1

124	59	112	102	46
32	6	119	105	53
61	121	116	66	64
120	33	87	23	52
114	96	12	86	7

Matrix Level 2

25	108	47	24	89
70	39	54	44	14
125	72	41	4	99
106	85	28	17	13
56	11	82	74	92

Matrix Level 3

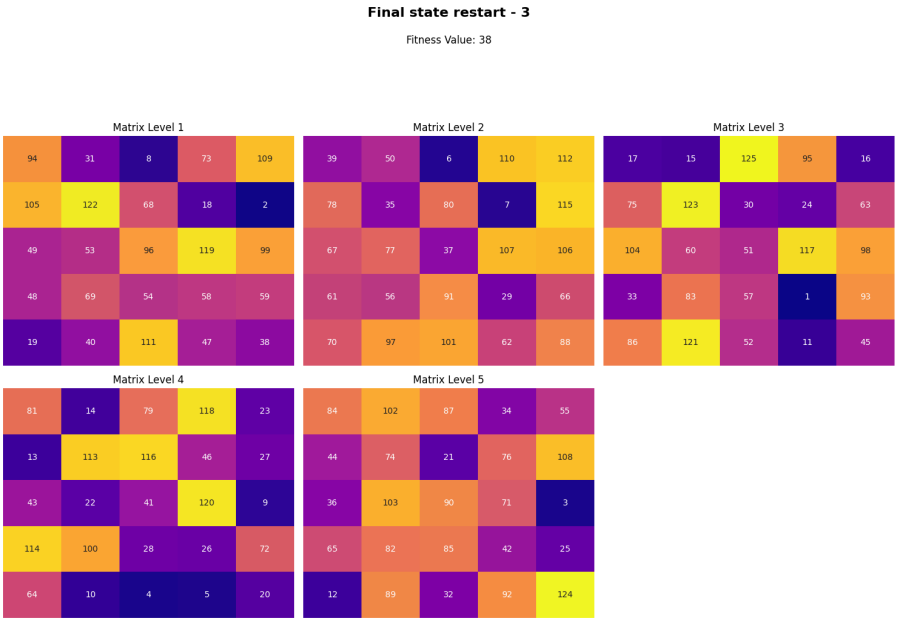
37	10	38	76	3
30	43	8	93	78
42	45	94	21	97
51	77	60	58	100
71	18	80	67	83

Matrix Level 4

26	15	27	50	98
118	117	90	75	55
29	40	2	68	35
73	34	36	1	101
69	109	122	16	22

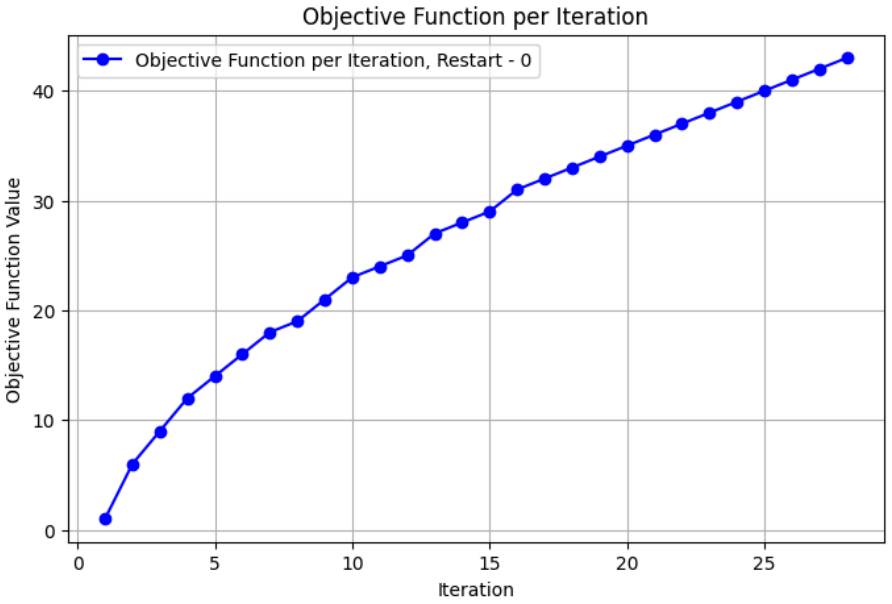
Matrix Level 5

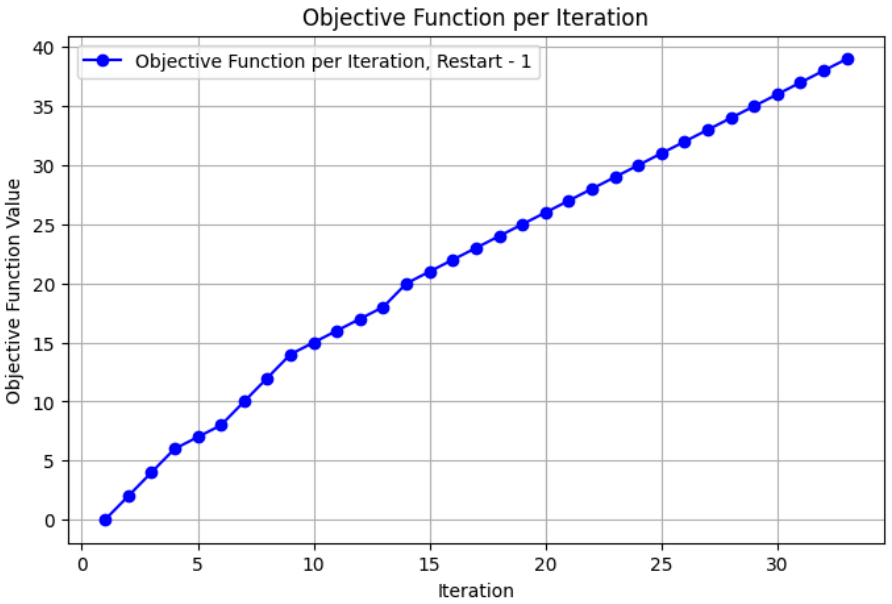
103	123	91	63	79
65	110	95	107	115
113	111	62	9	20
31	84	104	48	49
5	81	19	88	57

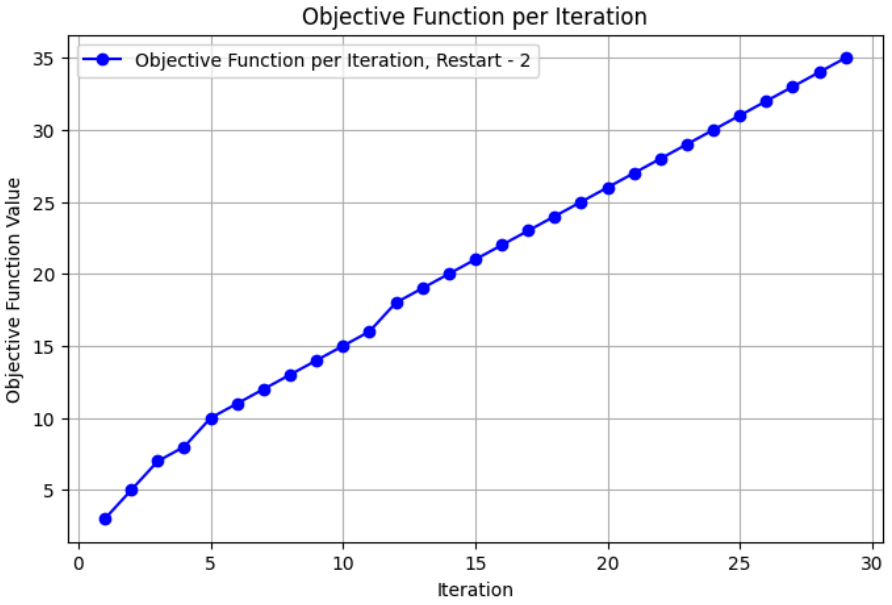


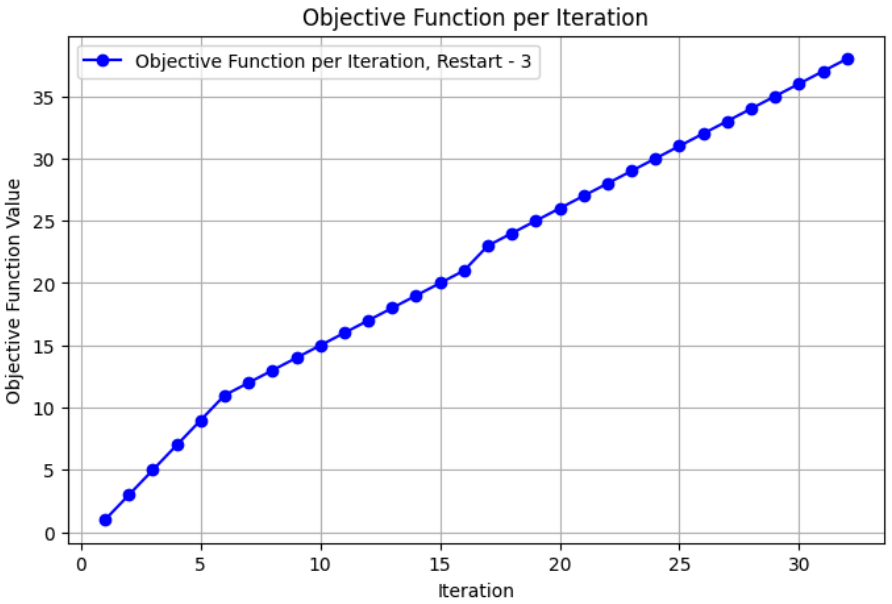
Nilai Objective Function Terbesar: 43

Plot Nilai Objective Function:



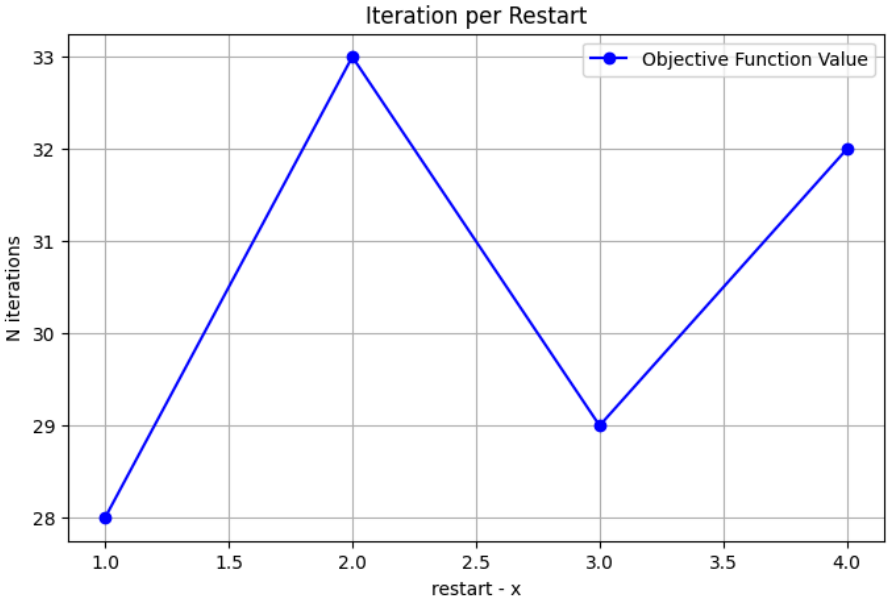

Objective Function per Iteration
Objective Function per Iteration, Restart - 1
Objective Function Value
Iteration
0
5
10
15
20
25
30
35
40


Objective Function per Iteration
Objective Function per Iteration, Restart - 2
Objective Function Value
Iteration
5
10
15
20
25
30
35



Banyak Restart: 3

Banyak Iterasi per Restart:

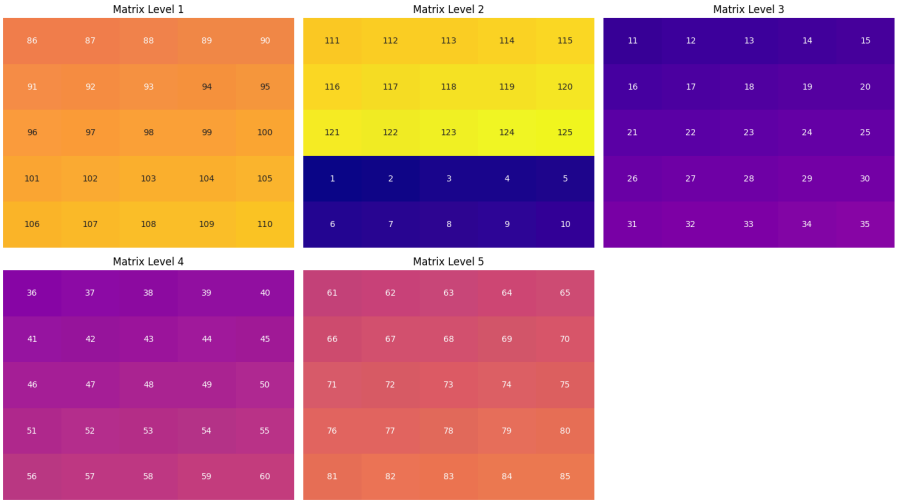


2.2.3.3. Percobaan 3

State Awal

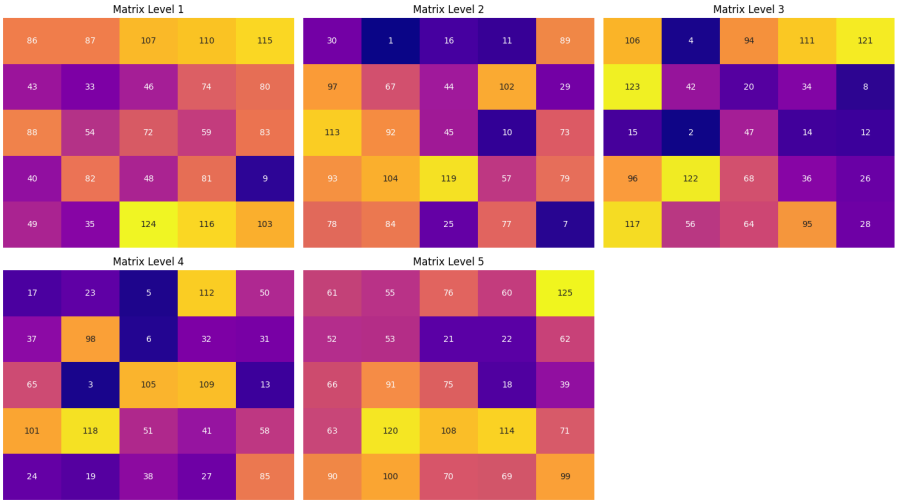
Initial state restart - 0

Fitness Value: 4



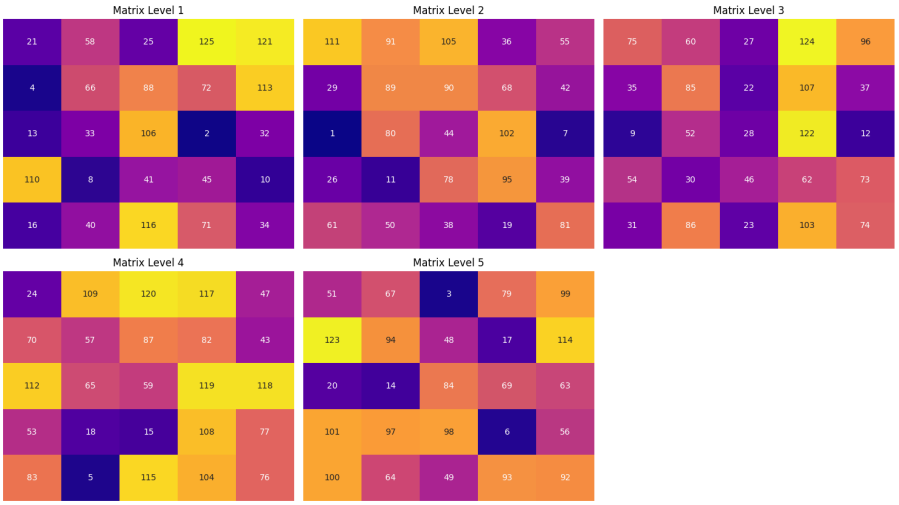
Initial state restart - 1

Fitness Value: 1



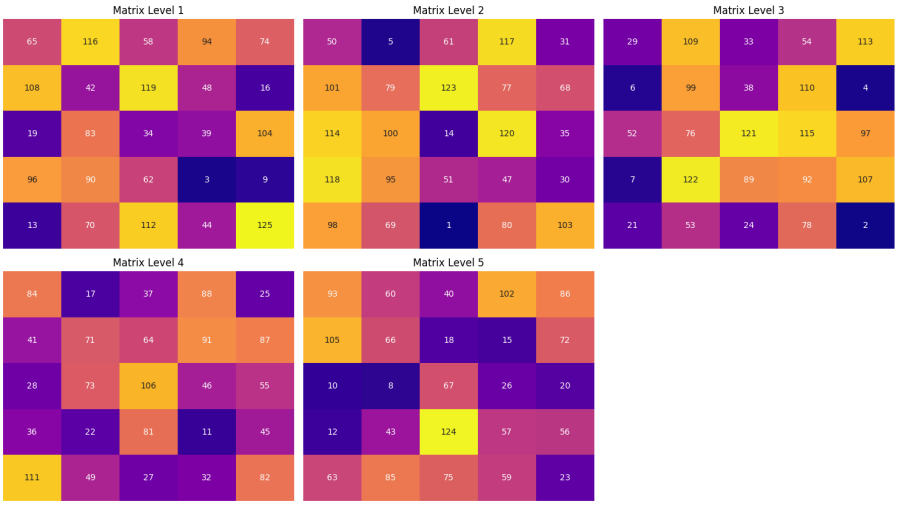
Initial state restart - 2

Fitness Value: 1



Initial state restart - 3

Fitness Value: 2



State Akhir

Initial state restart - 0

Fitness Value: 43

Matrix Level 1

86	87	88	89	80
66	92	93	94	95
26	97	28	9	100
31	102	103	24	55
106	107	108	99	110

Matrix Level 2

111	112	113	109	115
76	117	118	119	25
121	122	73	124	125
1	2	3	4	5
6	7	8	34	10

Matrix Level 3

11	12	13	14	15
16	17	18	19	20
21	22	98	104	90
96	82	48	59	30
101	32	33	114	35

Matrix Level 4

36	27	38	39	40
116	42	43	44	45
46	47	123	49	50
51	52	53	54	105
56	57	58	84	60

Matrix Level 5

61	62	63	64	65
41	67	68	69	70
71	72	23	74	75
91	77	78	79	120
81	37	83	29	85

Initial state restart - 1

Fitness Value: 39

Matrix Level 1

35	28	76	89	117
43	65	53	74	80
88	54	31	59	83
40	82	48	81	57
109	86	107	44	103

Matrix Level 2

96	8	16	11	110
60	67	10	102	29
15	49	45	116	90
4	104	119	9	79
19	87	125	77	7

Matrix Level 3

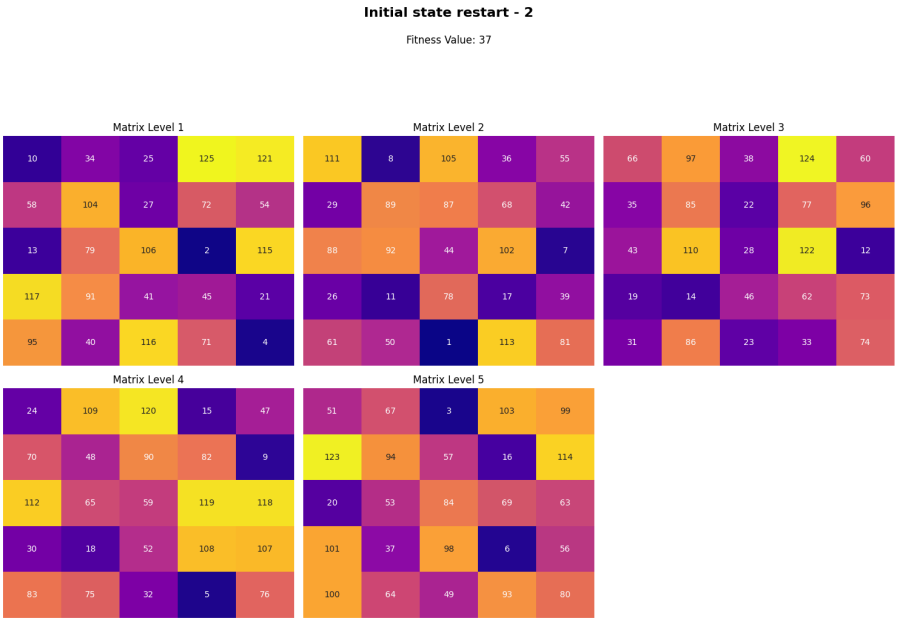
106	93	94	6	13
123	42	111	18	21
113	2	47	14	12
30	122	68	36	26
115	56	64	98	84

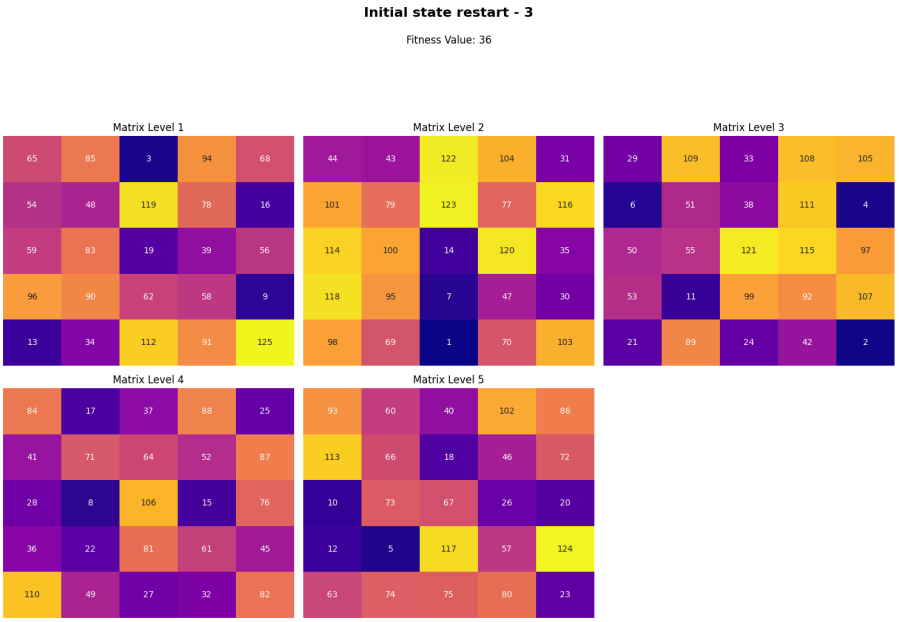
Matrix Level 4

17	23	5	112	50
37	95	20	32	1
33	3	91	92	121
38	118	51	41	58
24	78	101	27	85

Matrix Level 5

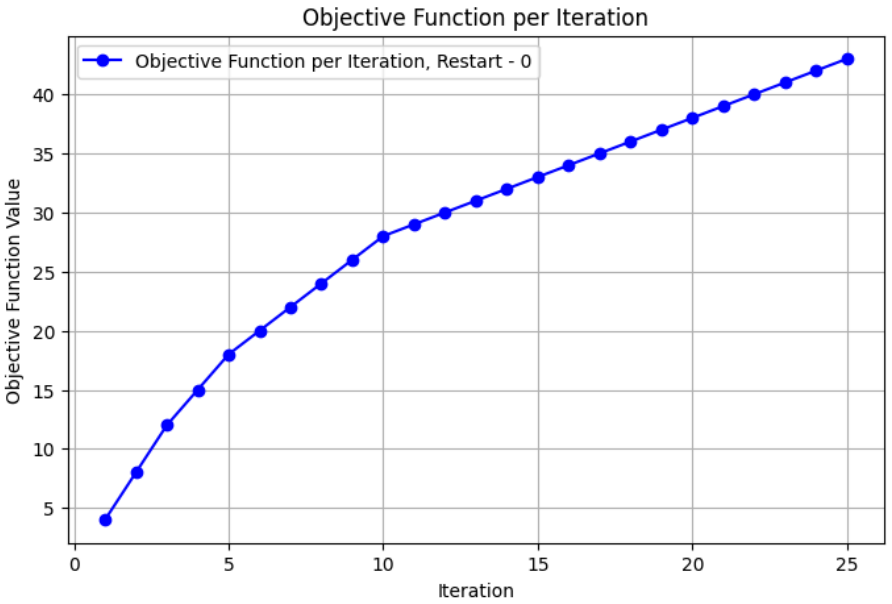
61	55	124	97	25
52	46	72	22	62
66	105	75	34	39
63	120	108	114	71
73	100	70	69	99

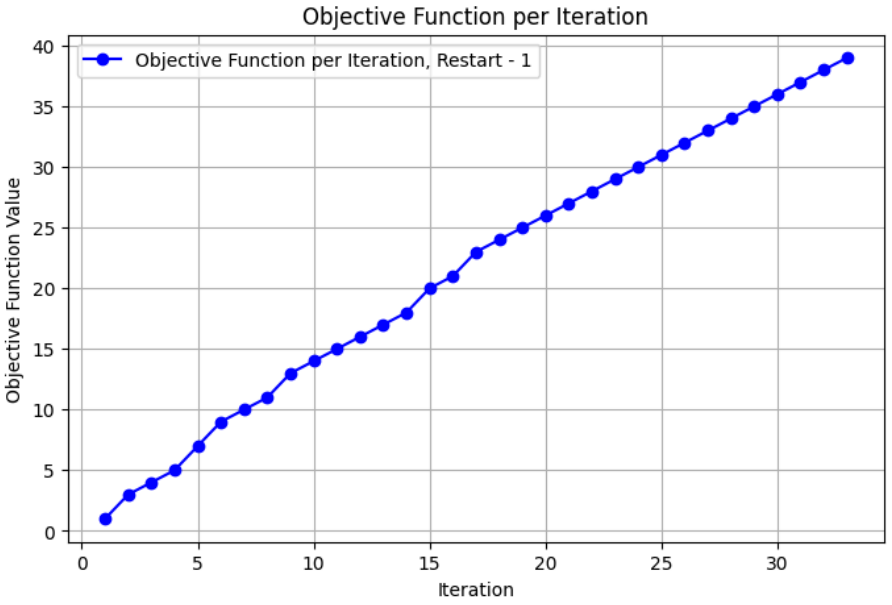


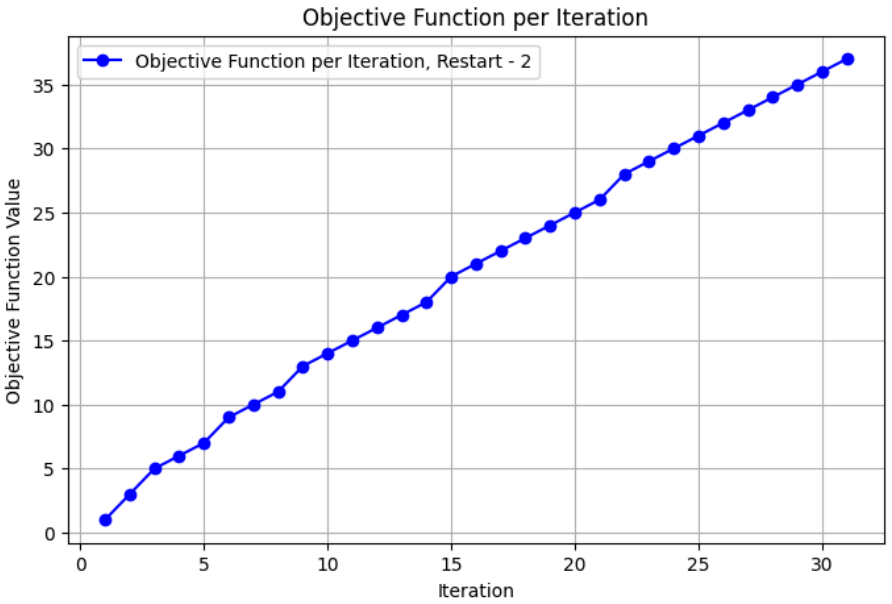


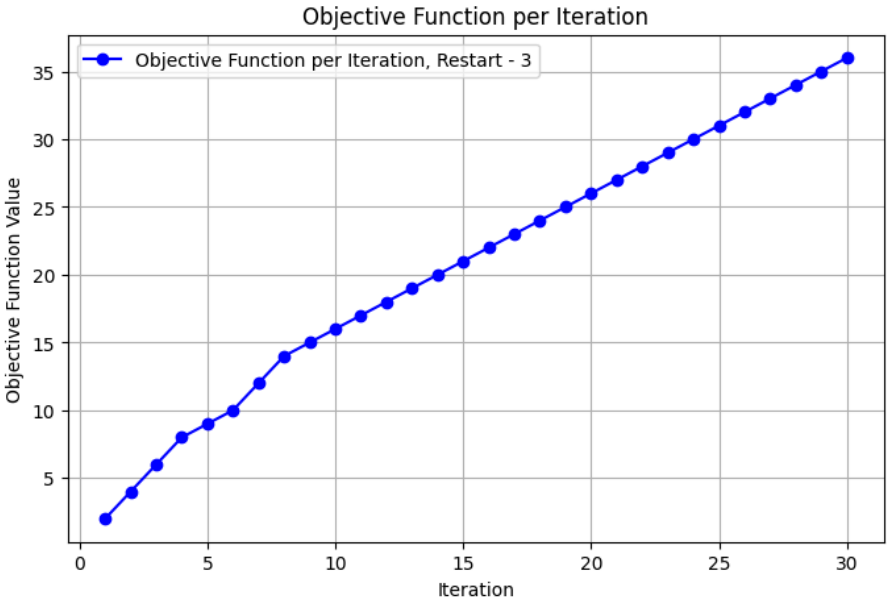
Nilai Objective Function: 43

Plot Nilai Objective Function:


Objective Function per Iteration
Objective Function per Iteration, Restart - 0
Objective Function Value
Iteration
0
5
10
15
20
25
5
10
15
20
25
30
35
40

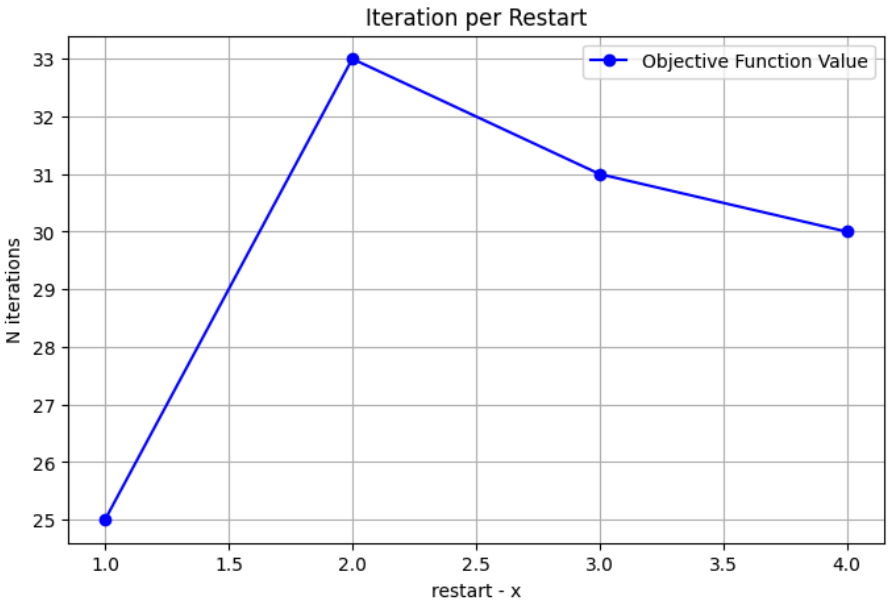

Objective Function per Iteration
Objective Function per Iteration, Restart - 1
Objective Function Value
Iteration
0
5
10
15
20
25
30
0
5
10
15
20
25
30
35
40





Banyak Restart: 3

Banyak Iterasi per Restart:



2.2.4. Stochastic Hill-climbing

Langkah-langkah pada algoritma ini:

1. Membangkitkan cube random untuk dilakukan local search. Diinisiasi juga iterasi untuk menghitung banyaknya iterasi.
2. Mulai loop, bangkitkan cube baru yang merupakan successor random dari state sekarang
3. Jika nilai state dari neigbour tersebut lebih bagus dari state sekarang maka pindah ke state tersebut, jika tidak maka diam di state sekarang
4. Lakukan loop selama iterasi belum melebihi max iterasi
5. Setelah loop selesai, maka mengembalikan state akhir, value dari state akhir, dan banyak iterasi yang dilakukan

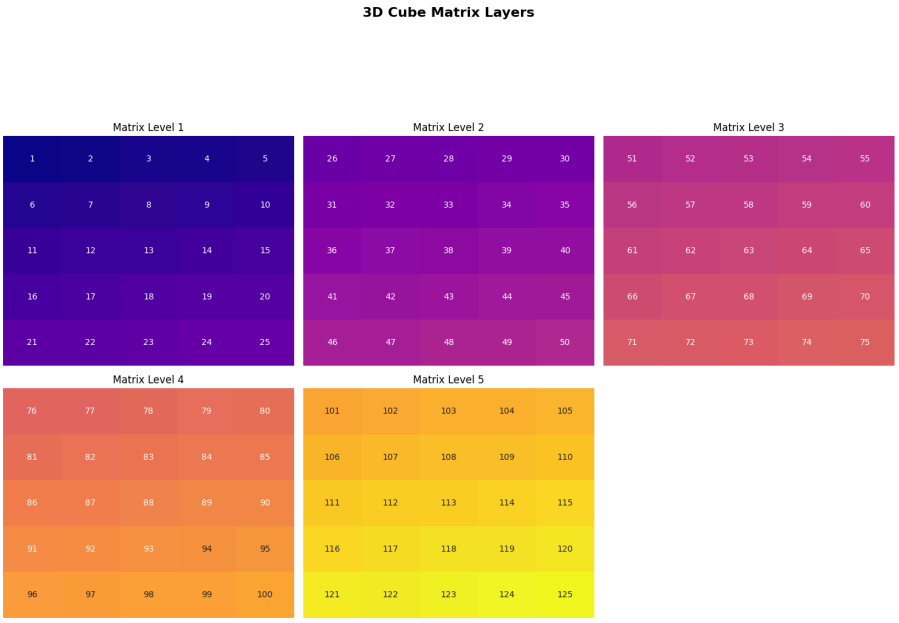
Source Code

```
def StochasticHillClimbingCube(init_cube, max_iter):
    current_cube = Cube(5, 5, 5, False, init_cube)
    current_value = current_cube.state_value
    iter = 0

    values = [current_value]
    cubes = [current_cube]

    while iter <= max_iter:
        iter += 1
        random_neighbor = current_cube.copy()
```

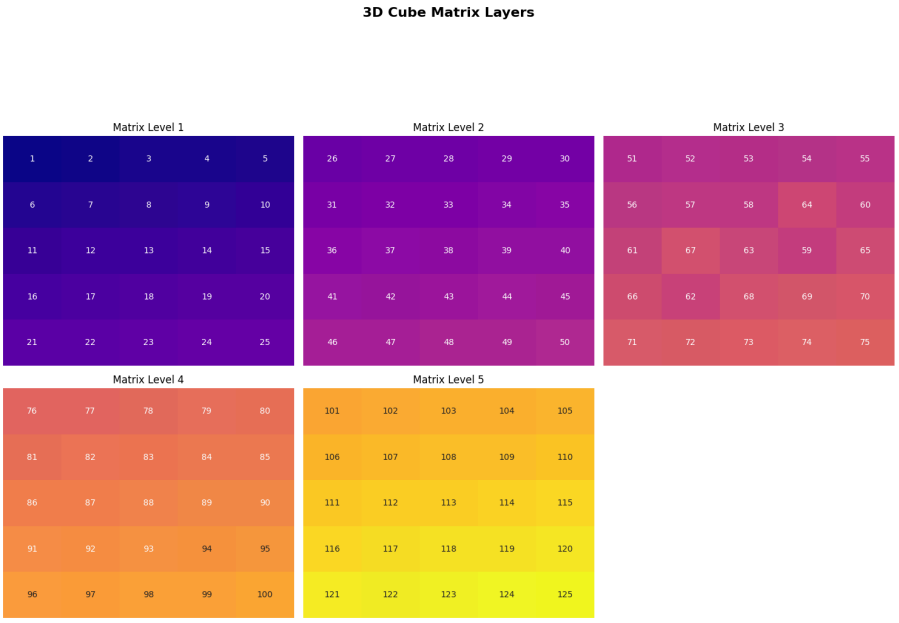
```
        z1, x1, y1 = random.randint(0, 4), random.randint(0, 4),
random.randint(0, 4)
        z2, x2, y2 = random.randint(0, 4), random.randint(0, 4),
random.randint(0, 4)
        while (z1, x1, y1) == (z2, x2, y2):
            z2, x2, y2 = random.randint(0, 4), random.randint(0, 4),
random.randint(0, 4)

        temp = random_neighbor.get_value(z1, x1, y1)
        random_neighbor.insert_value(z1, x1, y1,
random_neighbor.get_value(z2, x2, y2))
        random_neighbor.insert_value(z2, x2, y2, temp)
        random_neighbor.calculate_state_value()

        random_value = random_neighbor.state_value

        if random_value > current_value:
            current_cube = random_neighbor
            current_value = random_value
        print("Val:", current_value)

        values.append(current_value)
        cubes.append(current_cube)

    return cubes, values, iter
```

2.2.4.1. Percobaan 1

State Awal

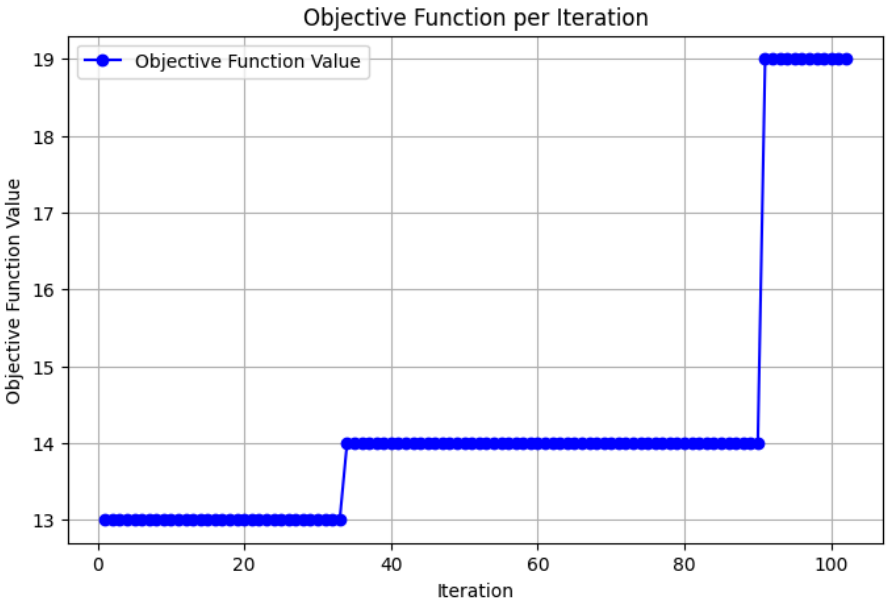


State Akhir



Nilai Objective Function: 19

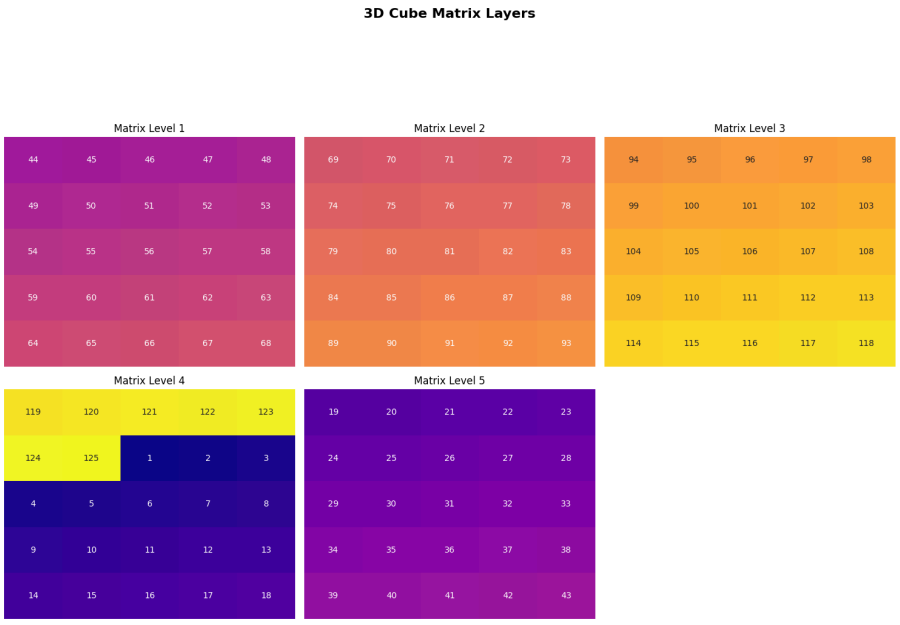
Plot Nilai Objective Function:



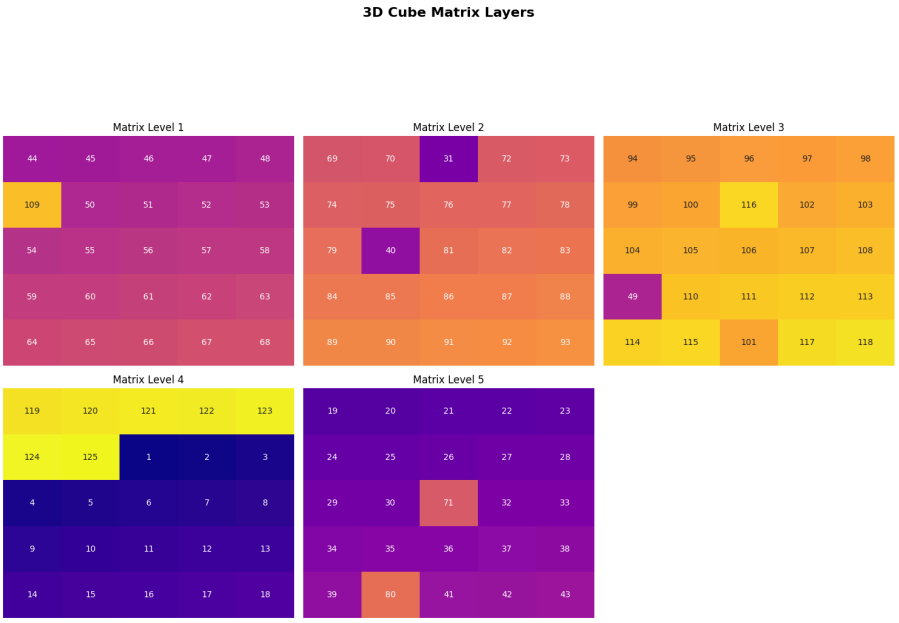
Banyak Iterasi: 101

2.2.4.2. Percobaan 2

State Awal

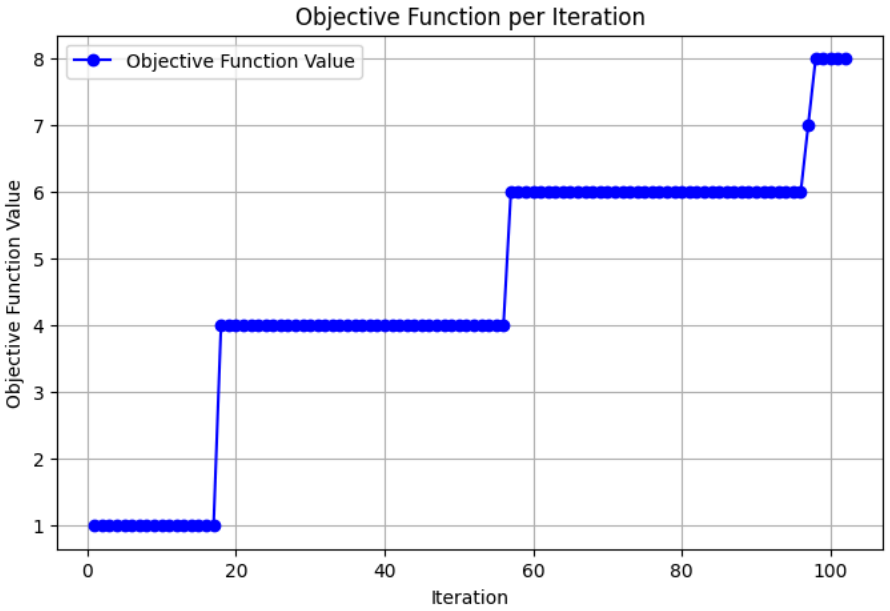


State Akhir



Nilai Objective Function: 8

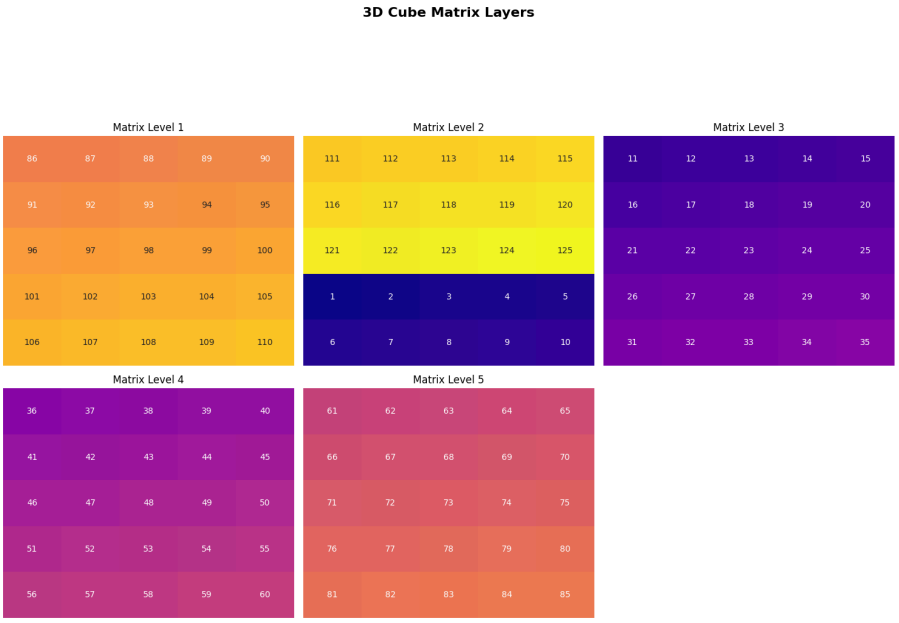
Plot Nilai Objective Function:



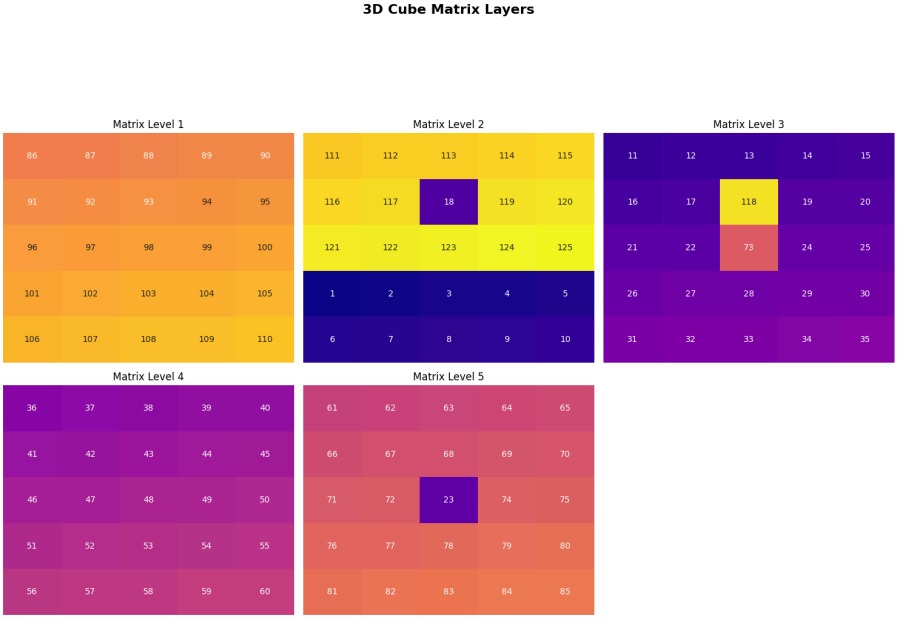
Banyak Iterasi: 101

2.2.4.3. Percobaan 3

State Awal

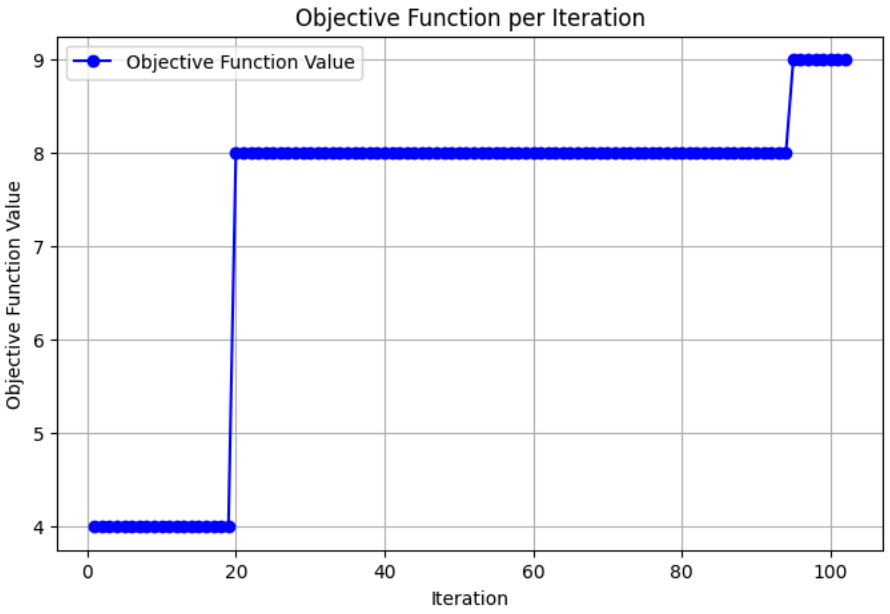


State Akhir



Nilai Objective Function: 4

Plot Nilai Objective Function:



Banyak Iterasi: 101

2.2.5. Simulated Annealing

Langkah-langkah pada algoritma ini :

1. Membangkitkan cube random untuk dilakukan local search. Diinisiasi juga nilai iterasi, count stuck
2. Lakukan loop dimana pertama dihitung dulu nilai T berdasarkan fungsi schedule
3. Jika T sama dengan 0 maka return
4. Increment jumlah iterasi dan bangkitkan random successor dari current state
5. Hitung nilai deltaE yang merupakan selisih dari neighbour state value dan current state value
6. Jika deltaE positif maka langsung pindah ke state tersebut, jika negatif maka hitung probabilitas menggunakan rumus $e^{\frac{\Delta E}{T}}$, kemudian menggunakan probabilitas tersebut tentukan apakah berpindah atau tidak
7. Fungsi mengembalikan semua cubes yang ditelusuri, semua value yang ditemukan, banyaknya iterasi, semua probabilitas e yang dihitung, dan banyaknya stuck

Source Code

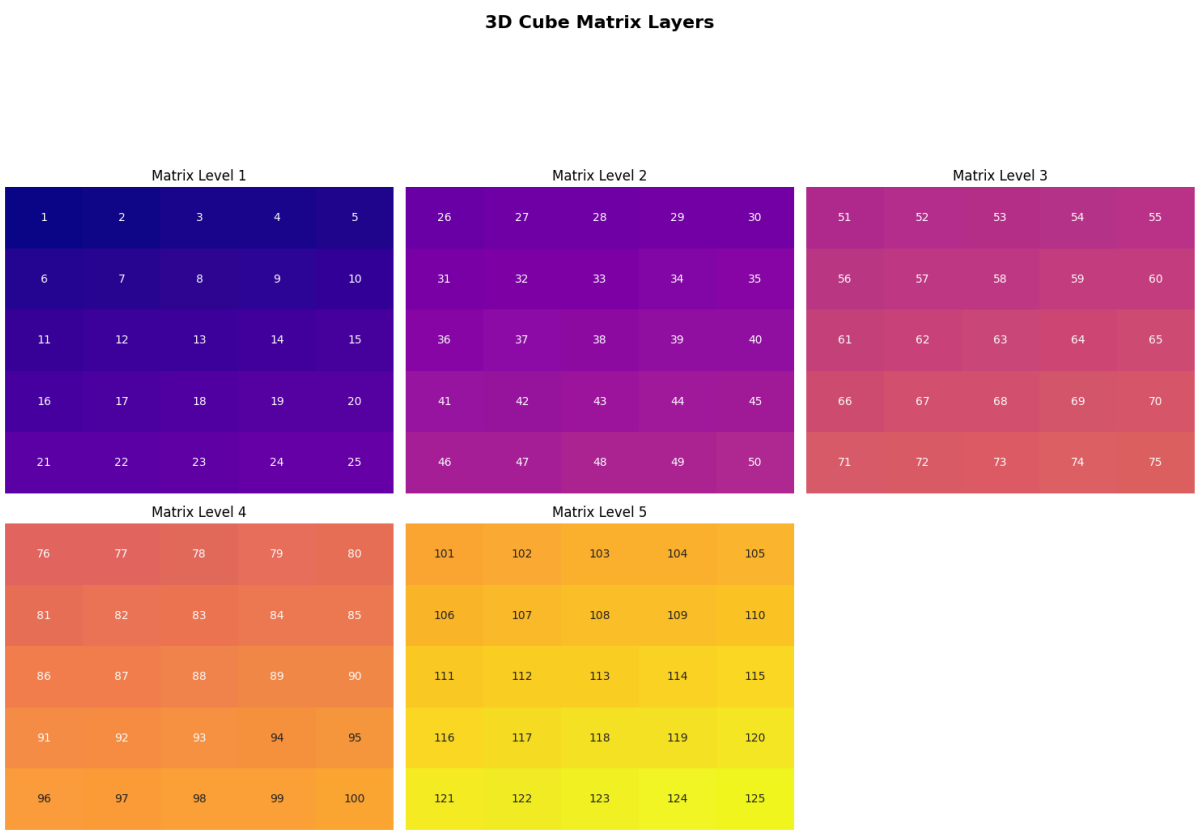
```
def schedule(t, initial_temperature, cooling_rate):
    return max(initial_temperature * math.pow(cooling_rate, t), 0)
```

```
def SimulatedAnnealingCube(init_cube, initial_temperature=1000,
cooling_rate=0.9, schedule=schedule):
    current_cube = Cube(5, 5, 5, False, init_cube)
    current_value = current_cube.state_value
    t = 1

    cubes = [current_cube]
    values = [current_value]
    count_iter = 0
    e_probs = {}
    count_stuck = 0

    while True:
        T = schedule(t, initial_temperature, cooling_rate)
        if T == 0:
            break
        count_iter += 1

        random_neighbor = current_cube.copy()

        a, b, c = random.randint(0, 4), random.randint(0, 4),
random.randint(0, 4)
        i, j, k = random.randint(0, 4), random.randint(0, 4),
random.randint(0, 4)
        while (a, b, c) == (i, j, k):
            i, j, k = random.randint(0, 4), random.randint(0, 4),
random.randint(0, 4)

        temp = random_neighbor.get_value(a, b, c)
        random_neighbor.insert_value(a, b, c,
random_neighbor.get_value(i, j, k))
        random_neighbor.insert_value(i, j, k, temp)
        random_neighbor.calculate_state_value()

        random_value = random_neighbor.state_value
        deltaE = random_value - current_value

        if deltaE > 0:
            current_cube = random_neighbor
            current_value = random_value
        else:
            if deltaE < 0:
                e_probs[count_iter] = math.exp(deltaE / T)
            if random.random() < math.exp(deltaE / T):
```

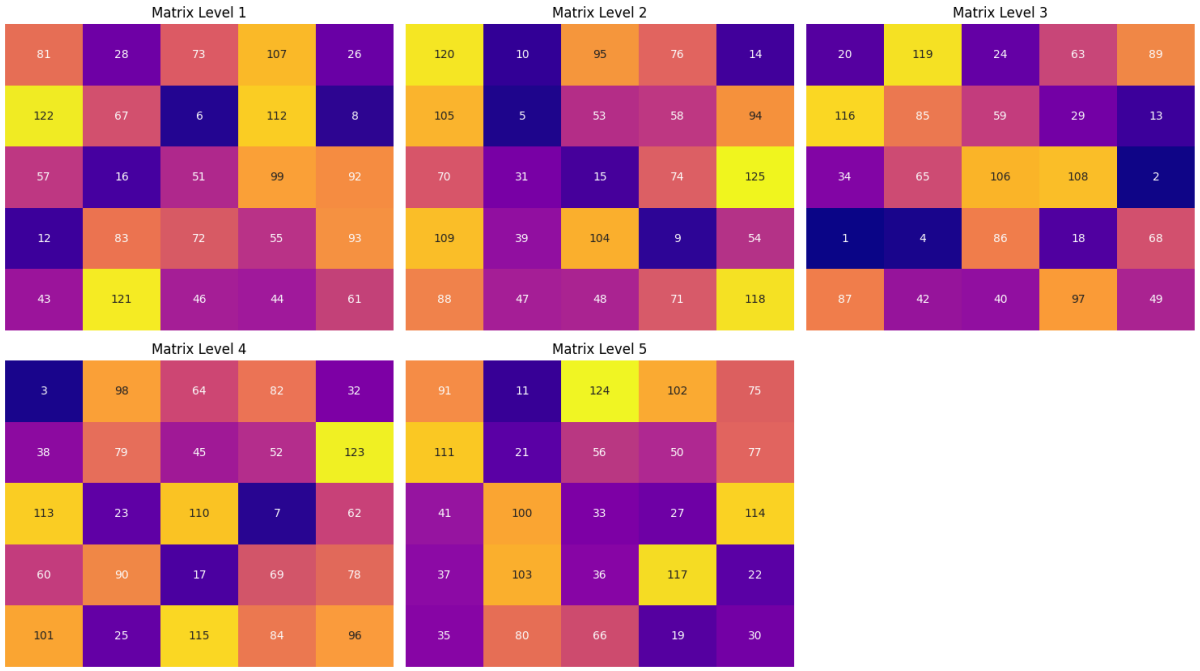
```
                count_stuck += 1
                current_cube = random_neighbor
                current_value = random_value

        cubes.append(current_cube)
        values.append(current_value)
        t += 1

    return cubes, values, count_iter, e_probs, count_stuck
```

2.2.5.1. Percobaan 1

State Awal

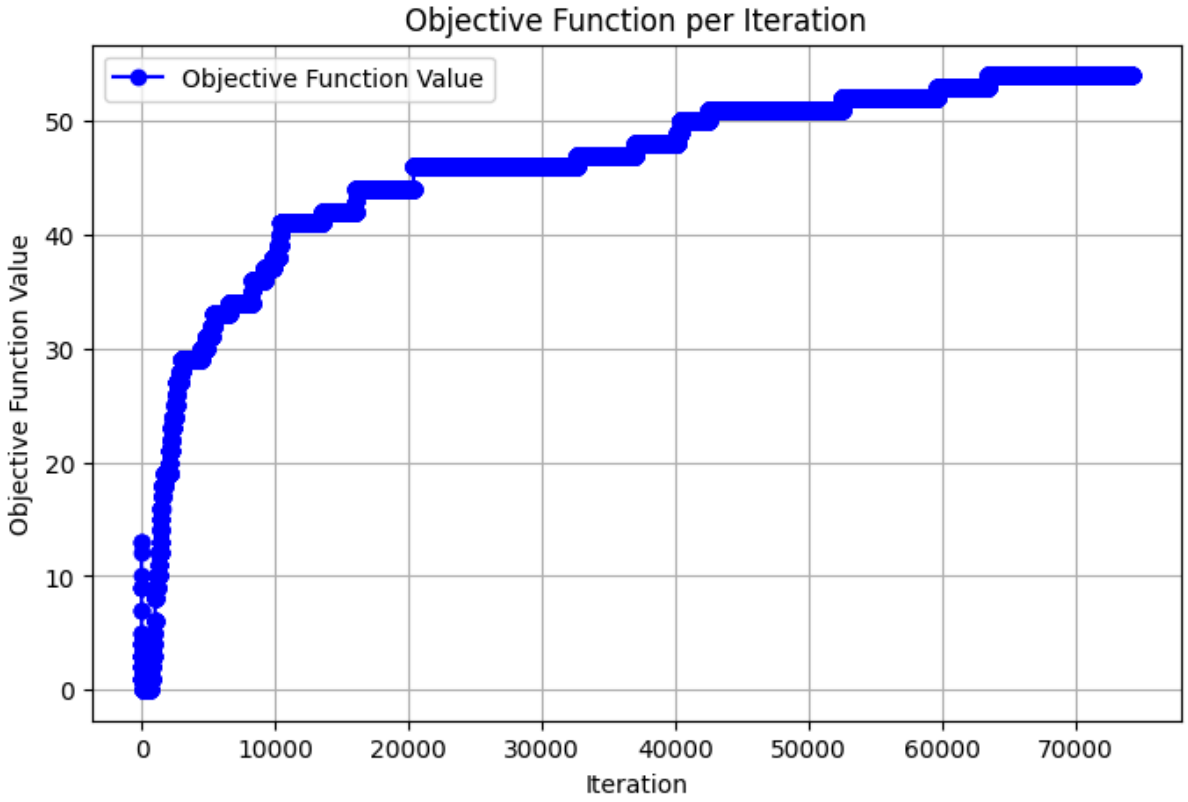


State Akhir

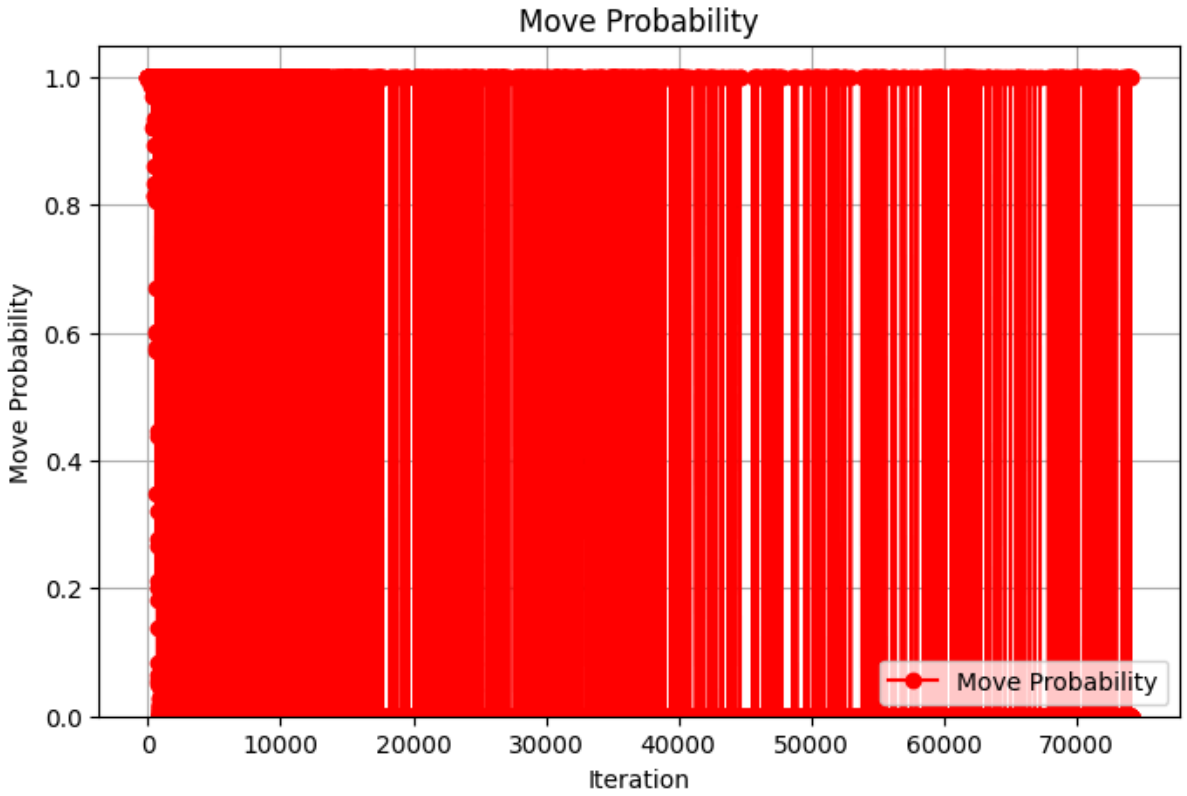


Nilai Objective Function: 54

Plot Nilai Objective Function:



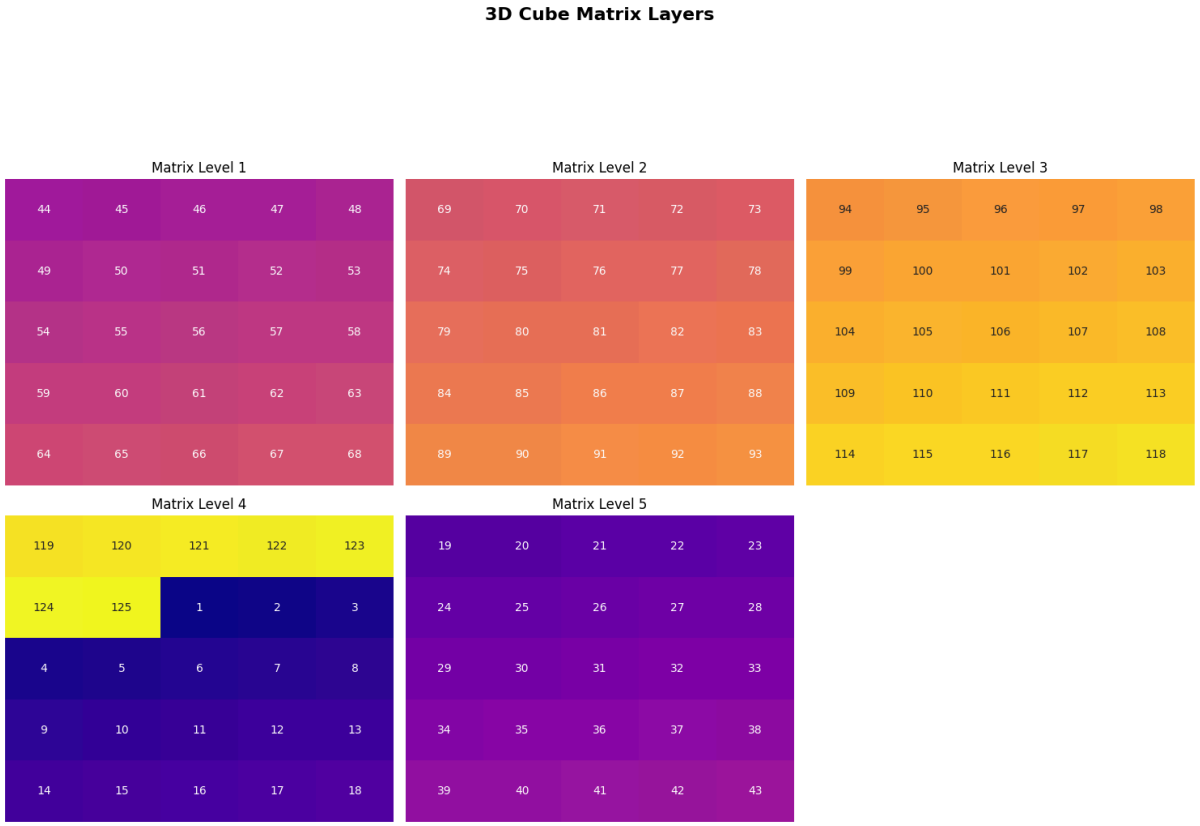
Plot $e^{\frac{\Delta E}{T}}$:



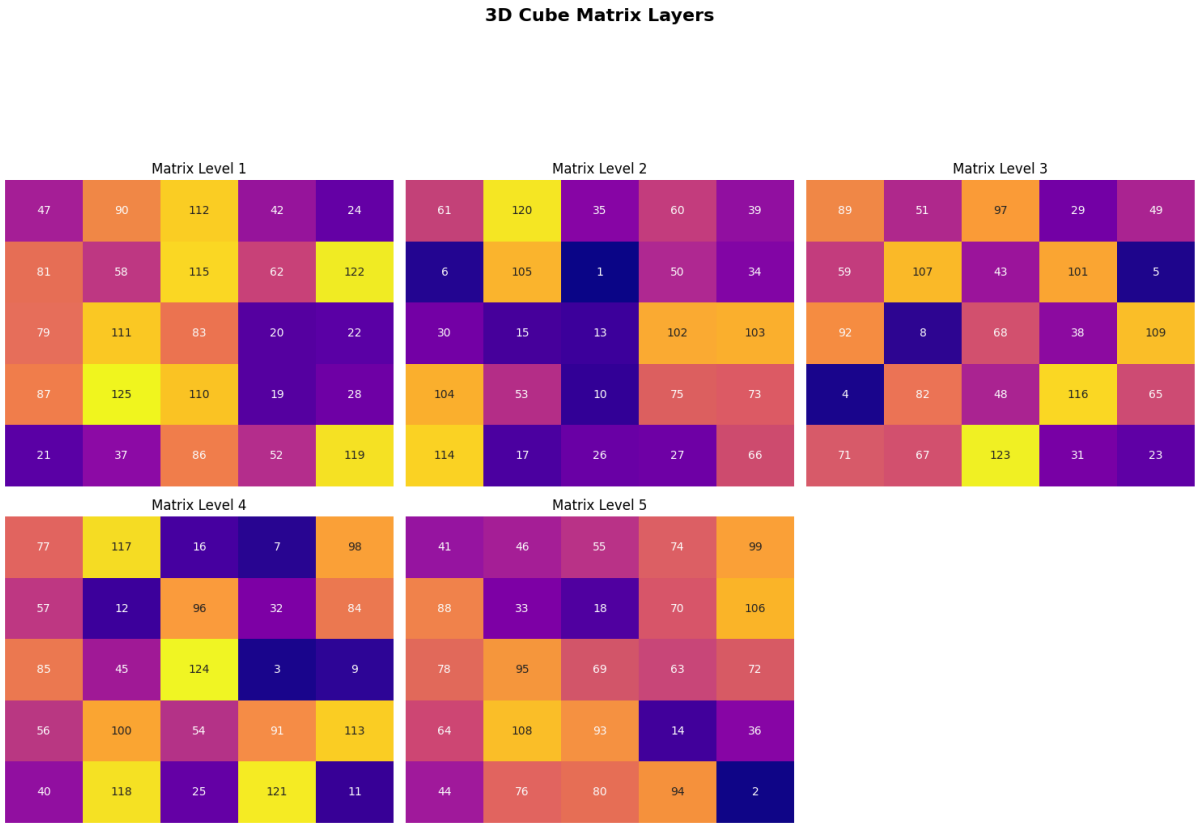
Frekuensi Stuck: 2460

2.2.5.2. Percobaan 2

State Awal

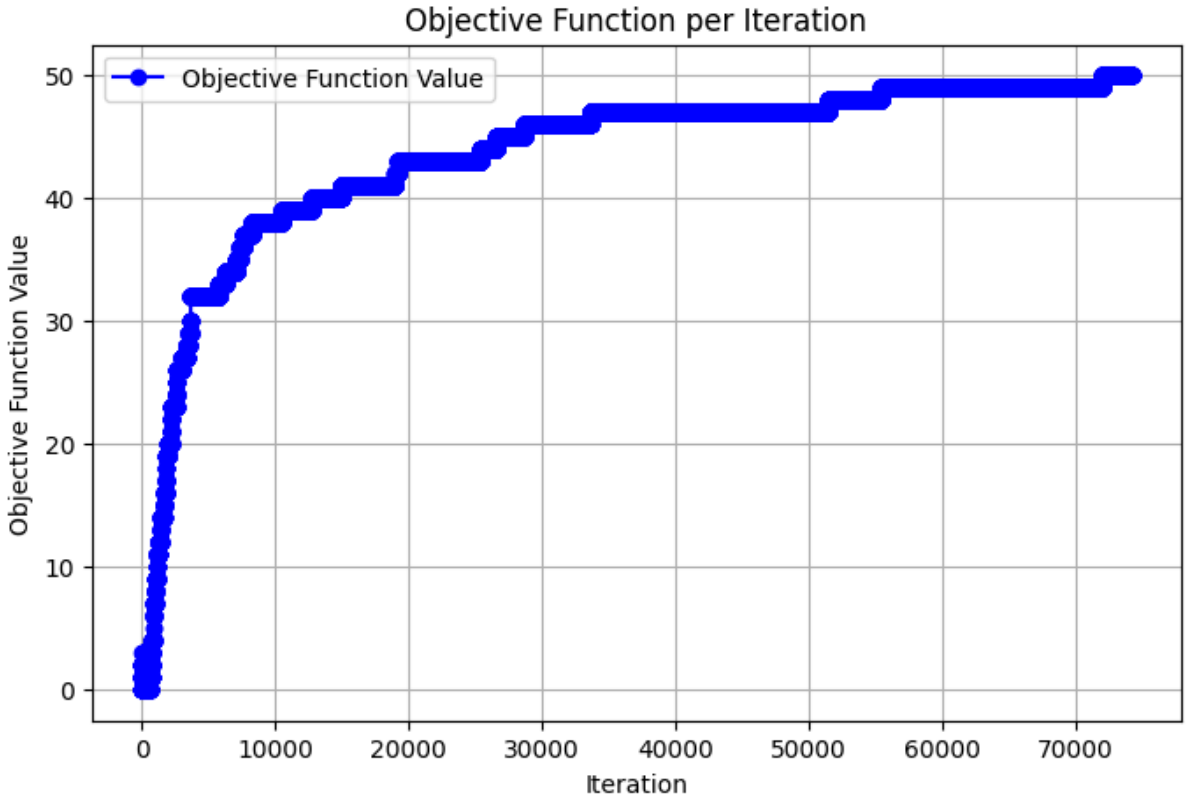


State Akhir

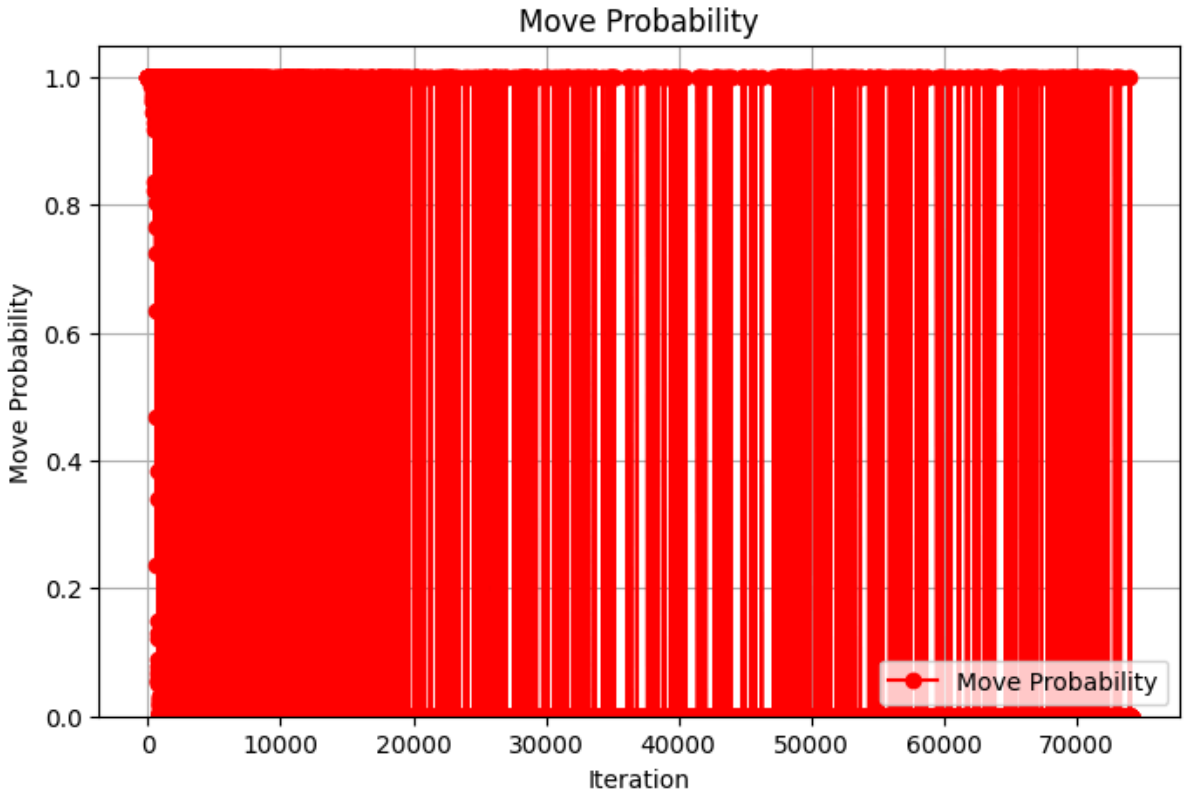


Nilai Objective Function: 50

Plot Nilai Objective Function:



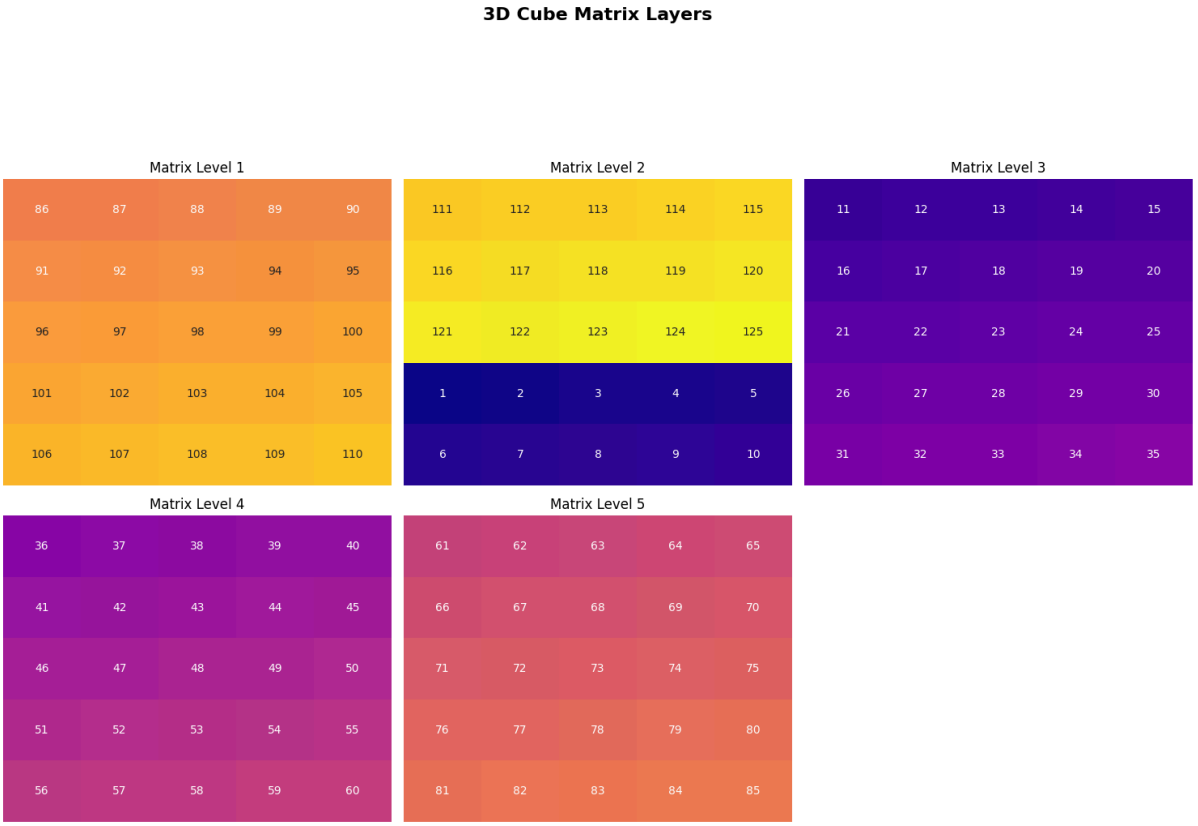
Plot $e^{\frac{\Delta E}{T}}$:



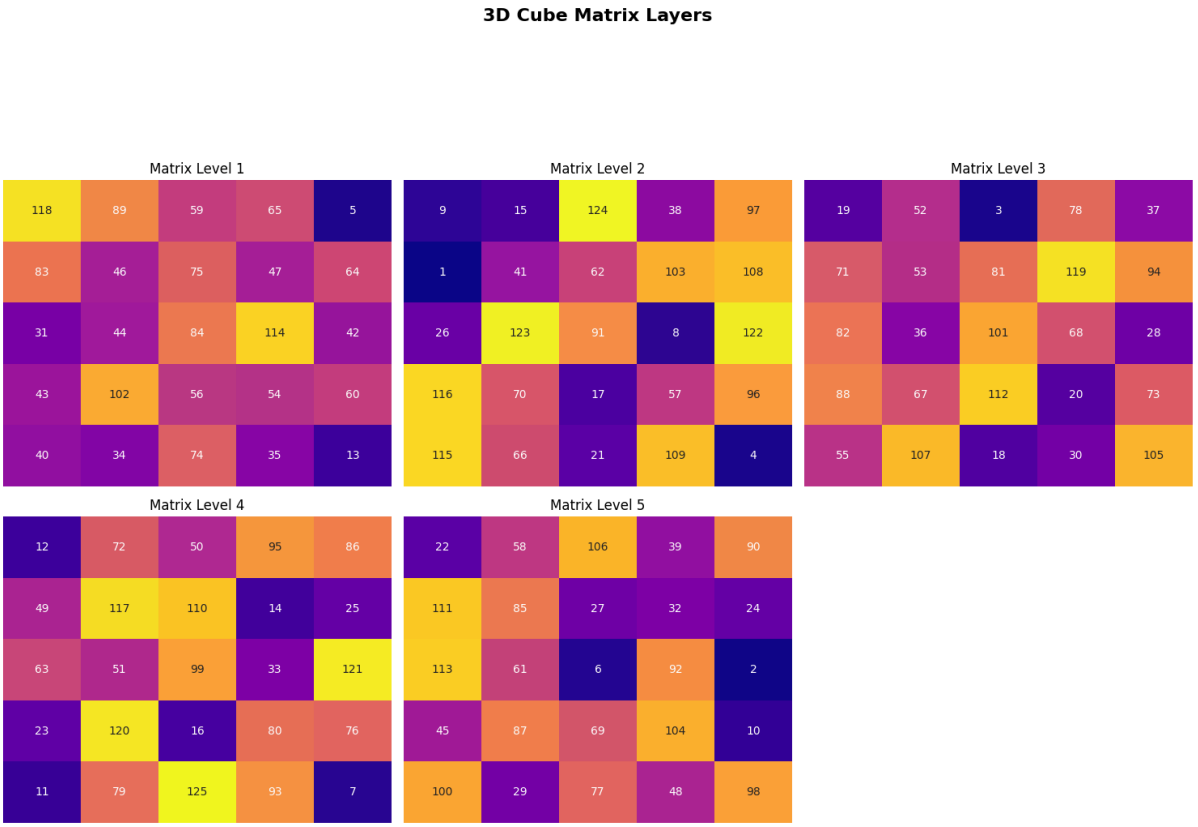
Frekuensi Stuck: 2012

2.2.5.3. Percobaan 3

State Awal

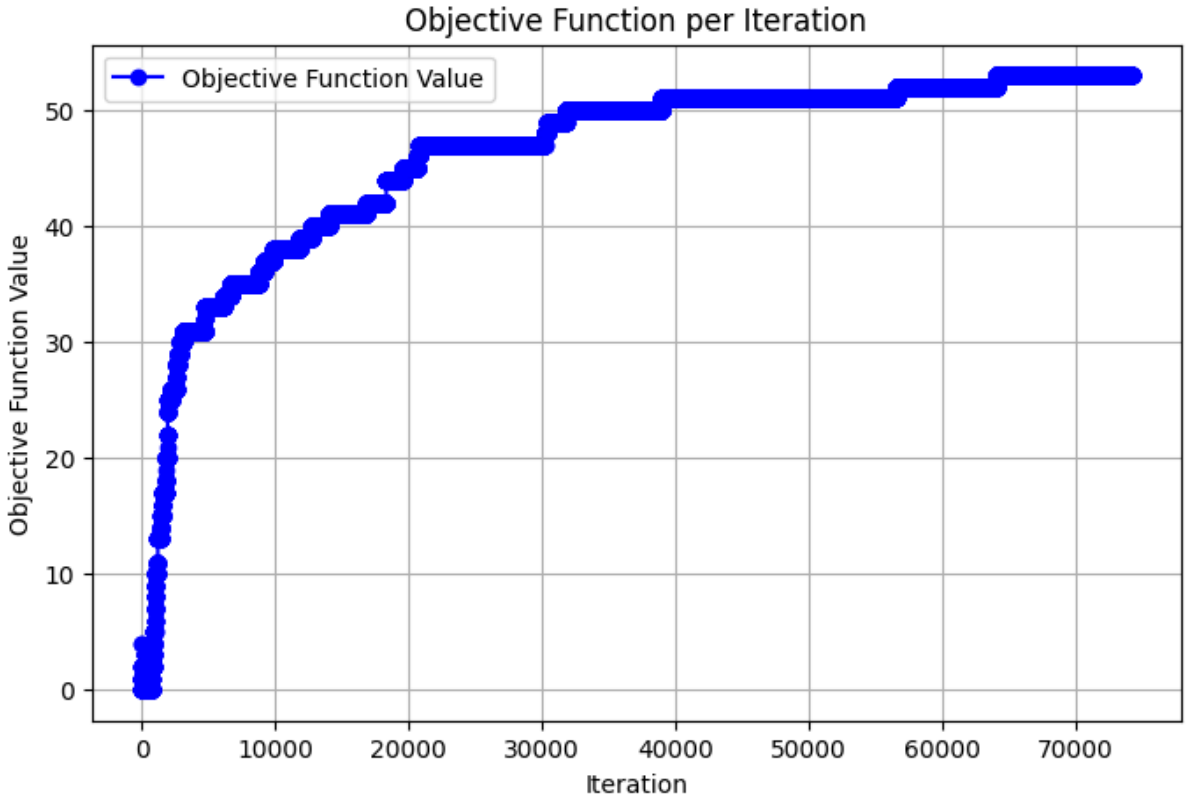


State Akhir

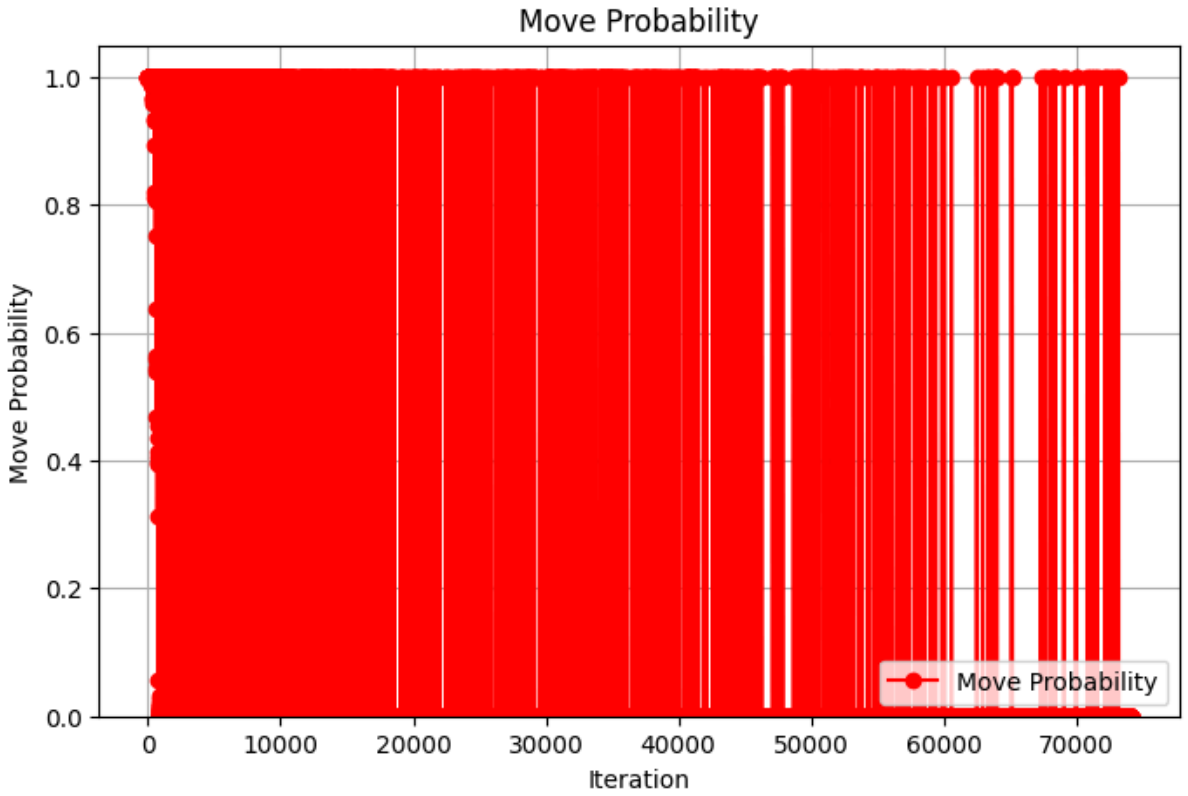


Nilai Objective Function: 53

Plot Nilai Objective Function:



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



Frekuensi Stuck: 1134

2.2.6. Genetic Algorithm

Langkah-langkah pada algoritma ini :

1. Menghasilkan populasi awal sebanyak *N* individu (*cube*).
2. Dari antara individu dalam populasi tersebut, dicari *current_max_fit* yakni individu (dalam contoh ini *cube*) yang memiliki nilai fitness terbesar
3. Setelah itu, dimulainya proses pembentukan populasi baru.
4. Pembentukan populasi baru dimulai dengan mengisi sebanyak *elite_size* individu yang memiliki fitness value terbesar.
5. Setelah itu dimulainya proses *roulette wheel selection* dari antara populasi awal untuk memilih *parents* yang akan digunakan pada proses *crossover*.
6. Setelah terpilih dua individu sebagai *parents*, dilakukan proses *crossover* (terpilih parent1 dan parent2).
7. Proses *crossover* dilakukan dengan metode *Ordered Crossover*, metode ini dimulai dengan memilih dua index yang akan menjadi *start* dan *end* area terpotong antara parent1 dan parent2 yang kemudian kedua area tersebut akan ditukar satu sama lain menghasilkan child1 dan child2.
8. Area kosong pada child1 dan child2 kemudian diisi dengan nilai - nilai pada salah satu parent (parent yang tidak memberikan potongan areanya).

9. child1 dan child2 kemudian melalui proses *mutation*, proses ini terjadi berdasarkan probabilitas tertentu.
10. Proses *mutation* dilakukan dengan melakukan *swap* antara dua element dalam *magic cube*.
11. Setelah melakukan proses *mutation*, dipilihnya antara child1 dan child2 sebagai hasil keturunan, child yang dipilih berdasarkan nilai fitness terbesar.
12. Langkah 5 sampai 11 dilakukan ulang sampai populasi baru (*new_population)* memiliki ukuran sebesar N.
13. Setelah mendapatkan *new_population* maka dipilihnya *current_max_fit* yakni individu (*cube*) yang memiliki *fitness value* terbesar dalam *new_population.*
14. *Population* kemudian diganti dengan *new_population*.
15. Langkah 3 - 14 dilakukan ulang sebanyak *max_genexrations*.

Source code

```
import random
import heapq
import multiprocessing as mp
import Cube as c

def mutate(child, mutation_rate=0.01):
    size = len(child.array)
    if random.random() < mutation_rate:
        i, j = random.sample(range(size), 2)
        child.array[i], child.array[j] = child.array[j],
child.array[i]
    child.calculate_state_value()
    return child

def crossover(parent1, parent2):
    size = len(parent1.array)
    child1_array = [None] * size
    child2_array = [None] * size
    start, end = sorted(random.sample(range(size), 2))

    child1_array[start : end+1] = parent1.array[start : end+1]
    child2_array[start : end+1] = parent2.array[start : end+1]

    used1 = set(child1_array[start : end+1])
    used2 = set(child2_array[start : end+1])

    # Create remaining values lists
```

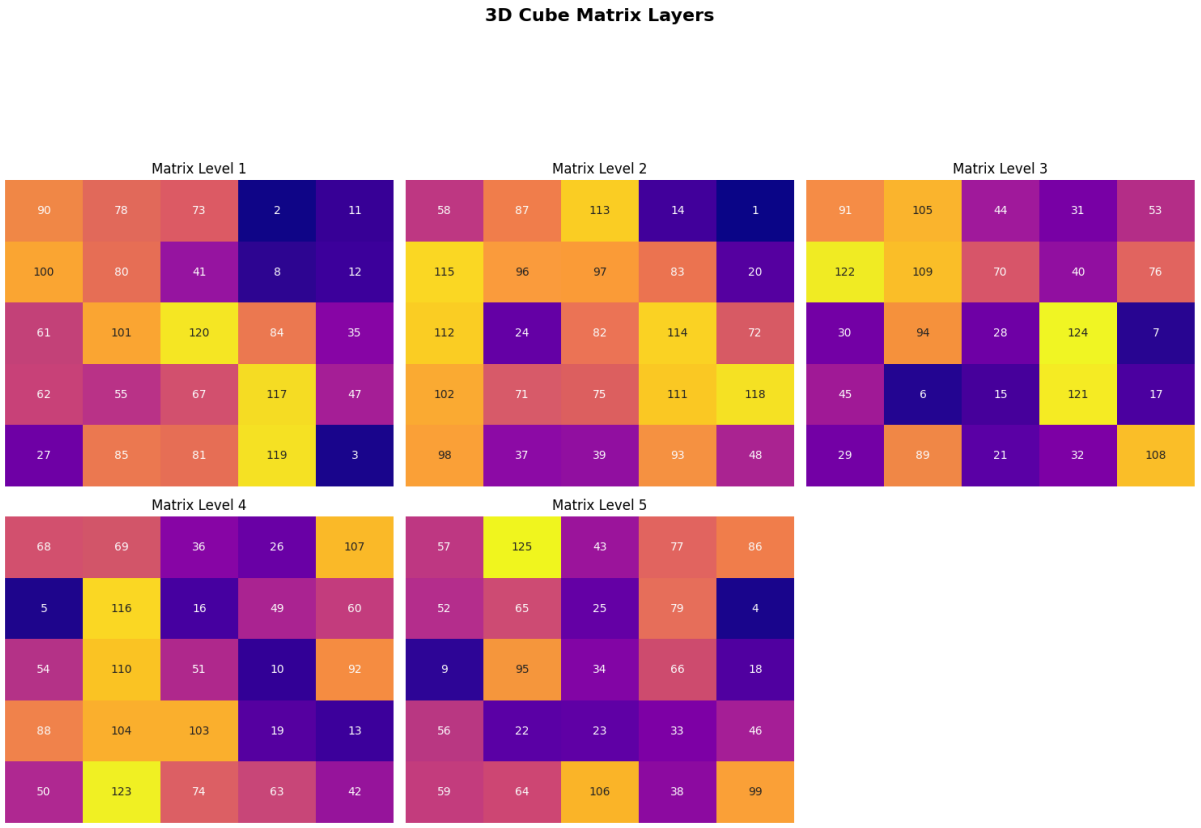
```
    remaining1 = [x for x in parent2.array if x not in used1]
    remaining2 = [x for x in parent1.array if x not in used2]

    idx1 = idx2 = 0
    for i in range(size):
        if i < start or i > end:
            child1_array[i] = remaining1[idx1]
            child2_array[i] = remaining2[idx2]
            idx1 += 1
            idx2 += 1

    child1 = c.Cube(parent1.x_size, parent1.y_size, parent1.z_size,
array=child1_array)
    child2 = c.Cube(parent2.x_size, parent2.y_size, parent2.z_size,
array=child2_array)

    return child1, child2

def roulette_wheel_selection(population):
    total_fitness = sum(ind.state_value for ind in population)
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual in population:
        current += individual.state_value
        if current > pick:
            return individual

def create_child_task(args):
    parent1, parent2, mutation_rate = args
    child1, child2 = crossover(parent1, parent2)
    child1 = mutate(child1, mutation_rate)
    child2 = mutate(child2, mutation_rate)
    return child1 if child1.state_value > child2.state_value else
child2

def genetic(N=100, max_generations=1000, initial_mutation_rate=0.05,
stagnation_limit=100):
    population = [c.Cube(5, 5, 5, True) for _ in range(N)]
    current_max_fit = max(population, key=lambda x: x.state_value)
    values = [current_max_fit.state_value]
    no_improvement_count = 0
    mutation_rate = initial_mutation_rate
    elite_size = int(0.05 * N)
    cubes = []
```

```
   with mp.Pool(mp.cpu_count()) as pool:
       for generation in range(max_generations):
           new_population = heapq.nlargest(elite_size, population,
key=lambda x: x.state_value)


           mutation_rate = min(0.3, mutation_rate * 1.2) if
no_improvement_count > stagnation_limit / 2 else
initial_mutation_rate

           tasks = [
               (roulette_wheel_selection(population),
roulette_wheel_selection(population), mutation_rate)
               for _ in range((N - elite_size) // 2)
           ]

           children = pool.map(create_child_task, tasks)
           new_population.extend(children)

           population = heapq.nlargest(N, new_population, key=lambda
x: x.state_value)
           best_child = population[0]

           if best_child.state_value > current_max_fit.state_value:
               current_max_fit = best_child
               no_improvement_count = 0
           else:
               no_improvement_count += 1

           if(generation%100 == 0):
               print(f"Generation {generation}: Current max fitness
= {current_max_fit.state_value}")

           if no_improvement_count >= stagnation_limit:
               print("STAGNATION ALERT !!!")
               random_individuals = [c.Cube(5, 5, 5, True) for _ in
range(int(0.5 * N))]
               population = population[:int(0.5 * N)] +
random_individuals
               no_improvement_count = 0

           values.append(current_max_fit.state_value)
           cubes.append(current_max_fit)
```

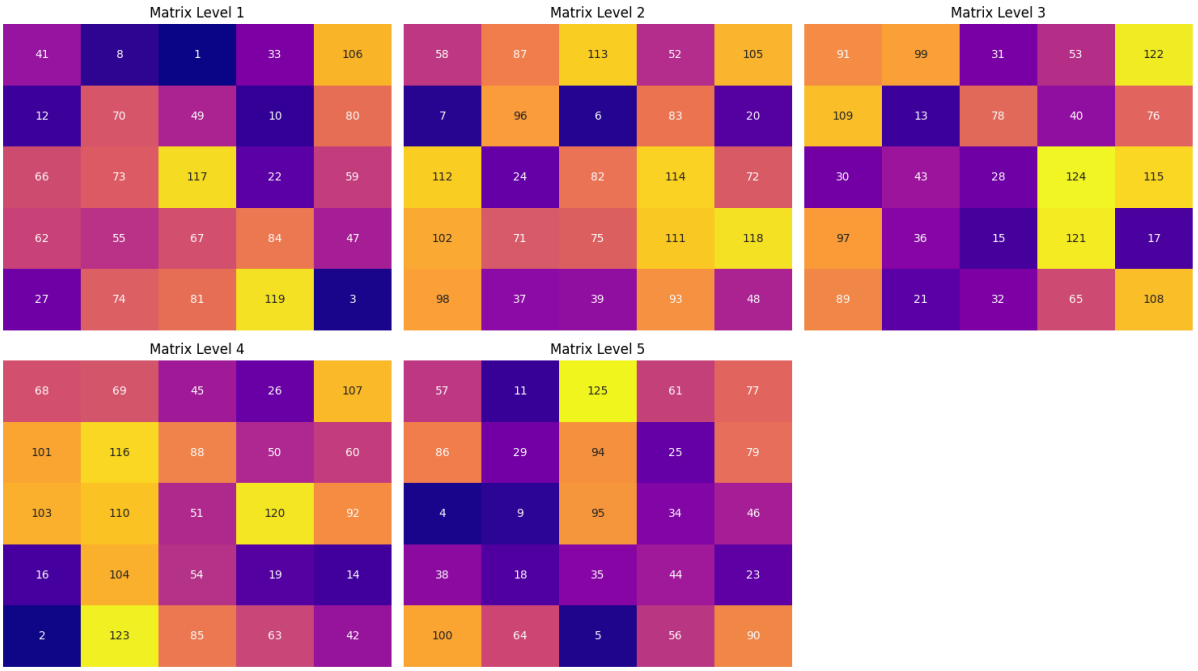
```
        print("Final solution:")
        current_max_fit.print_cube()

    return current_max_fit, values, cubes, generation + 1
```

2.2.6.1. Percobaan 1

State Awal

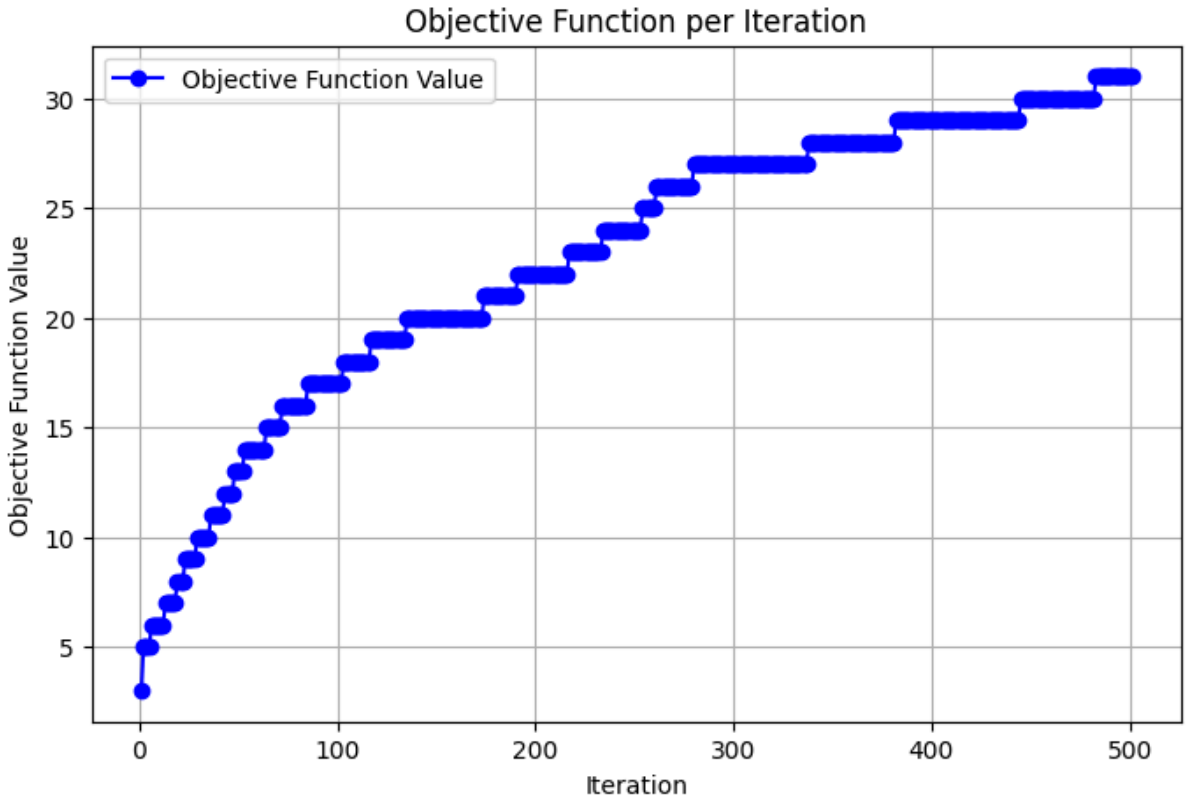


State Akhir

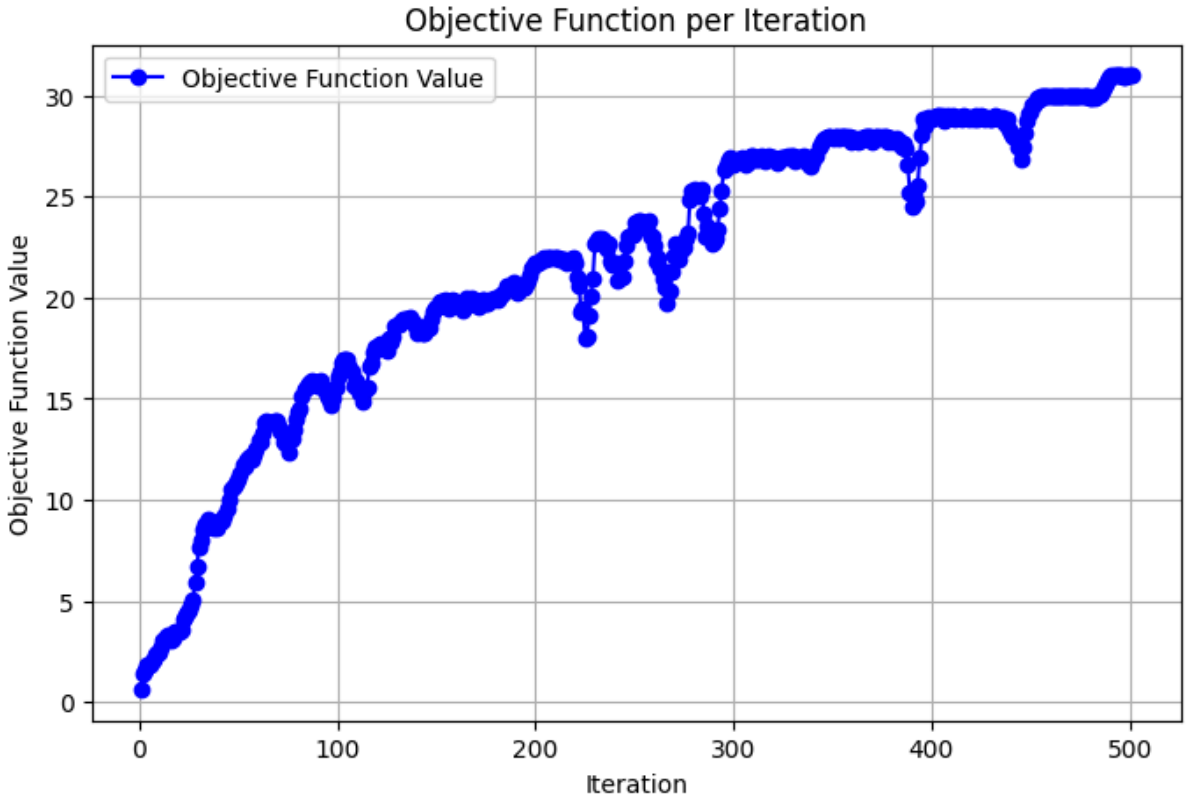


Nilai Objective Function: 31

Plot Nilai Objective Function Maximum:



Plot Nilai Objective Function Rata-Rata:



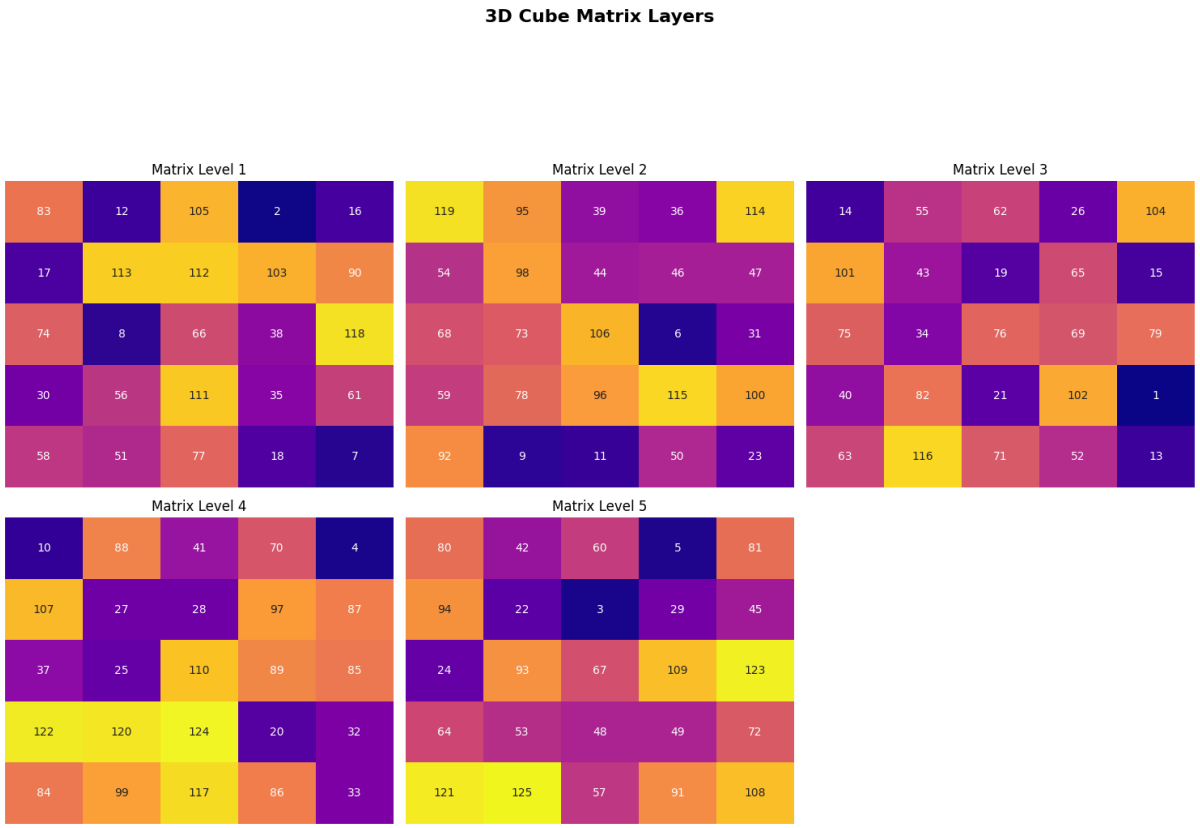
Jumlah Populasi: 200

Banyak Iterasi: 500

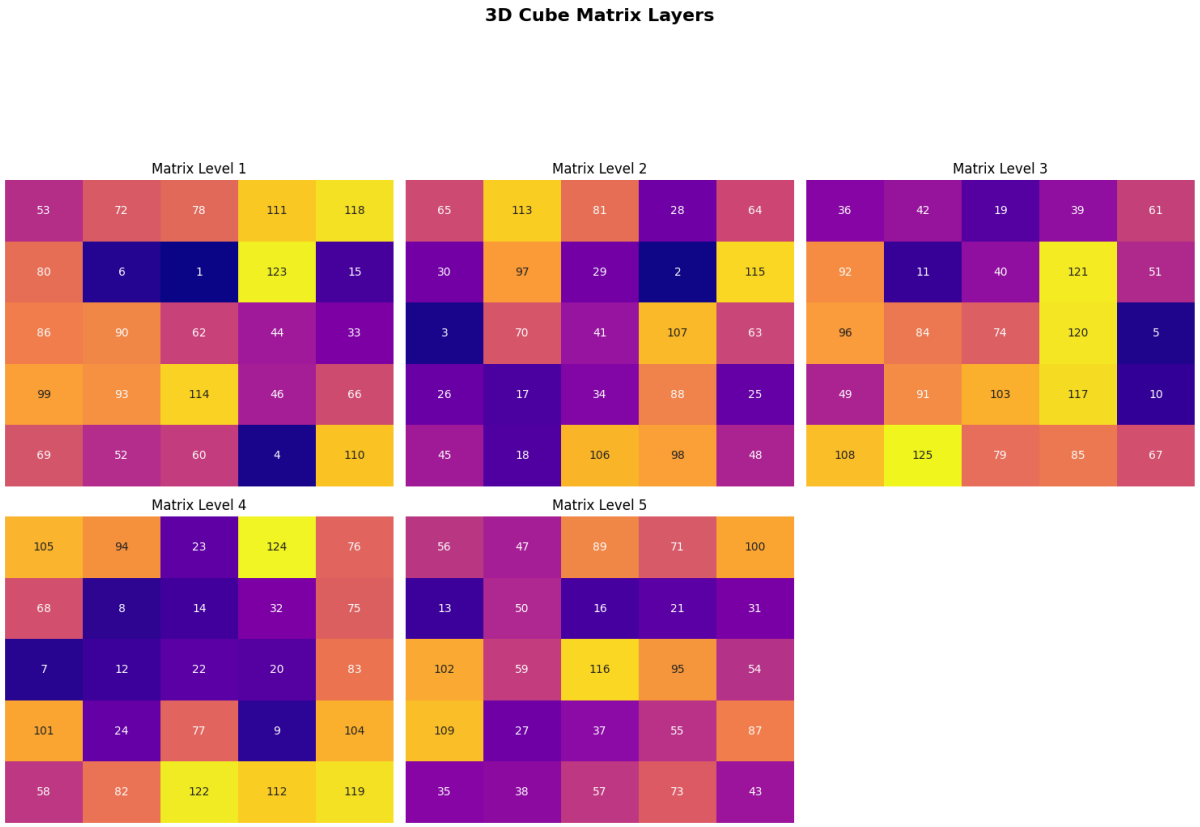
Durasi Proses Pencarian: 12 seconds

2.2.6.2. Percobaan 2

State Awal

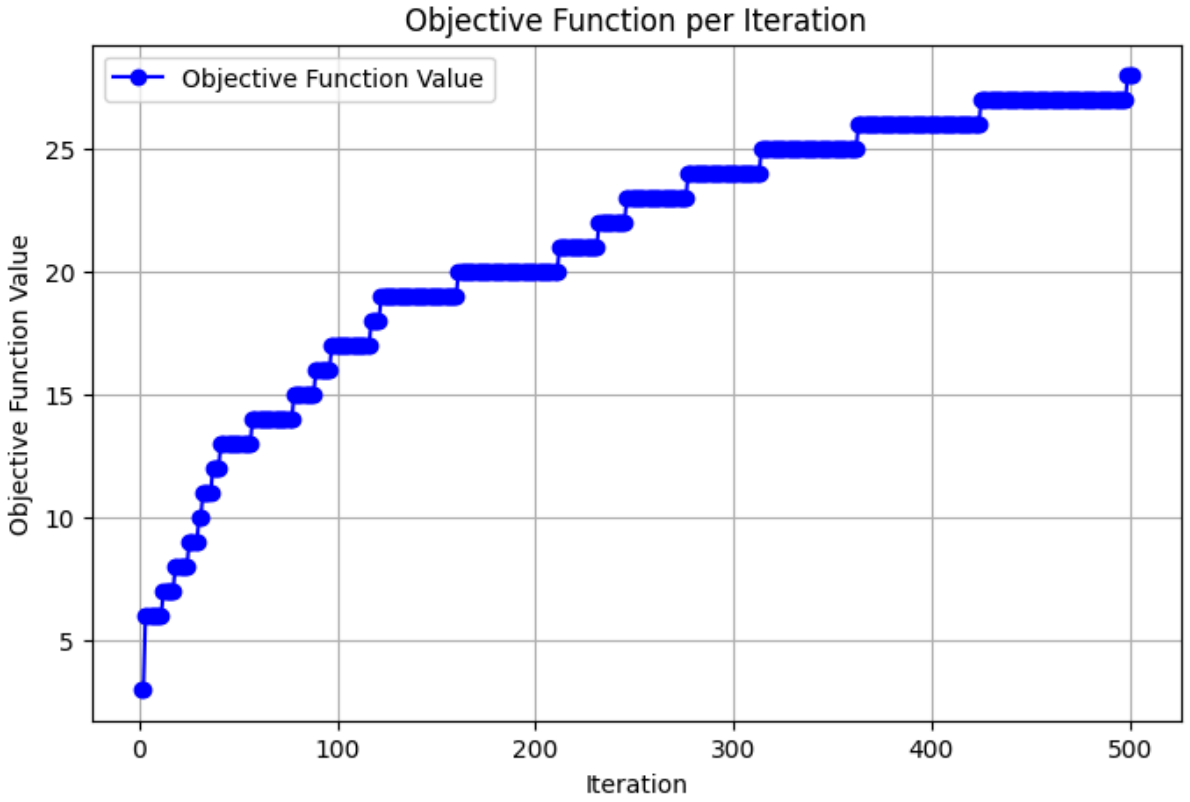


State Akhir

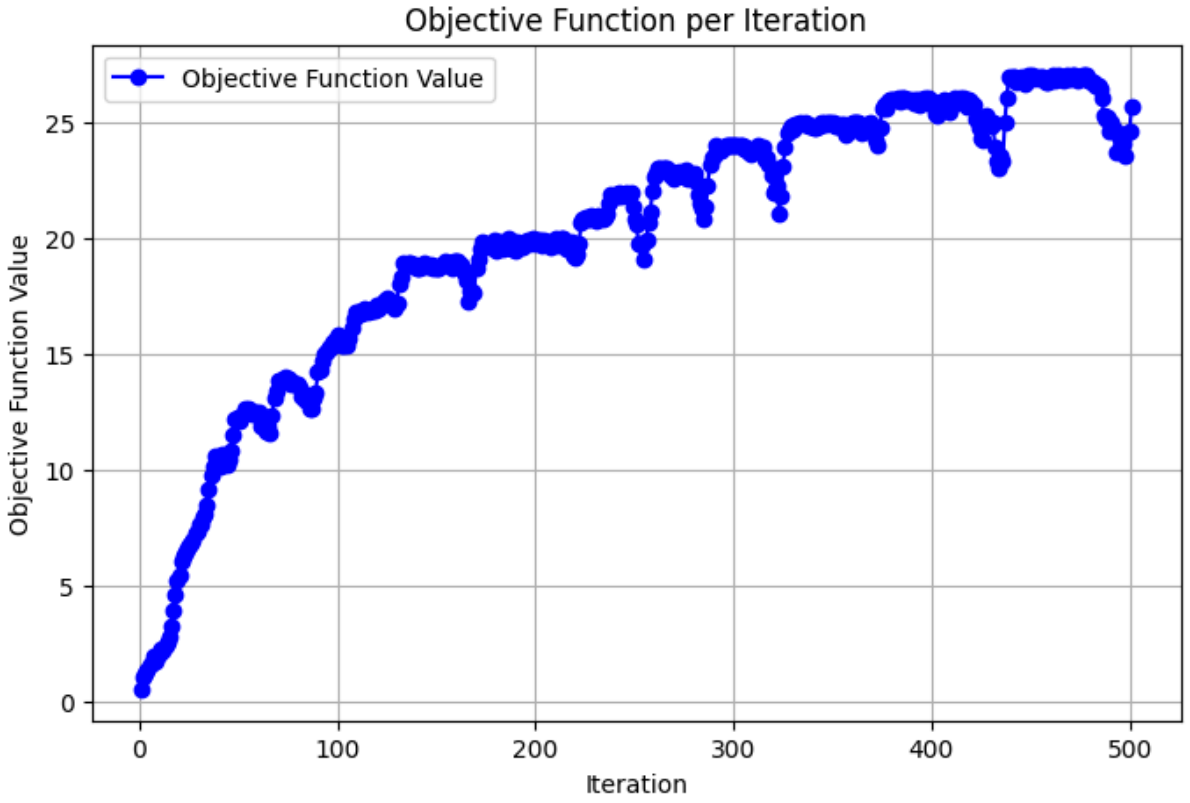


Nilai Objective Function: 31

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Average:



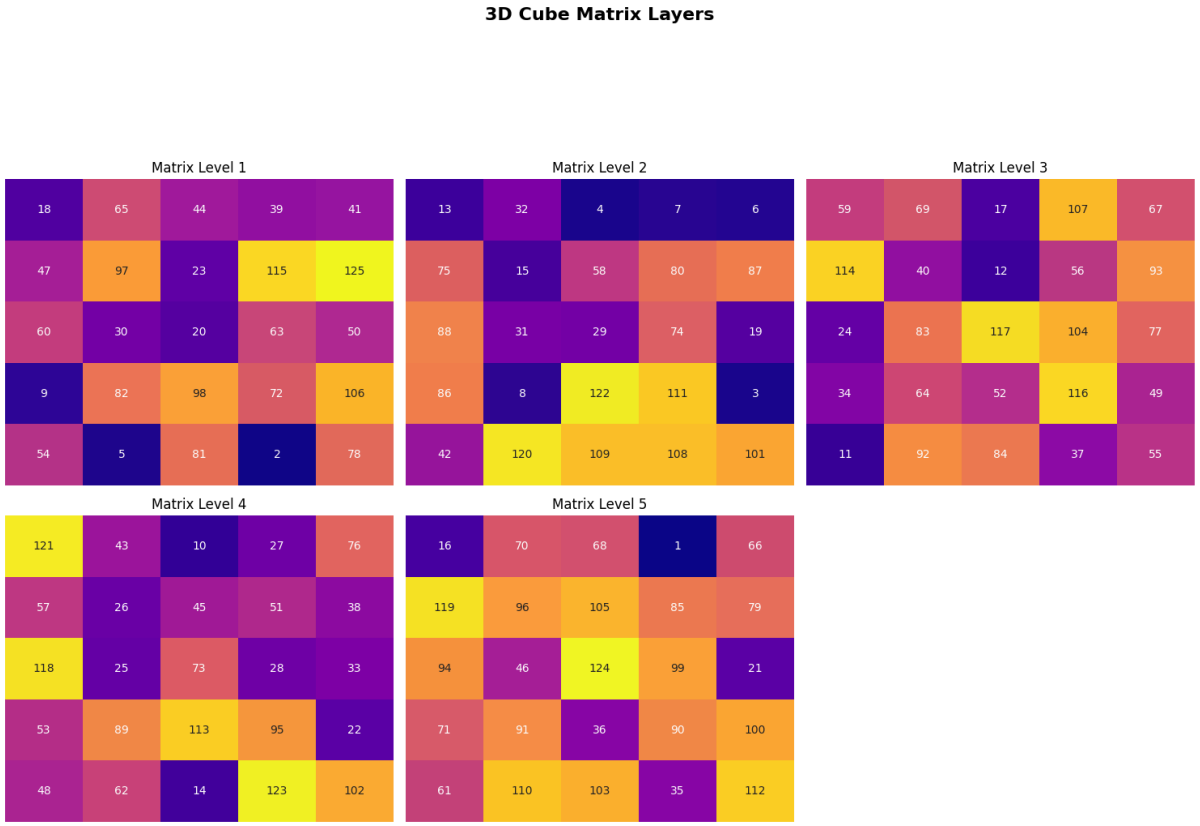
Jumlah Populasi: 200

Banyak Iterasi: 500

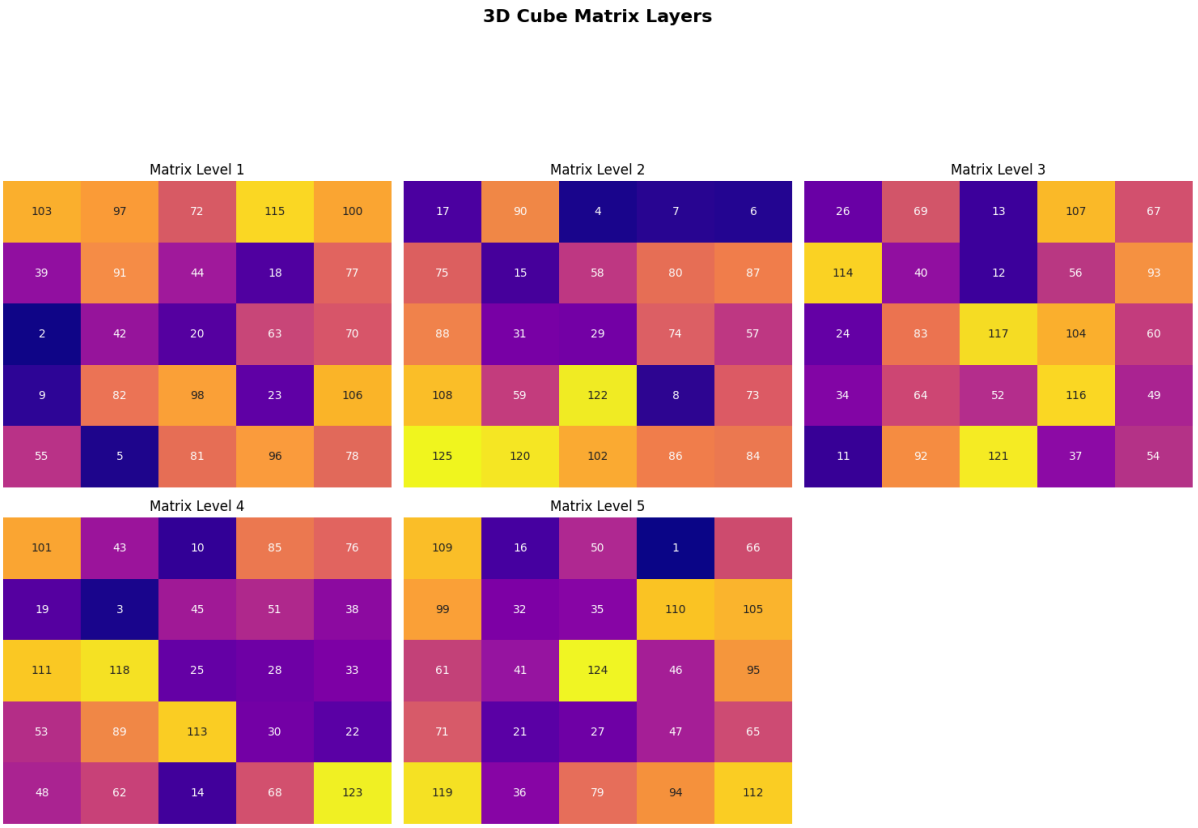
Durasi Proses Pencarian: 12 Seconds

2.2.6.3. Percobaan 3

State Awal

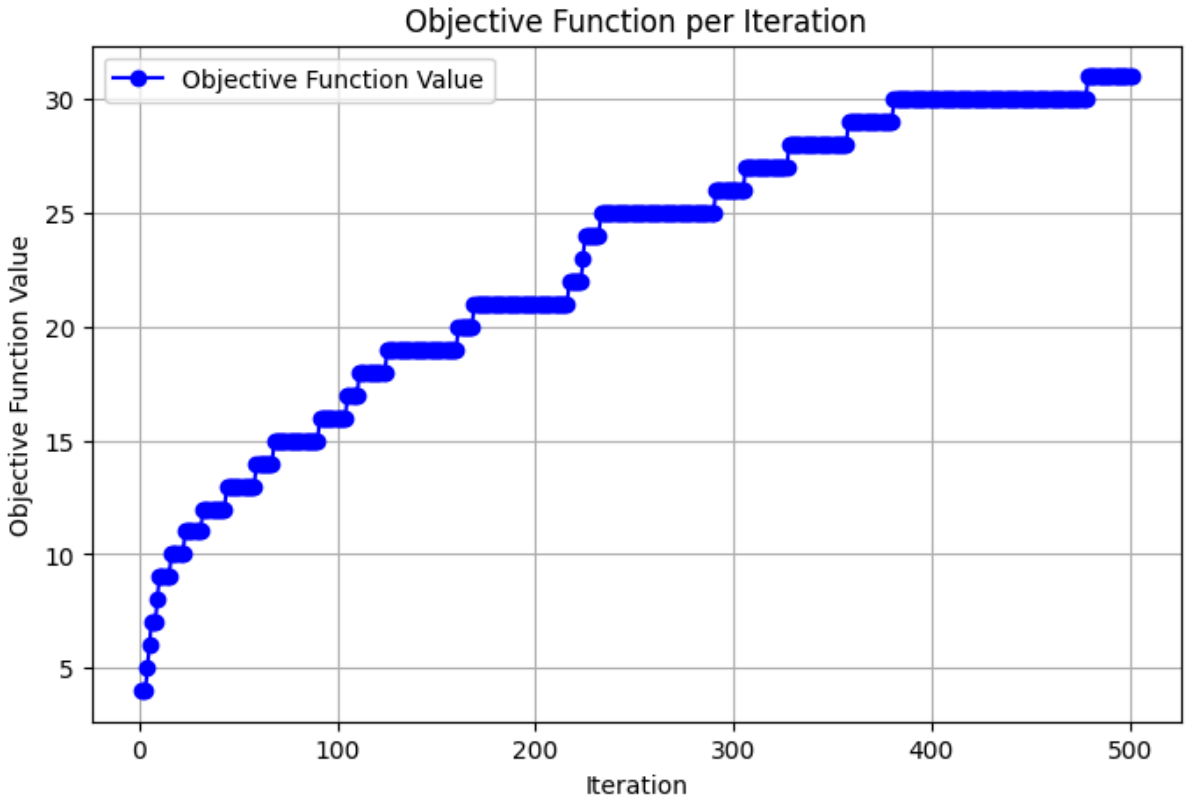


State Akhir

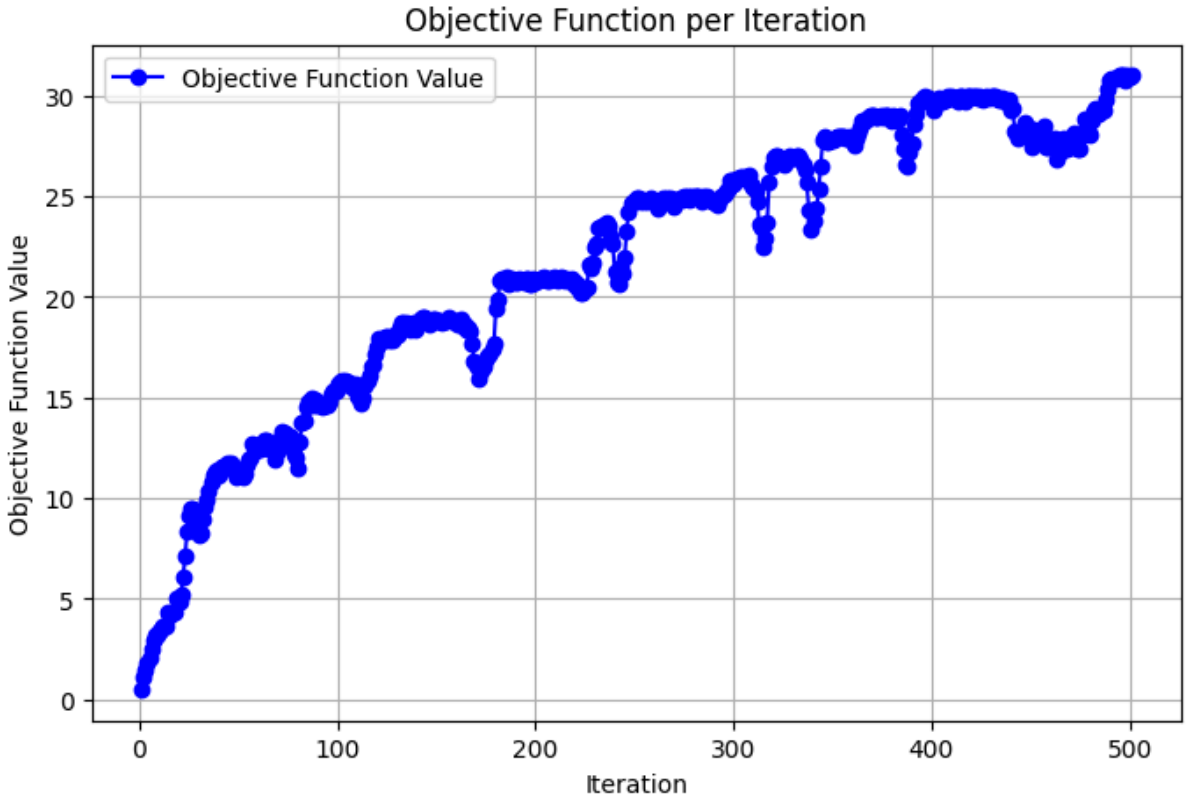


Nilai Objective Function: 31

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



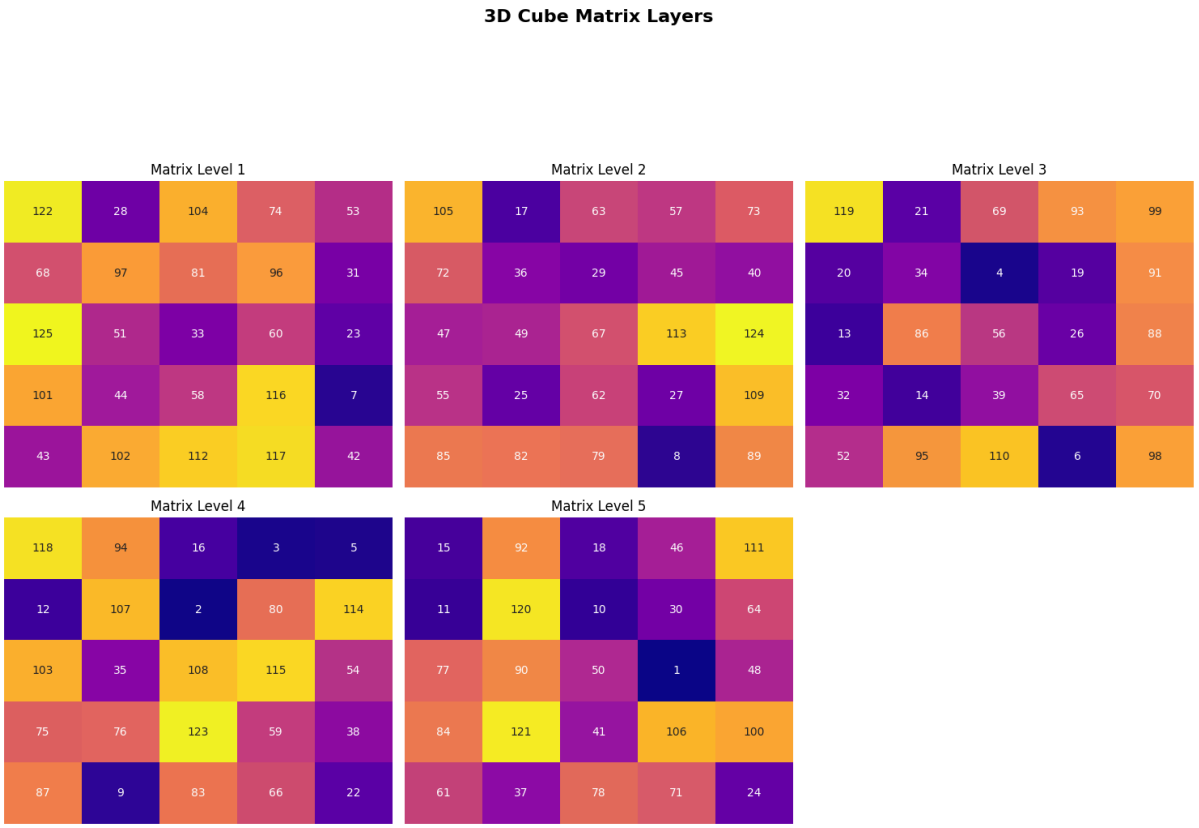
Jumlah Populasi: 200

Banyak Iterasi: 500

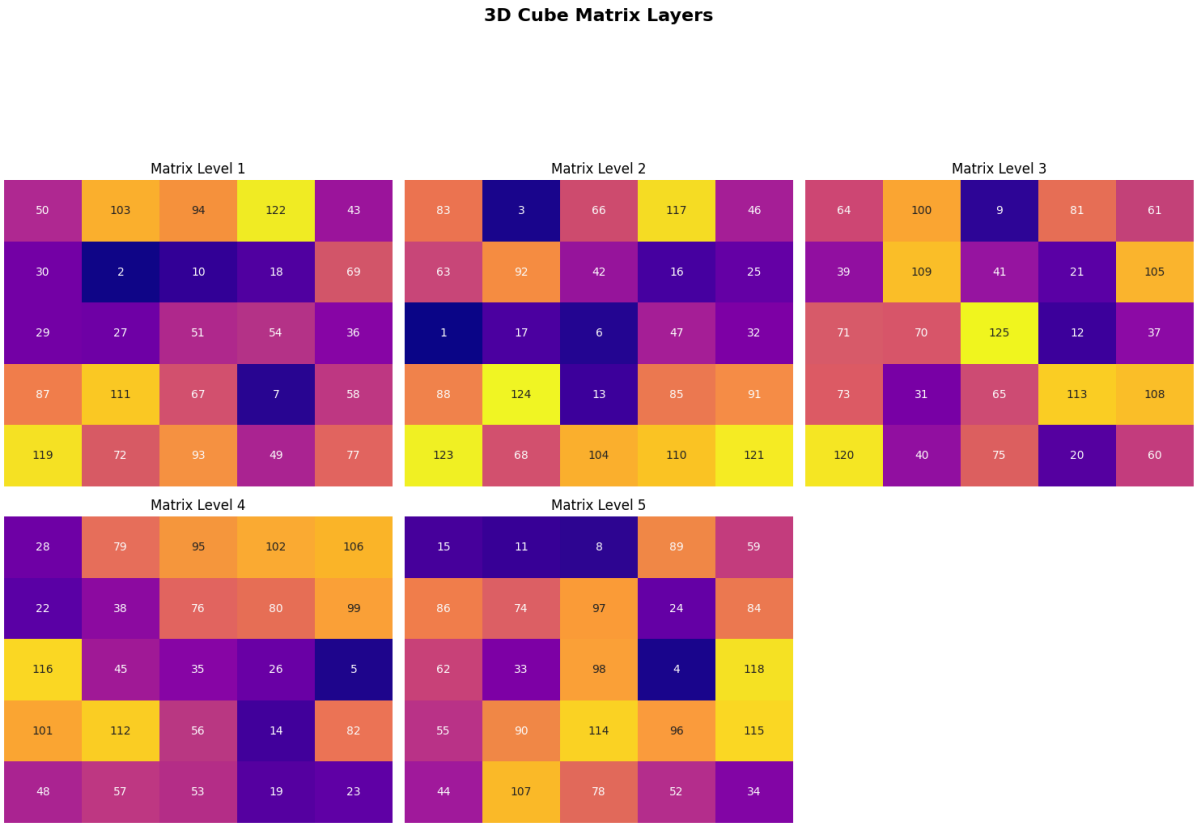
Durasi Proses Pencarian: 1.5 Second

2.2.6.4. Percobaan 4

State Awal

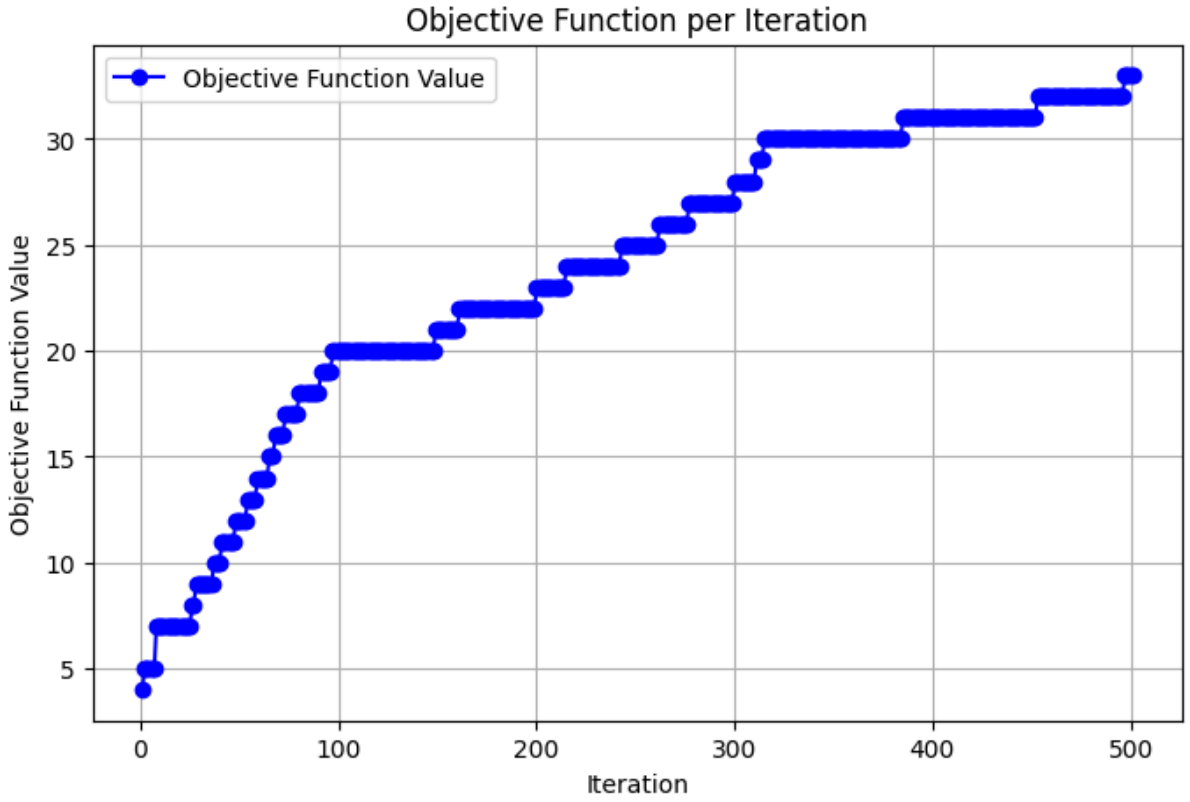


State Akhir

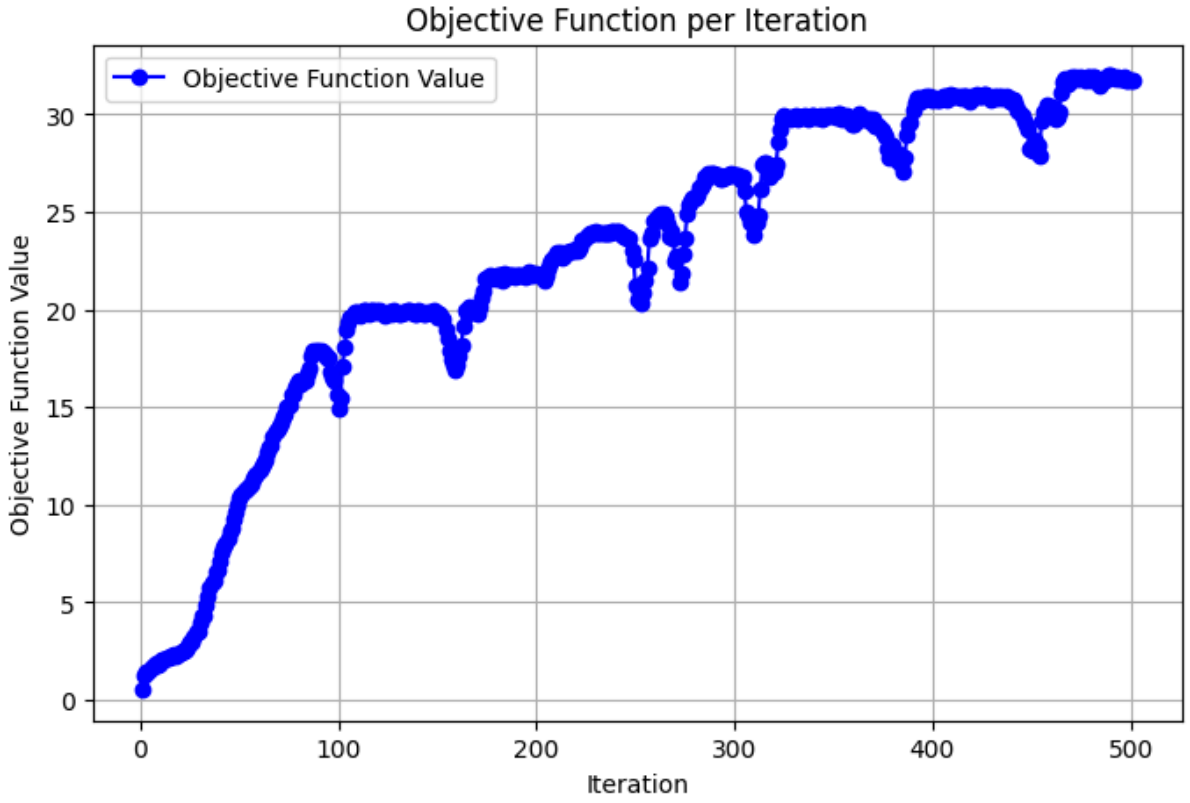


Nilai Objective Function: 33

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



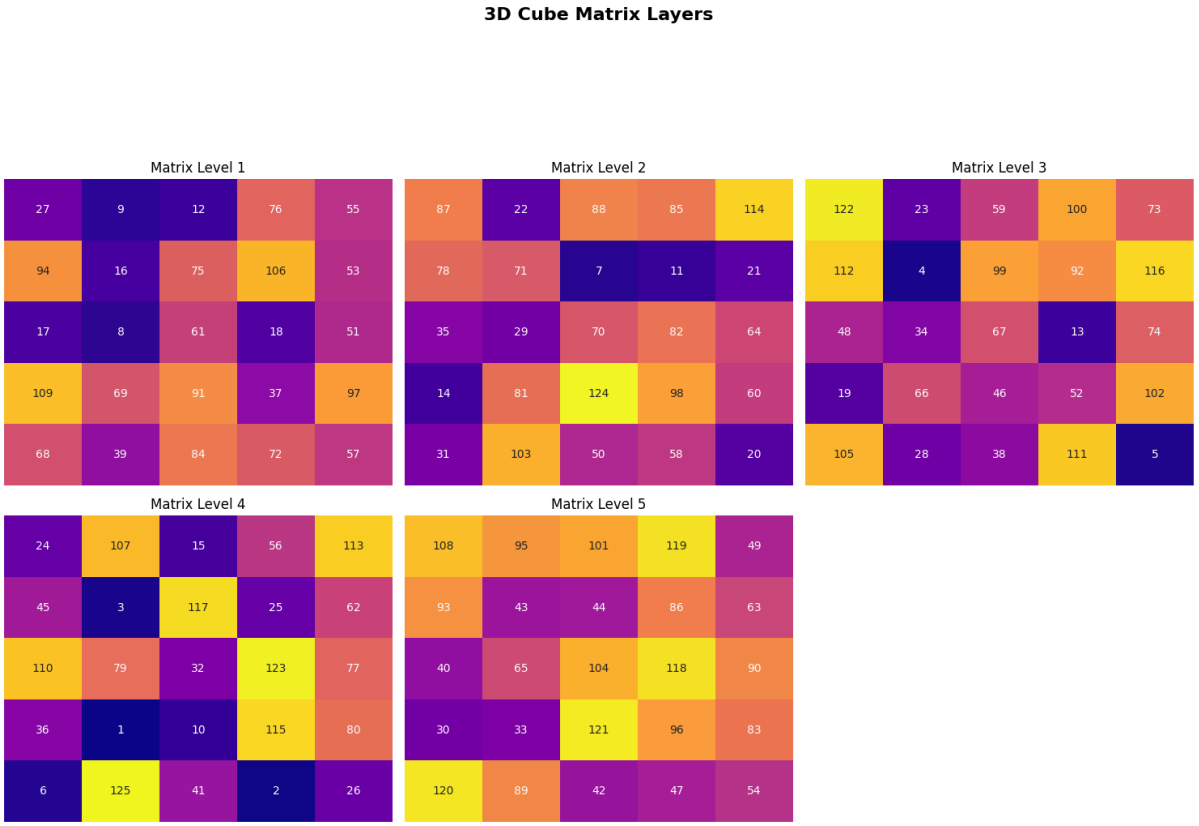
Jumlah Populasi: 400

Banyak Iterasi: 500

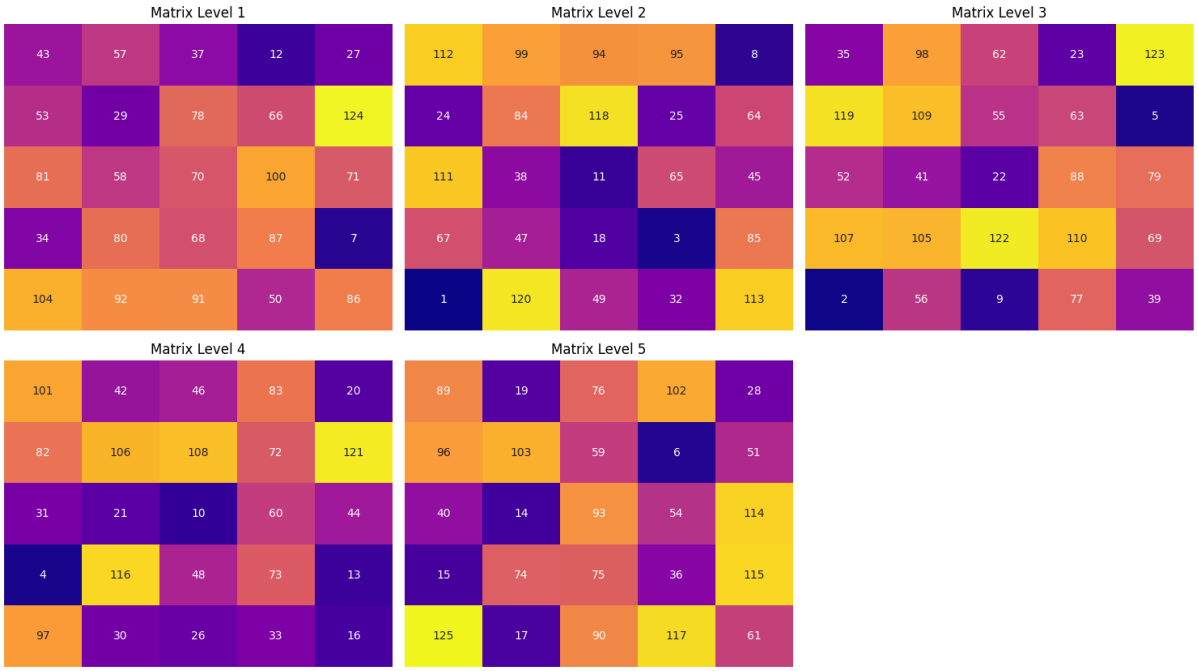
Durasi Proses Pencarian: 32 Second

2.2.6.5. Percobaan 5

State Awal

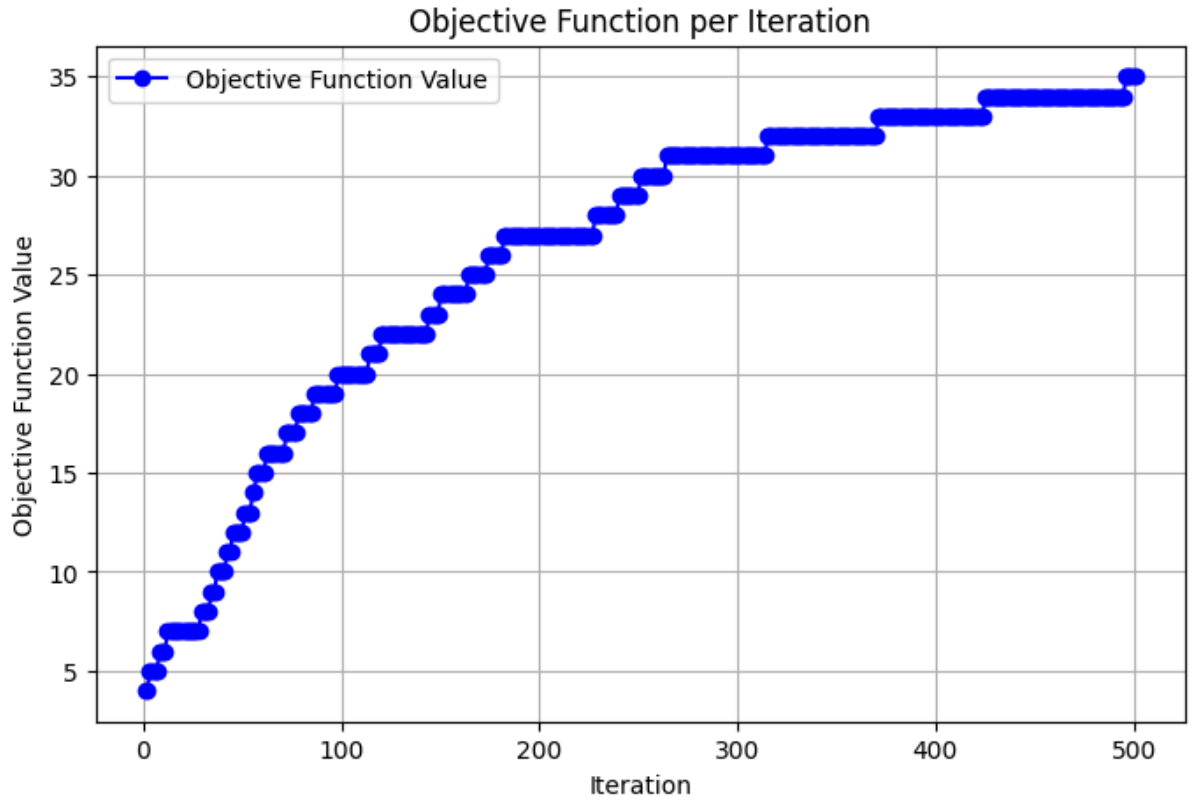


State Akhir

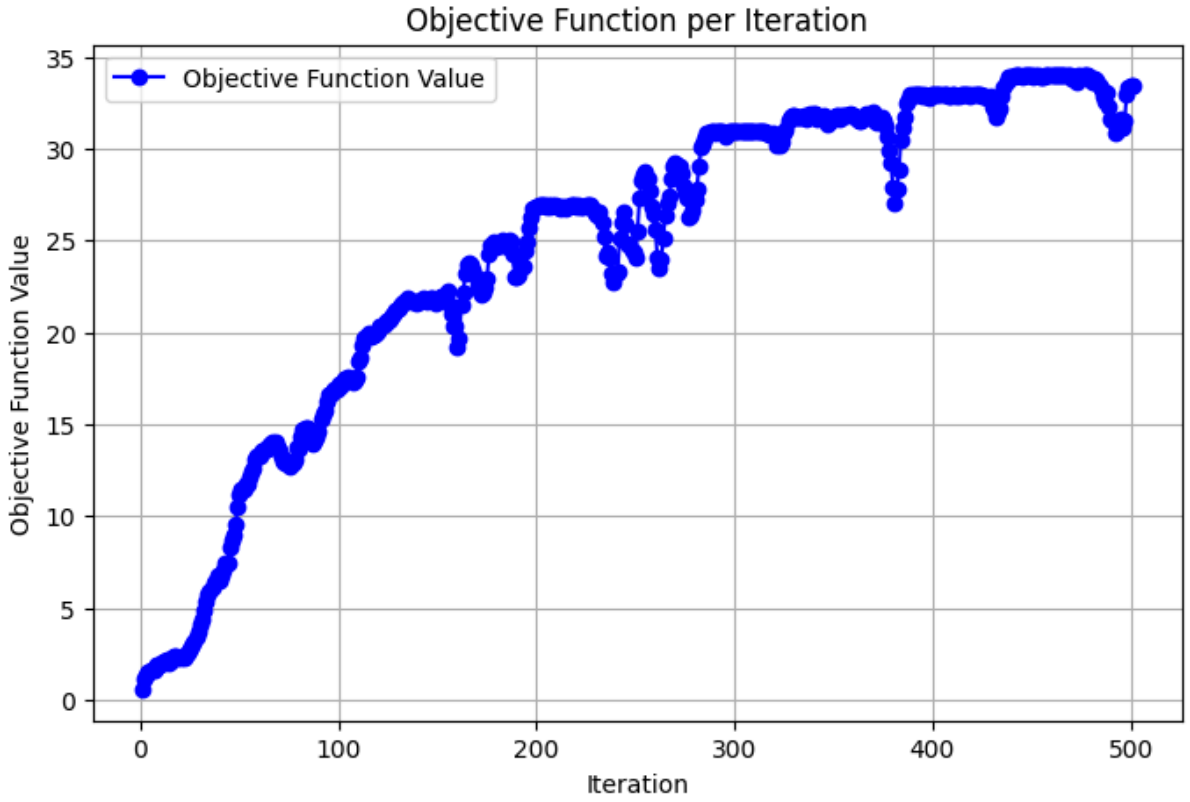


Nilai Objective Function: 35

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



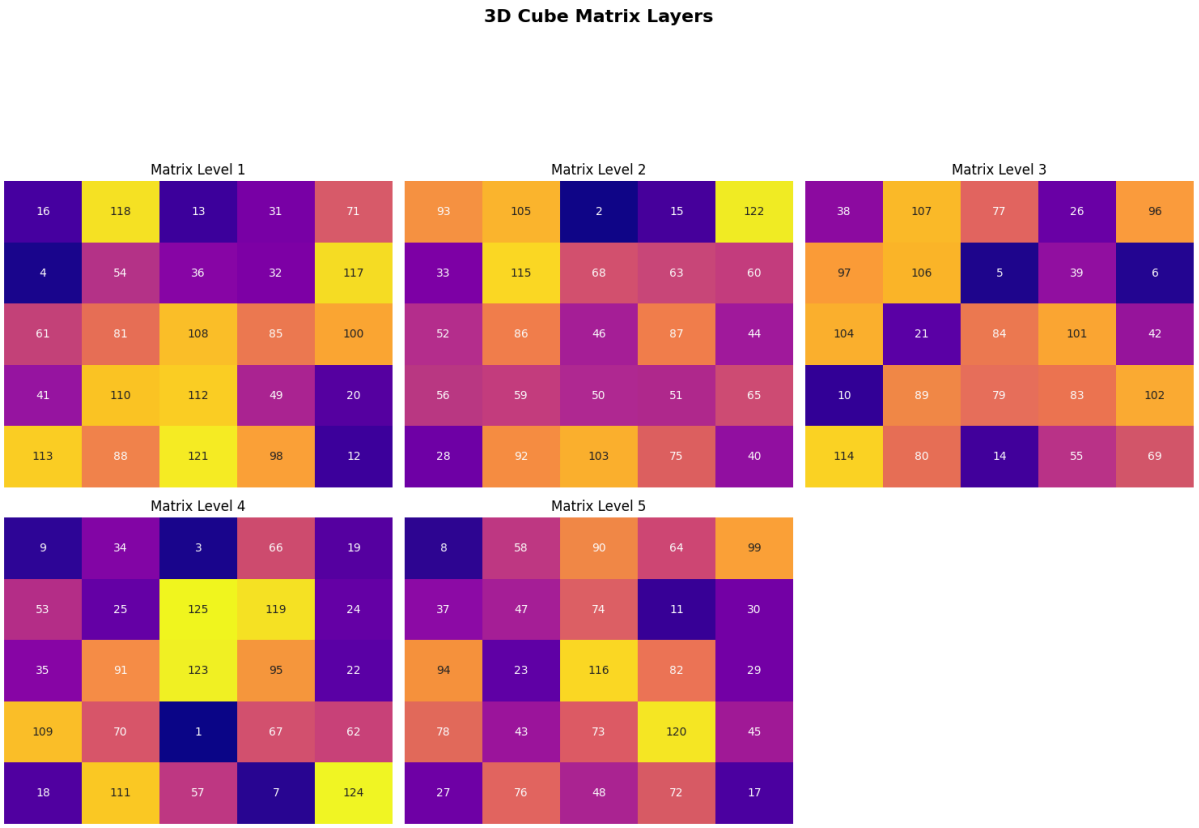
Jumlah Populasi: 400

Banyak Iterasi: 500

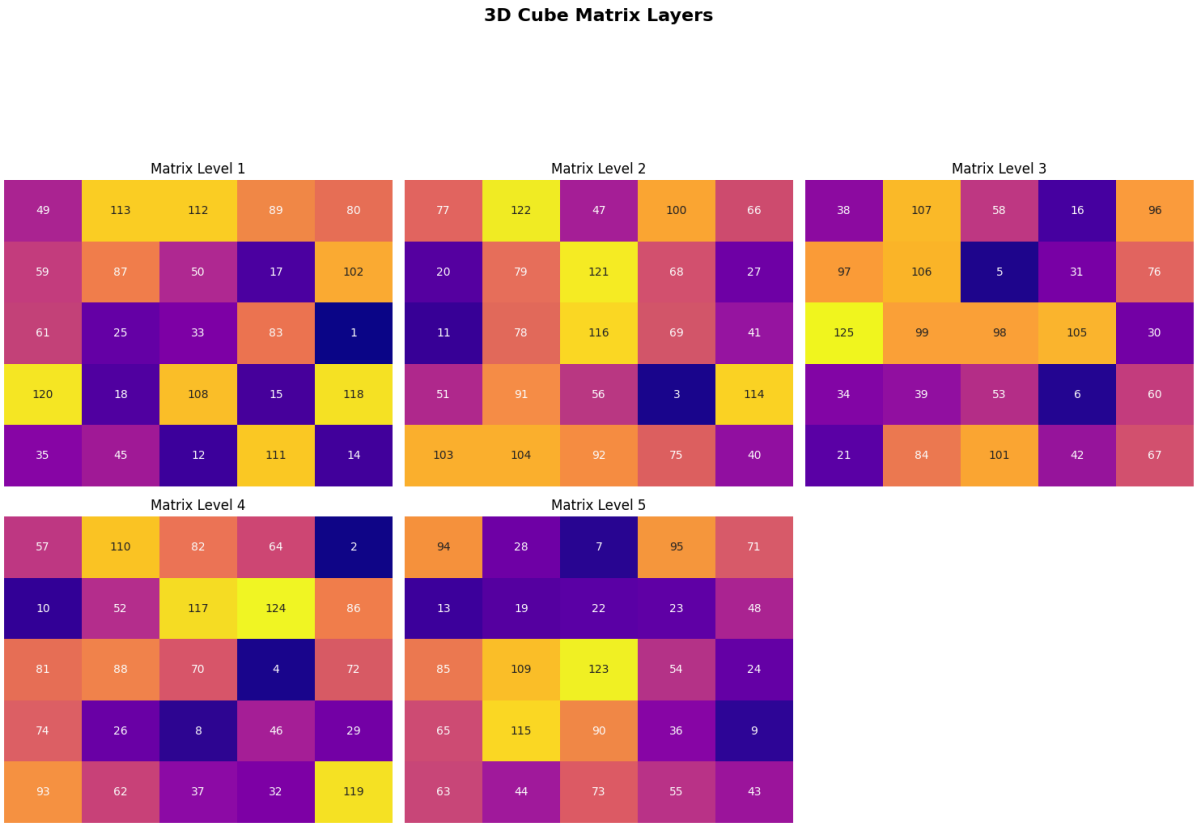
Durasi Proses Pencarian: 37 Second

2.2.6.6. Percobaan 6

State Awal

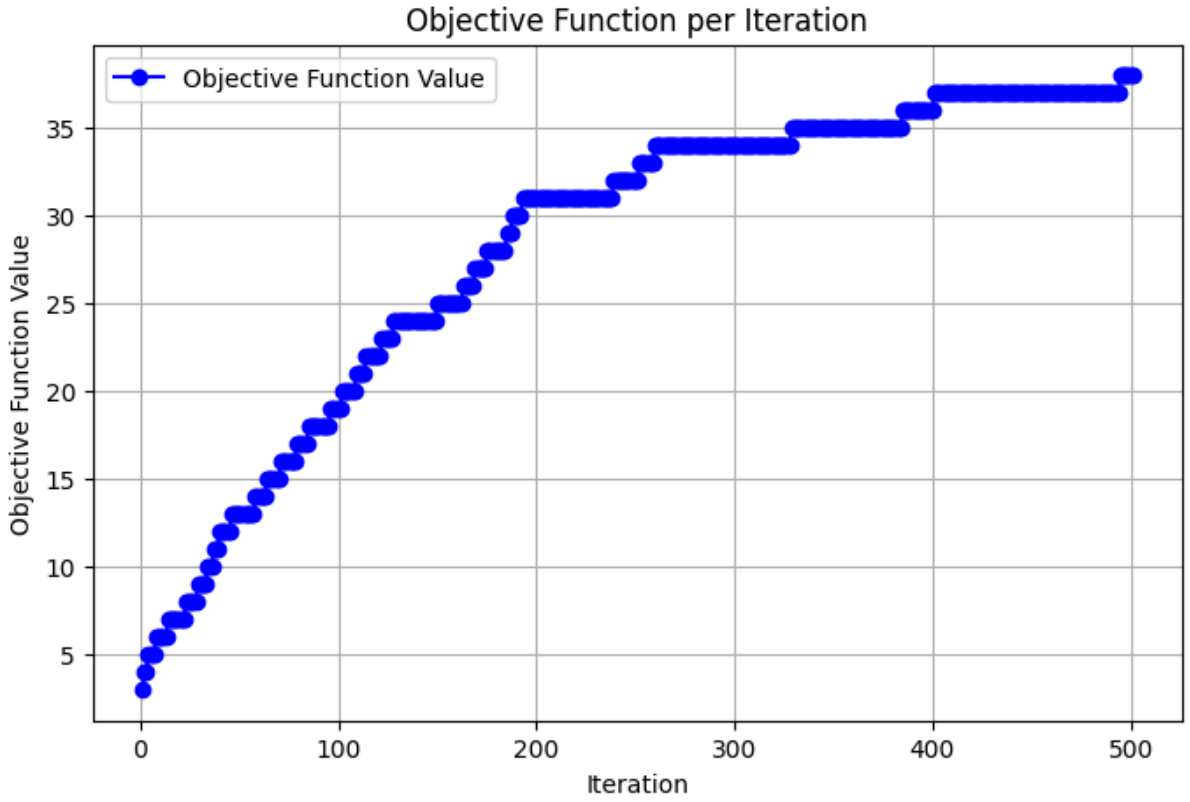


State Akhir

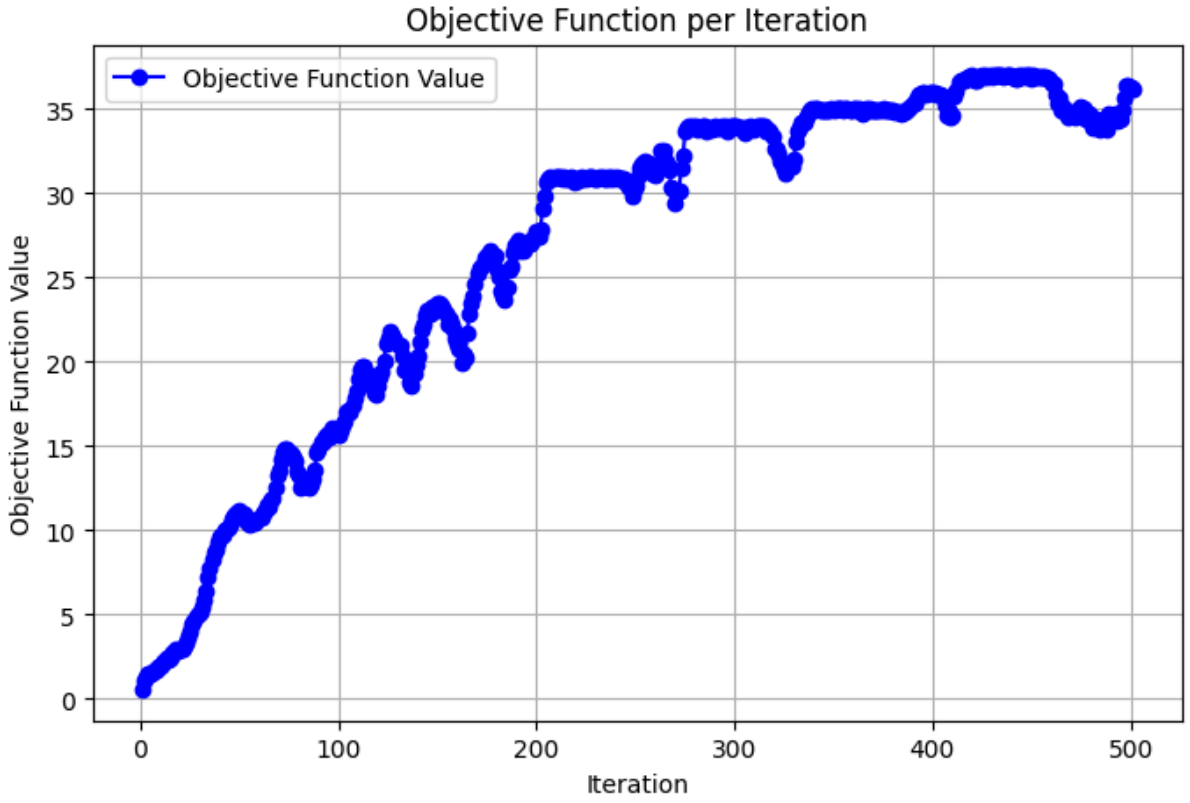


Nilai Objective Function: 38

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



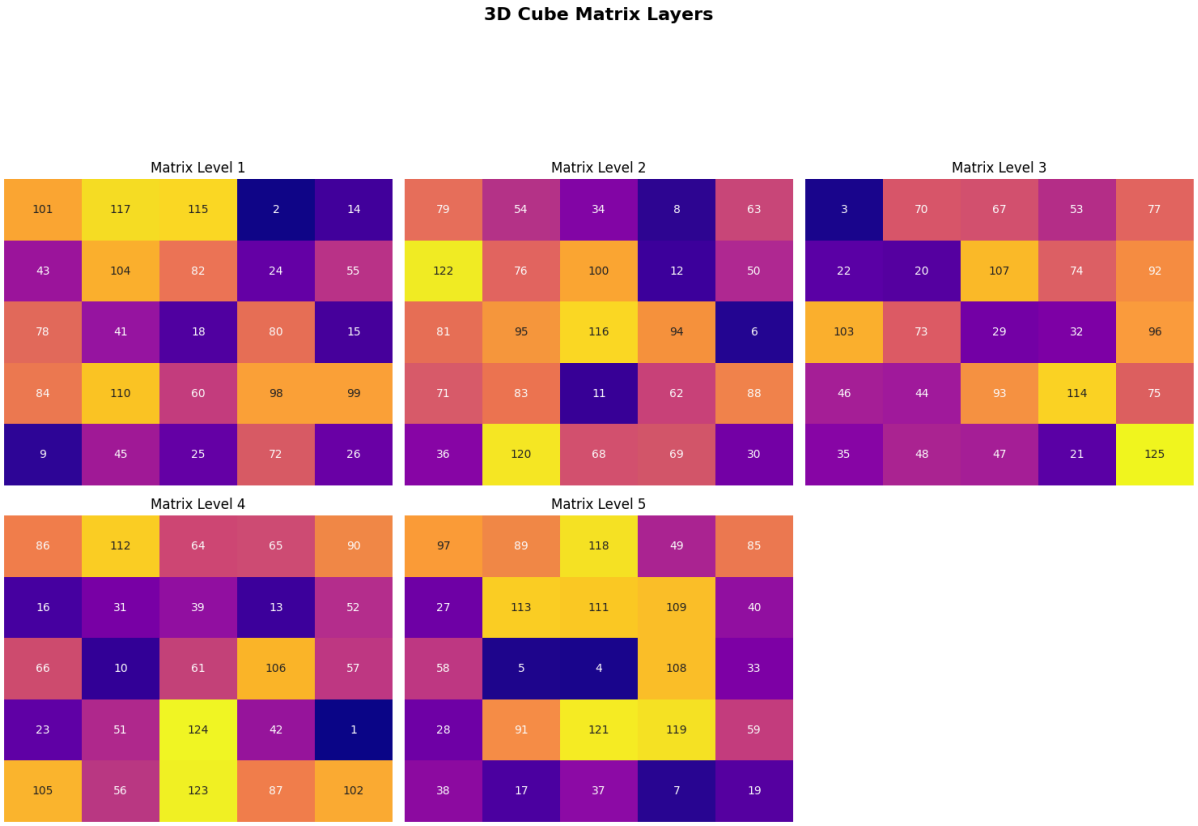
Jumlah Populasi: 400

Banyak Iterasi: 500

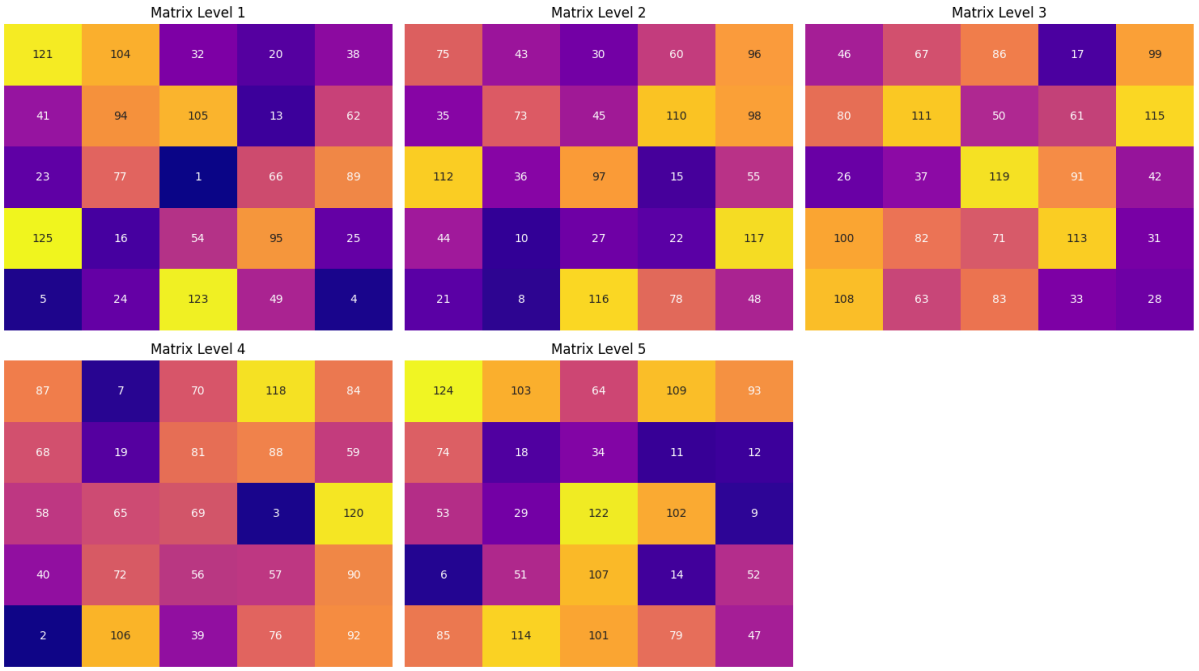
Durasi Proses Pencarian: 41 Second

2.2.6.7. Percobaan 7

State Awal

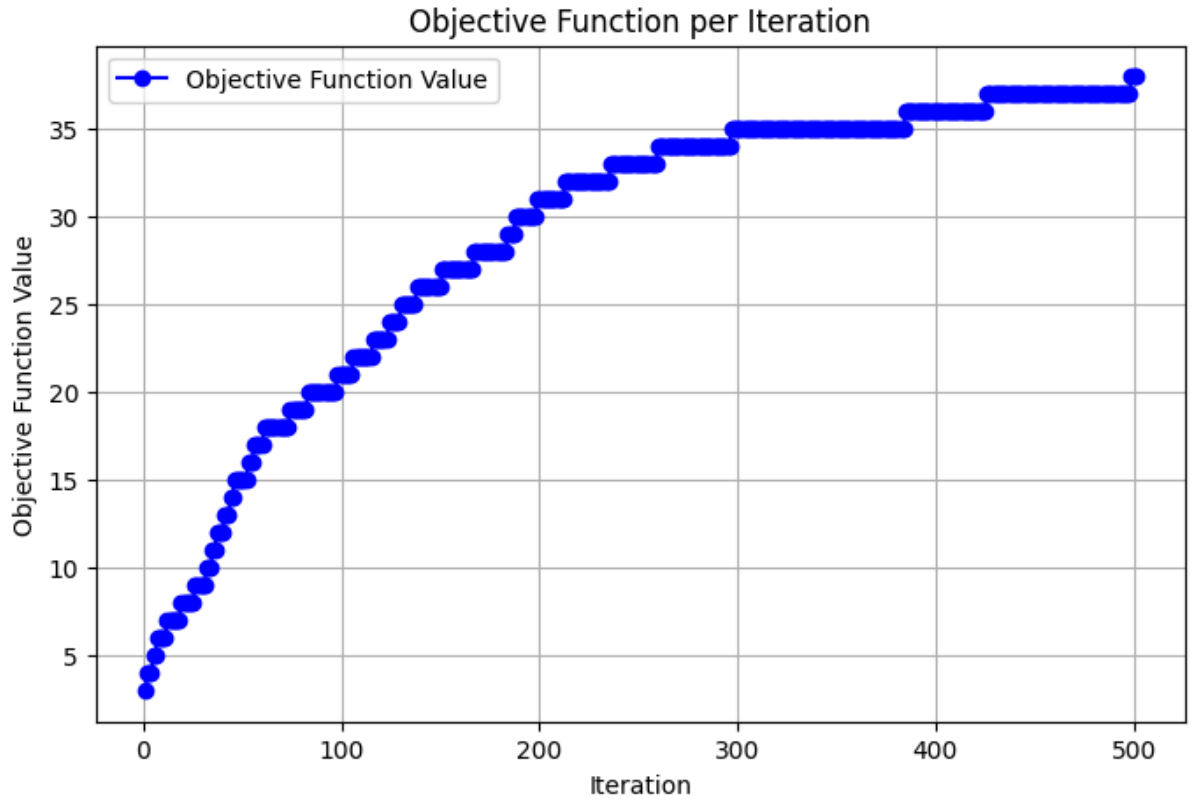


State Akhir

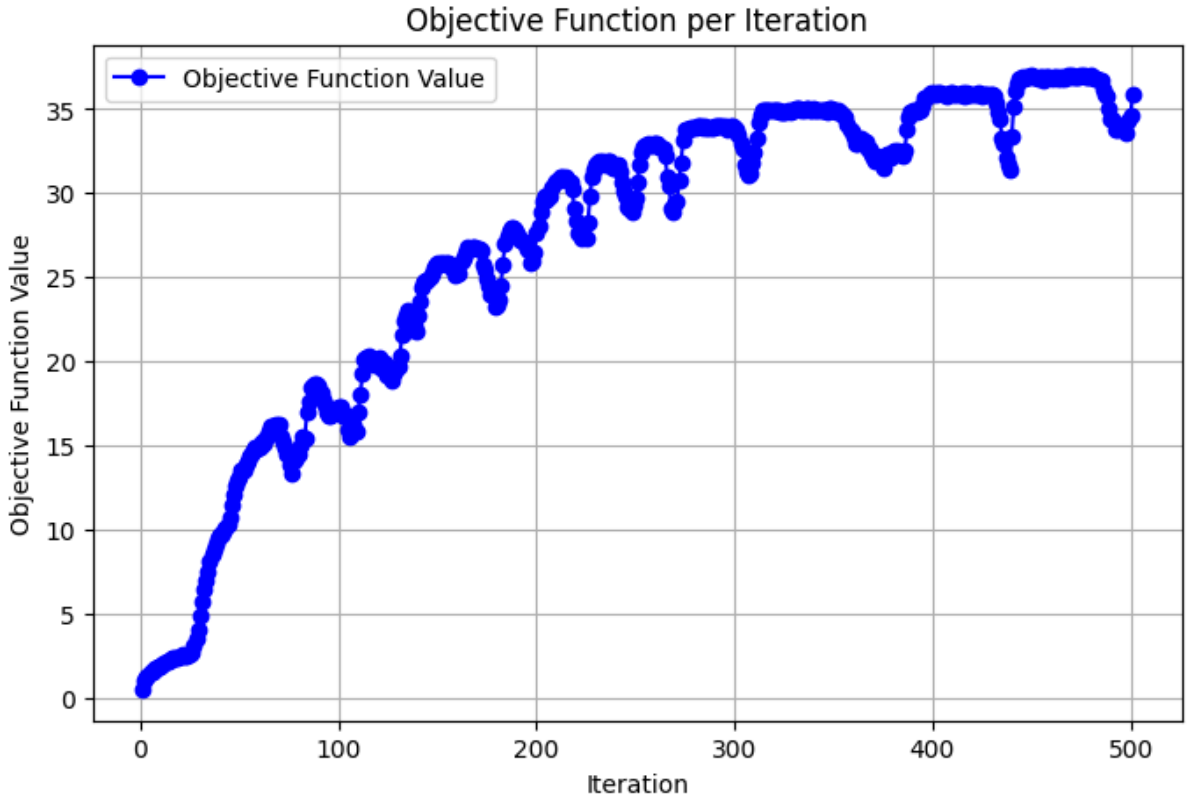


Nilai Objective Function: 38

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



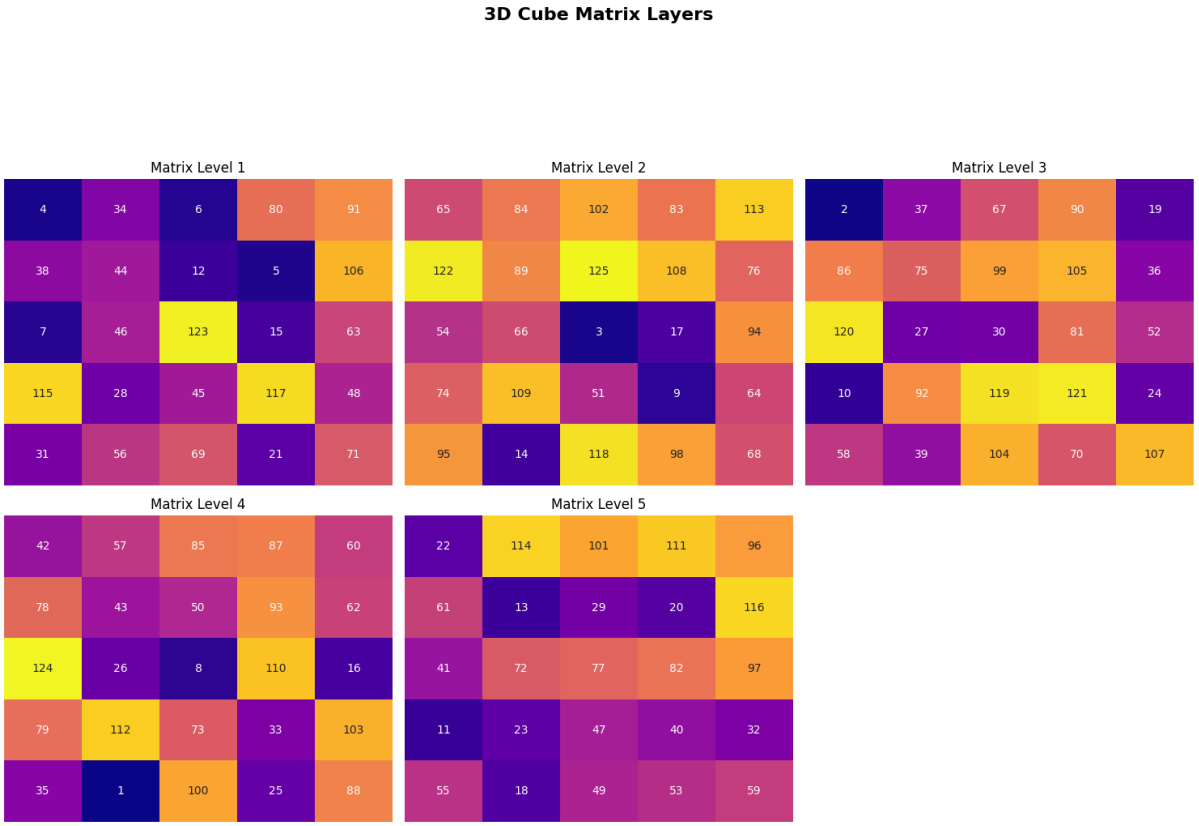
Jumlah Populasi: 600

Banyak Iterasi: 500

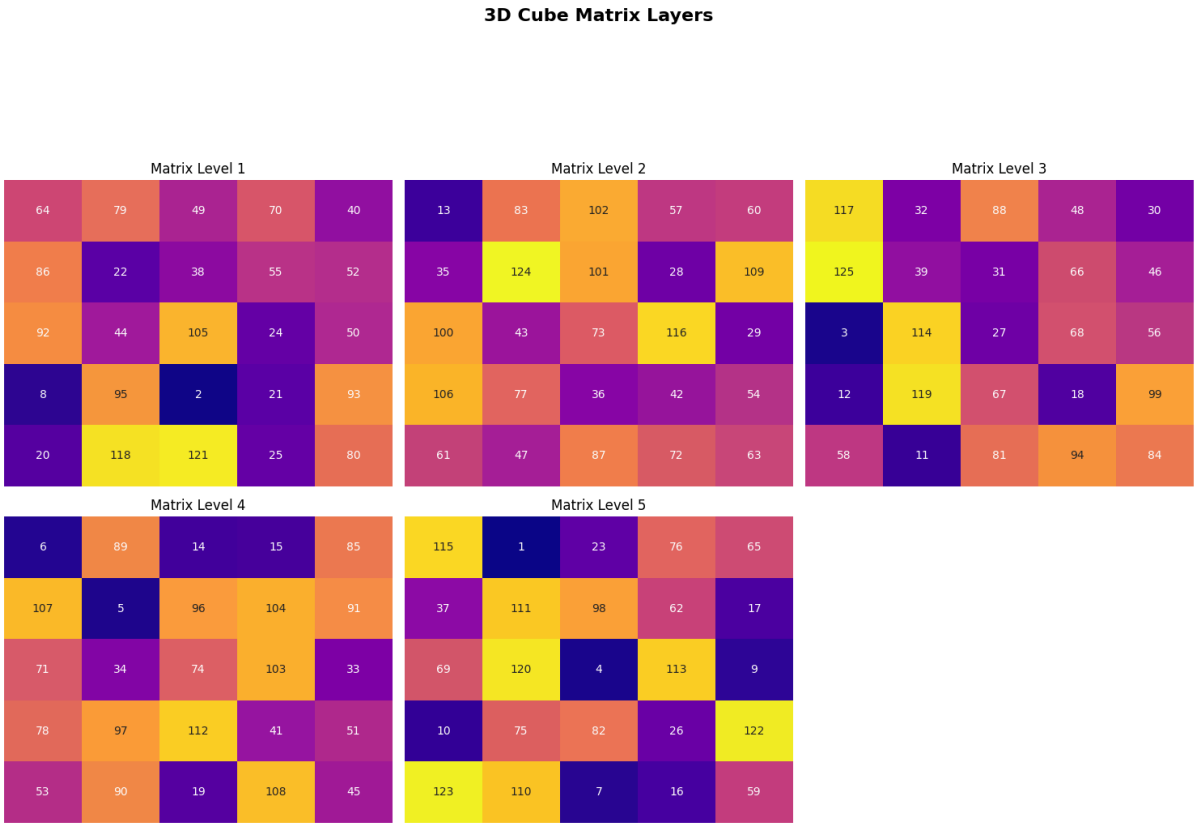
Durasi Proses Pencarian: 69 Second

2.2.6.8. Percobaan 8

State Awal

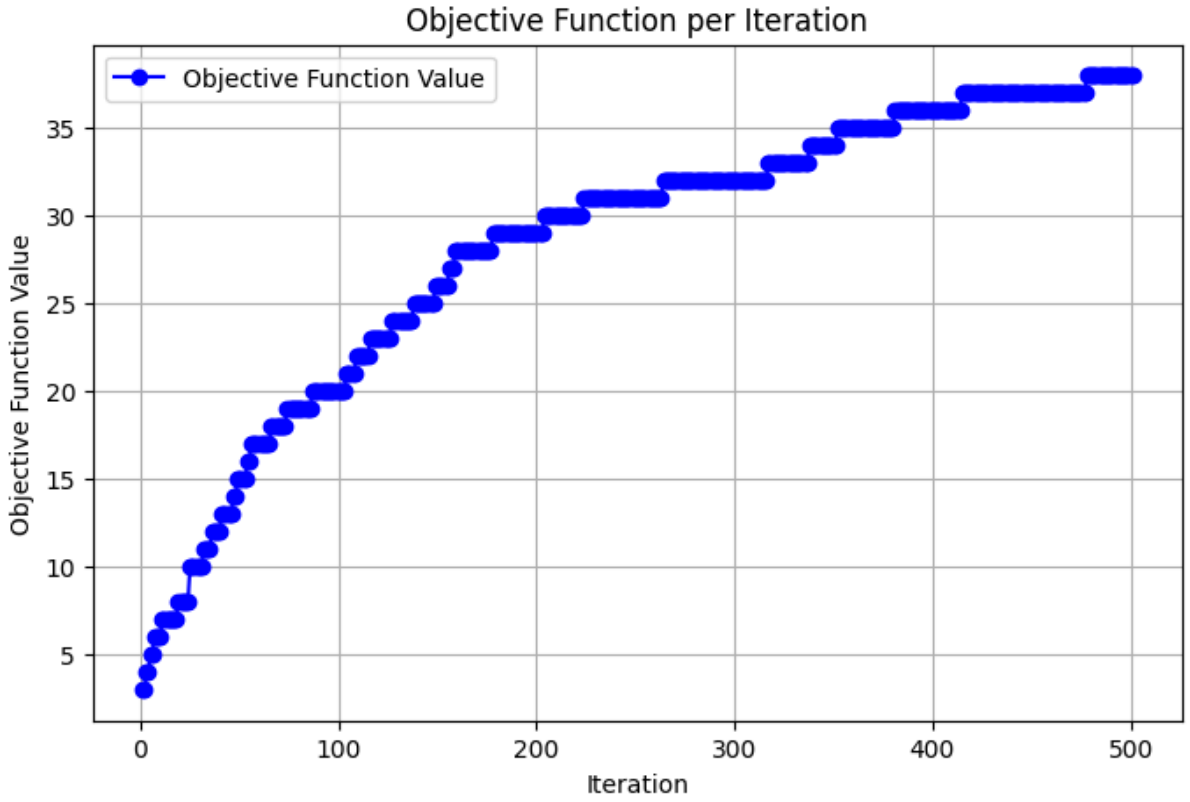


State Akhir

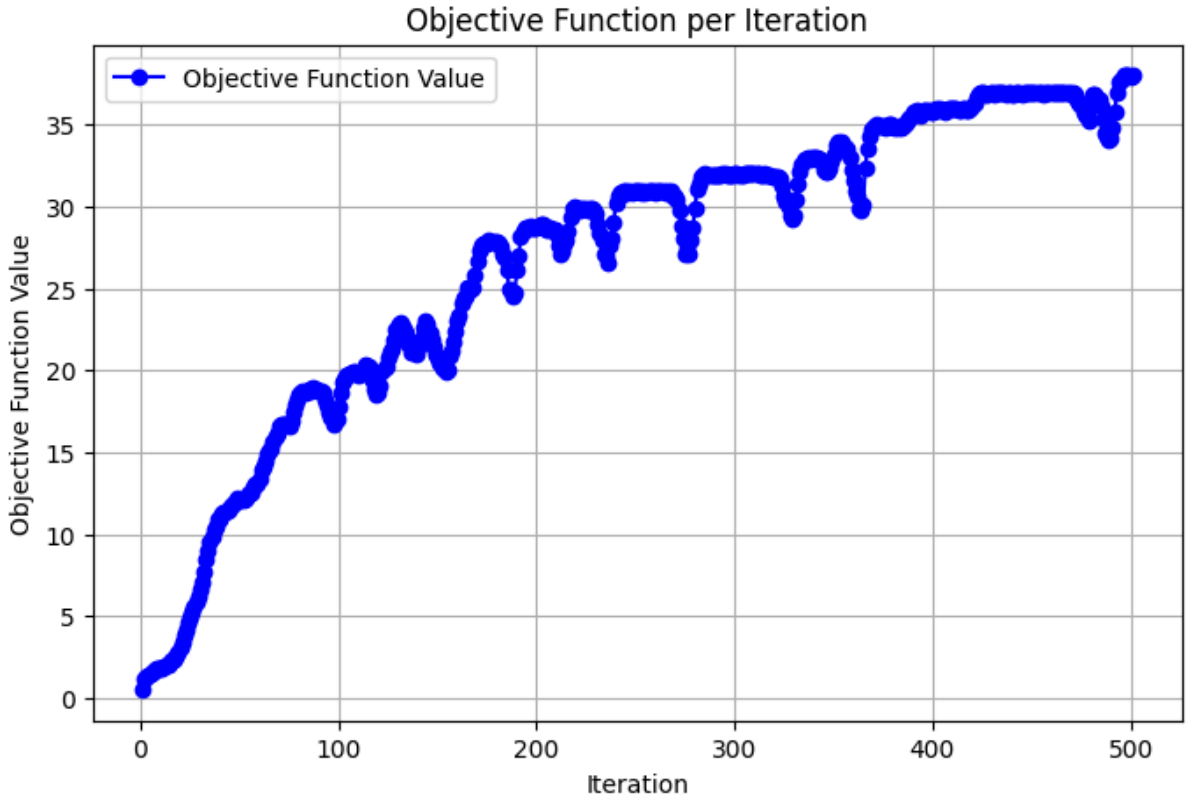


Nilai Objective Function: 38

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



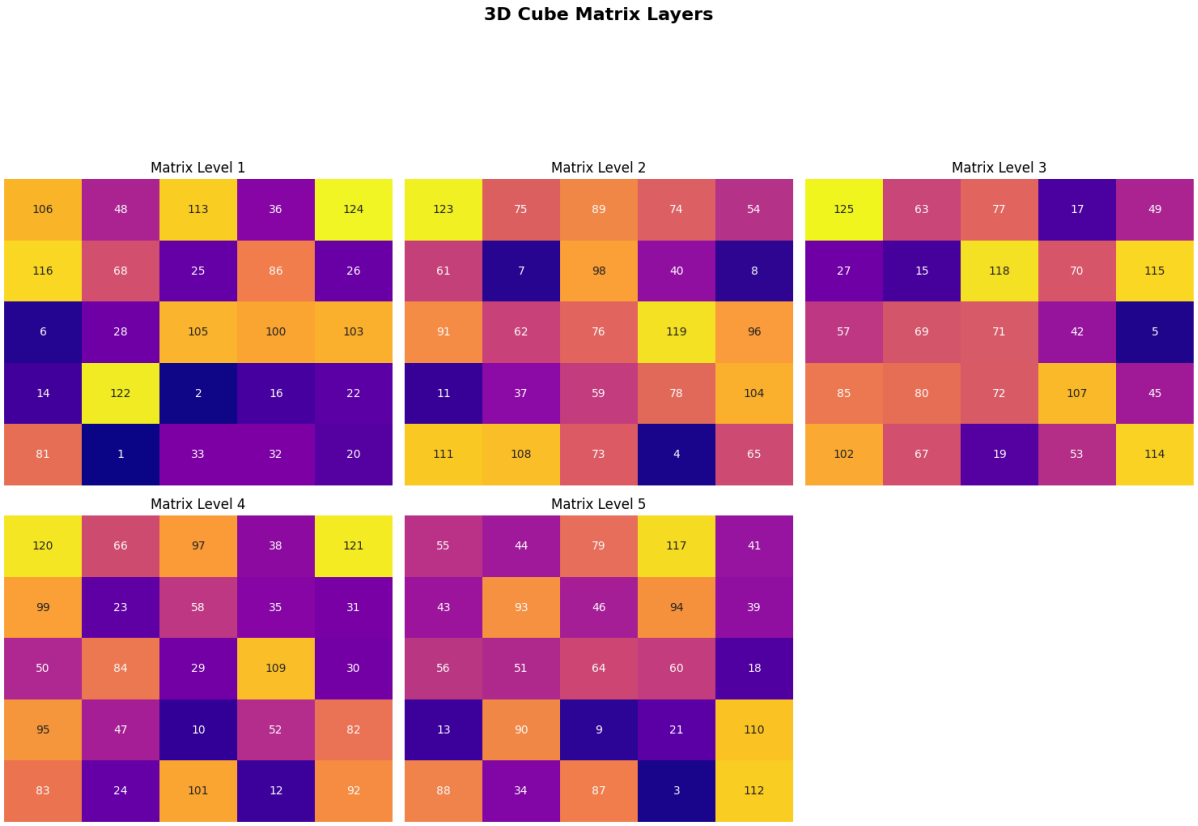
Jumlah Populasi: 600

Banyak Iterasi: 500

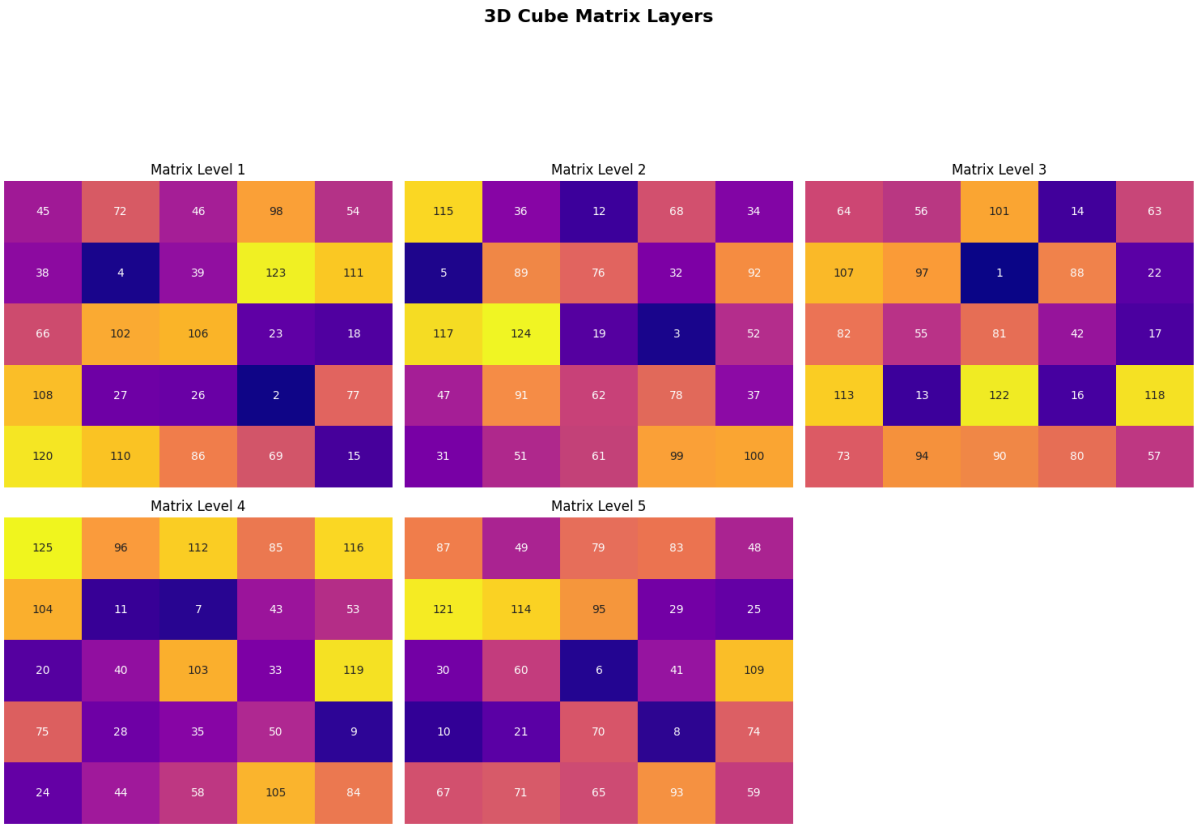
Durasi Proses Pencarian: 72 Second

2.2.6.9. Percobaan 9

State Awal

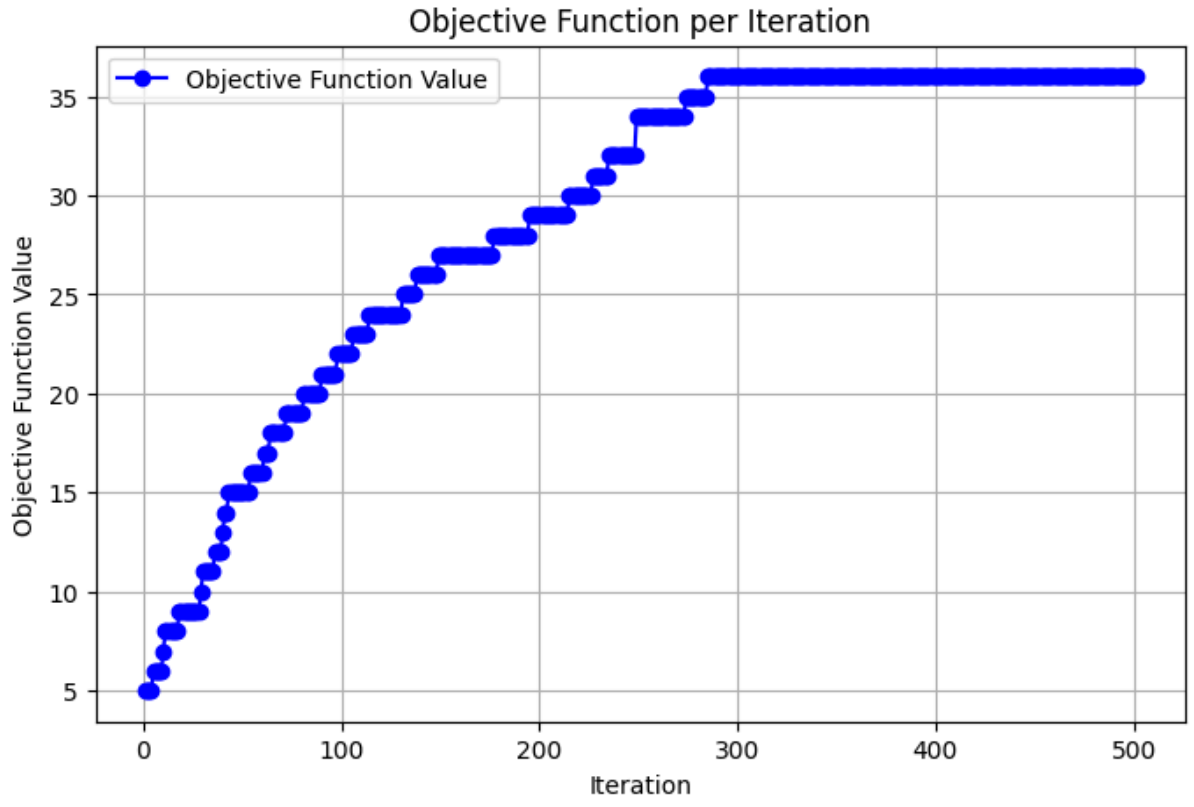


State Akhir

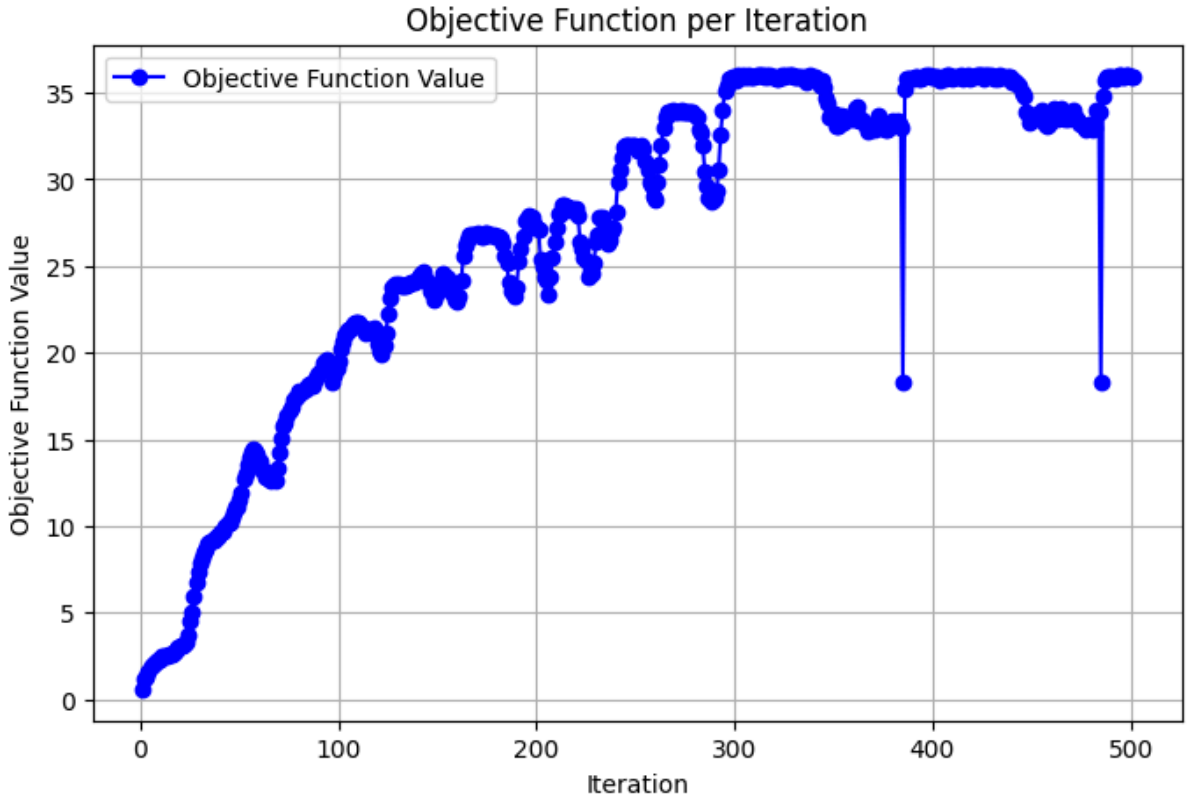


Nilai Objective Function: 36

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



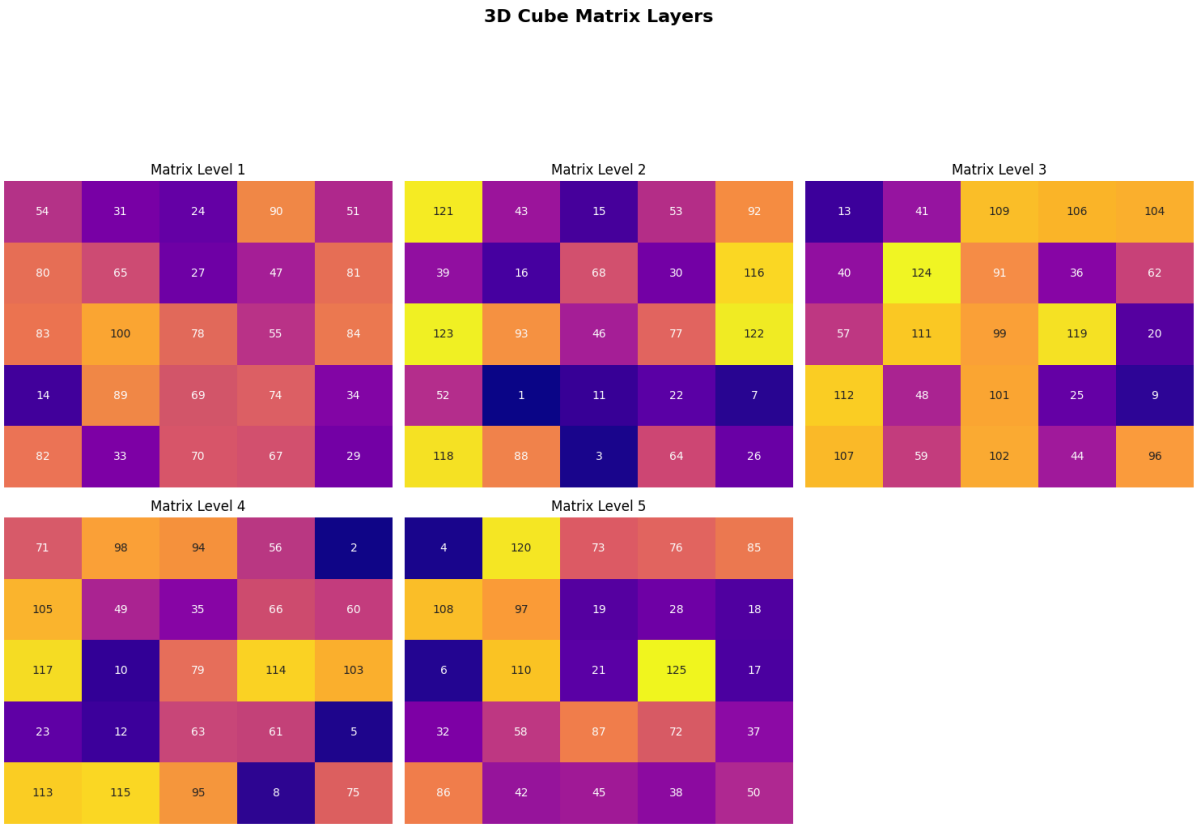
Jumlah Populasi: 600

Banyak Iterasi: 500

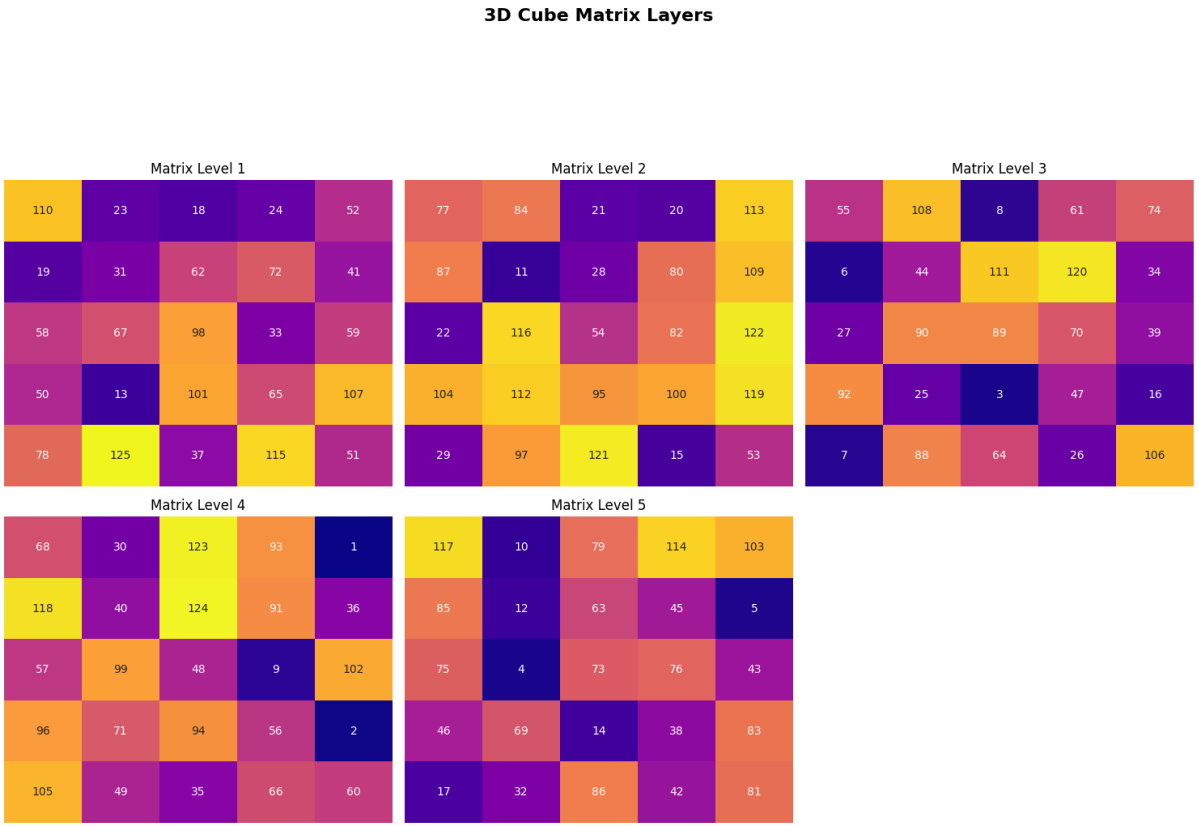
Durasi Proses Pencarian: 73 Second

2.2.6.10. Percobaan 10

State Awal

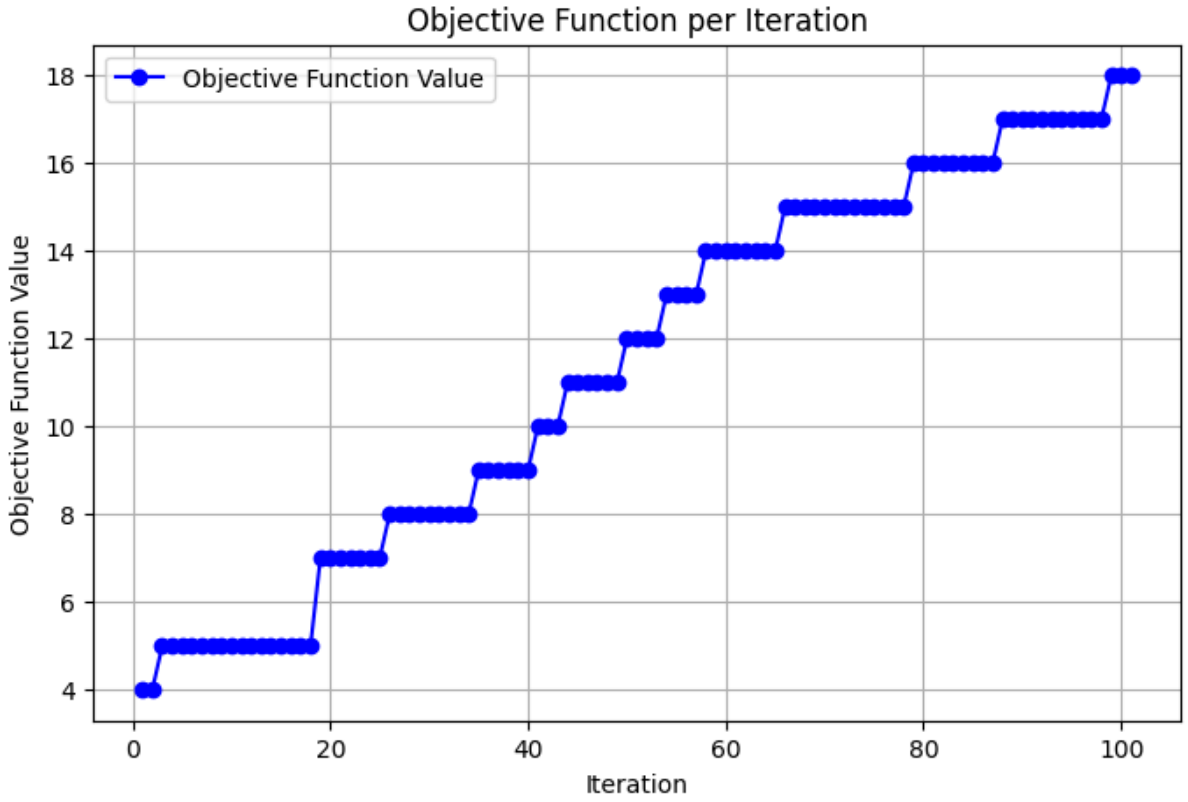


State Akhir

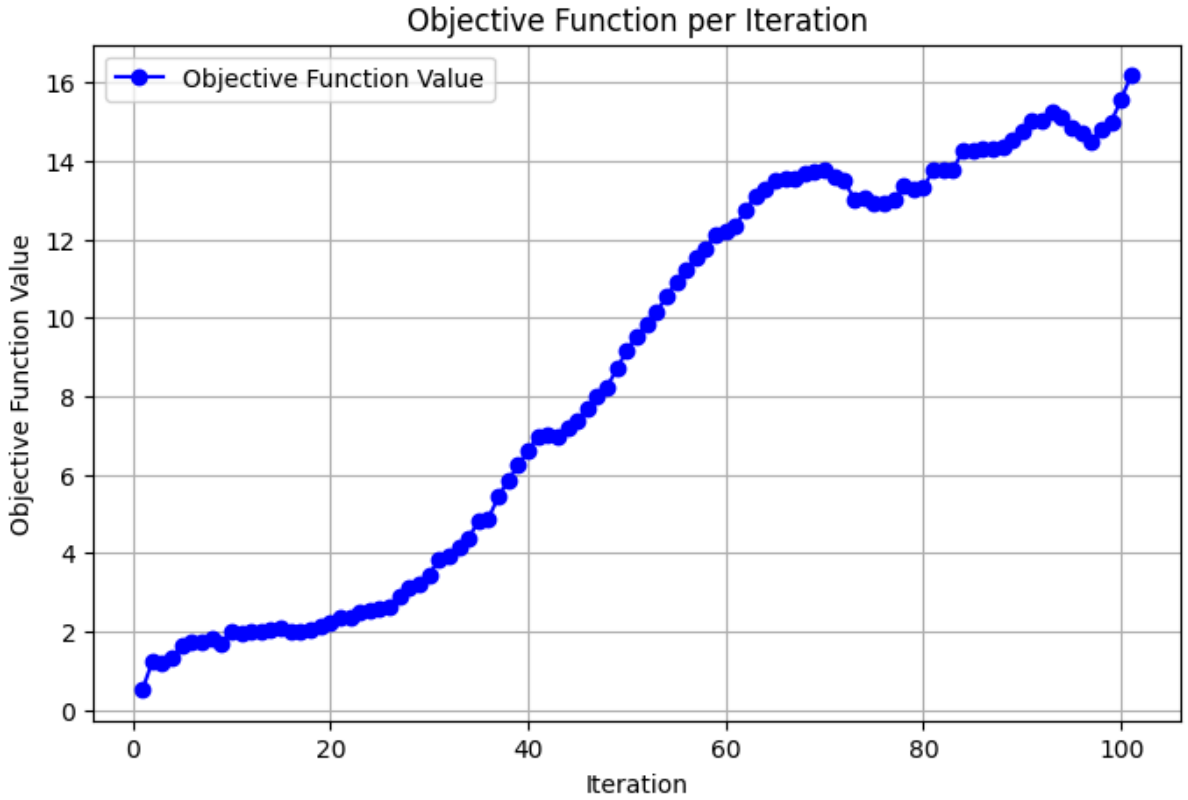


Nilai Objective Function: 18

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



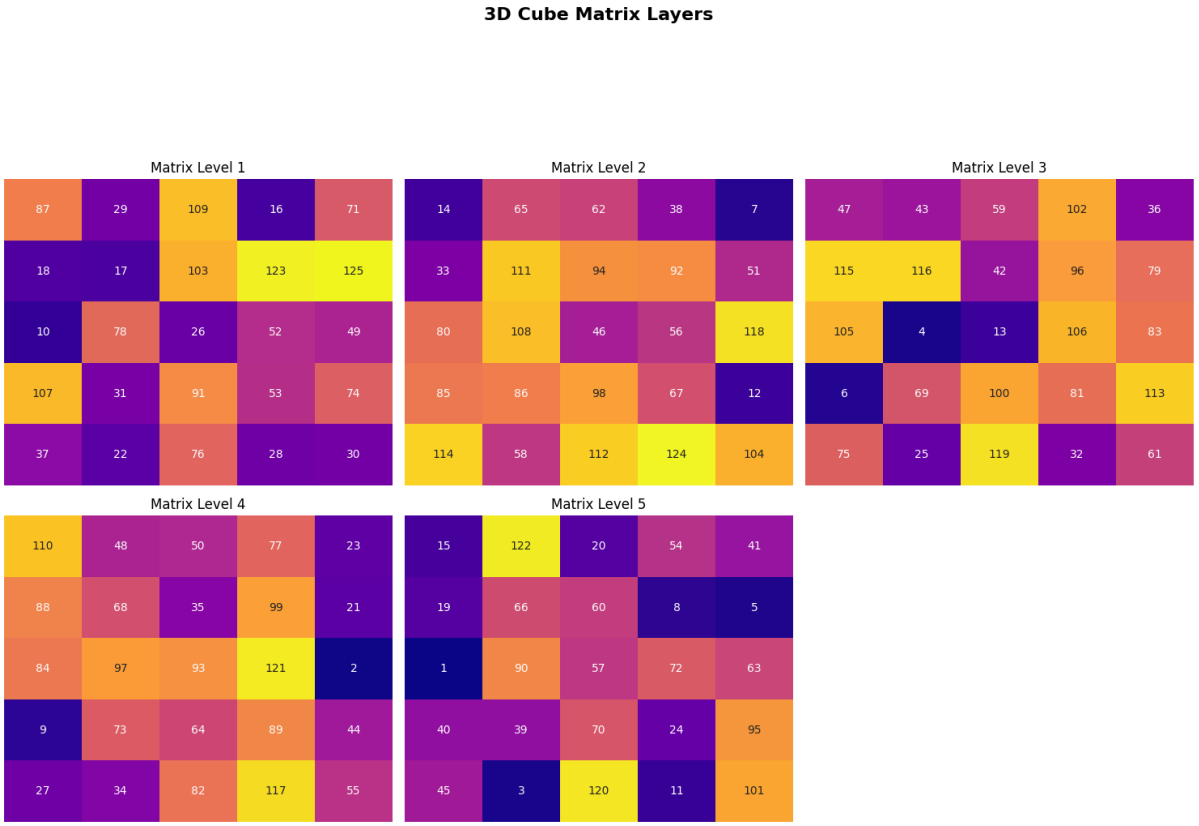
Jumlah Populasi: 300

Banyak Iterasi: 100

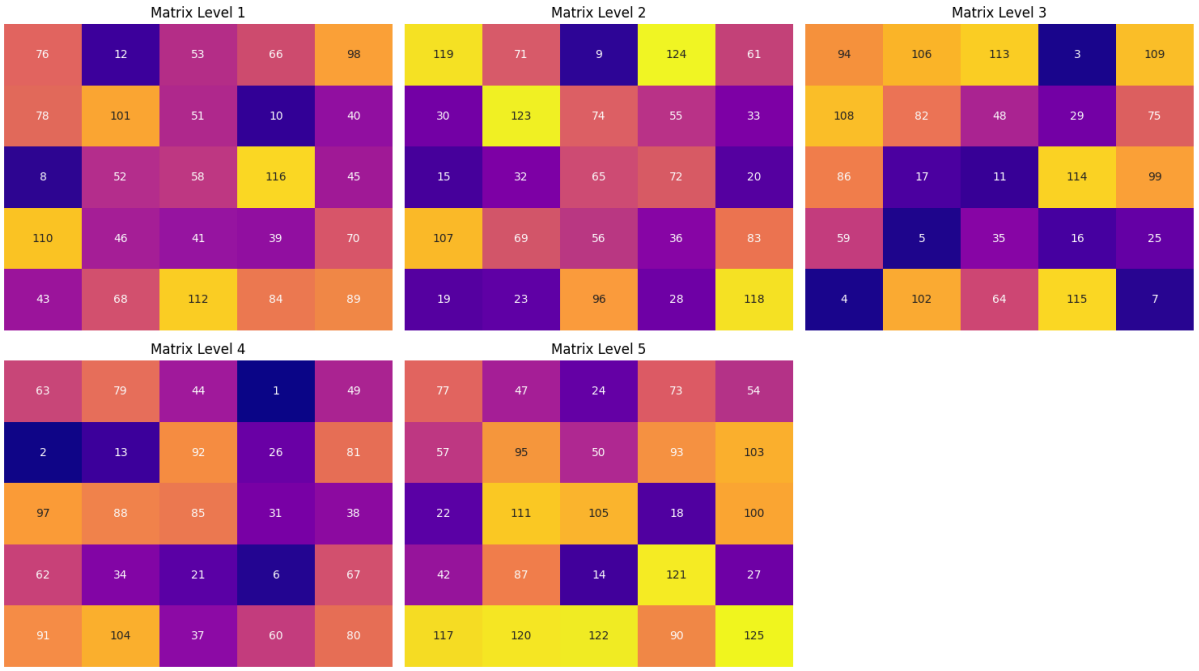
Durasi Proses Pencarian: 6 Second

2.2.6.11. Percobaan 11

State Awal

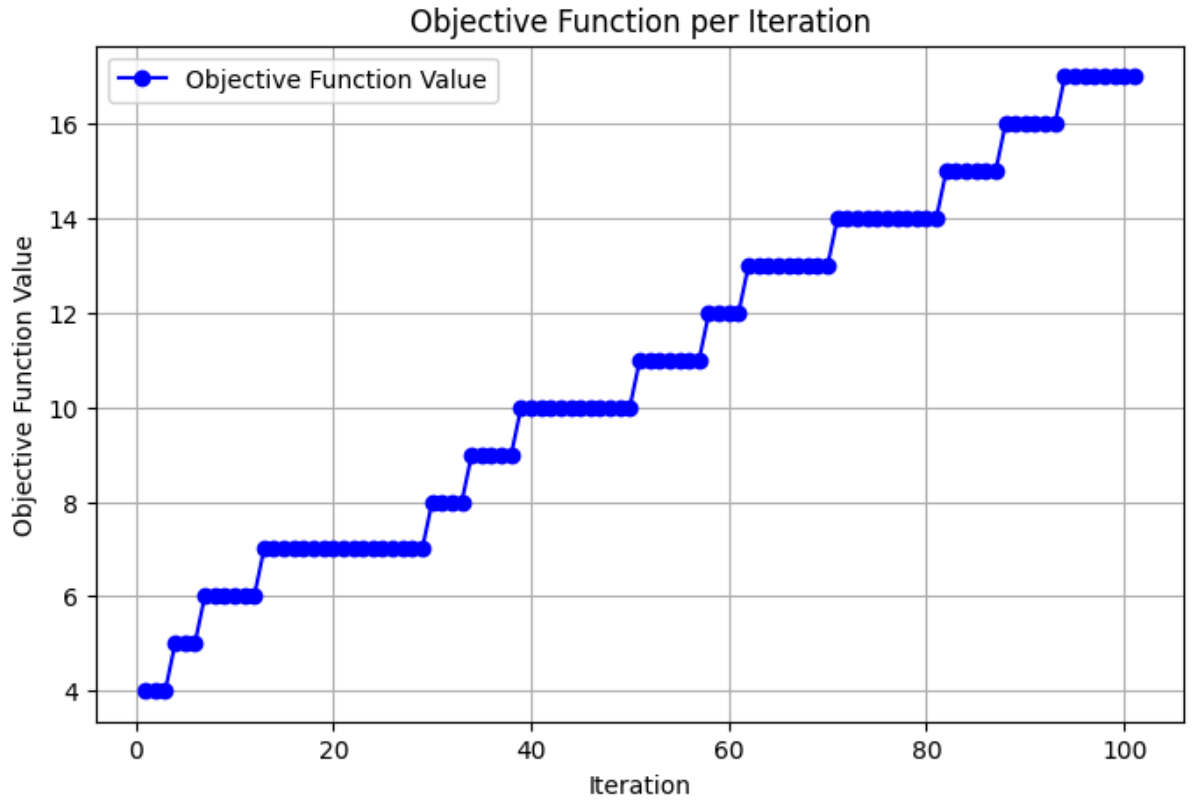


State Akhir

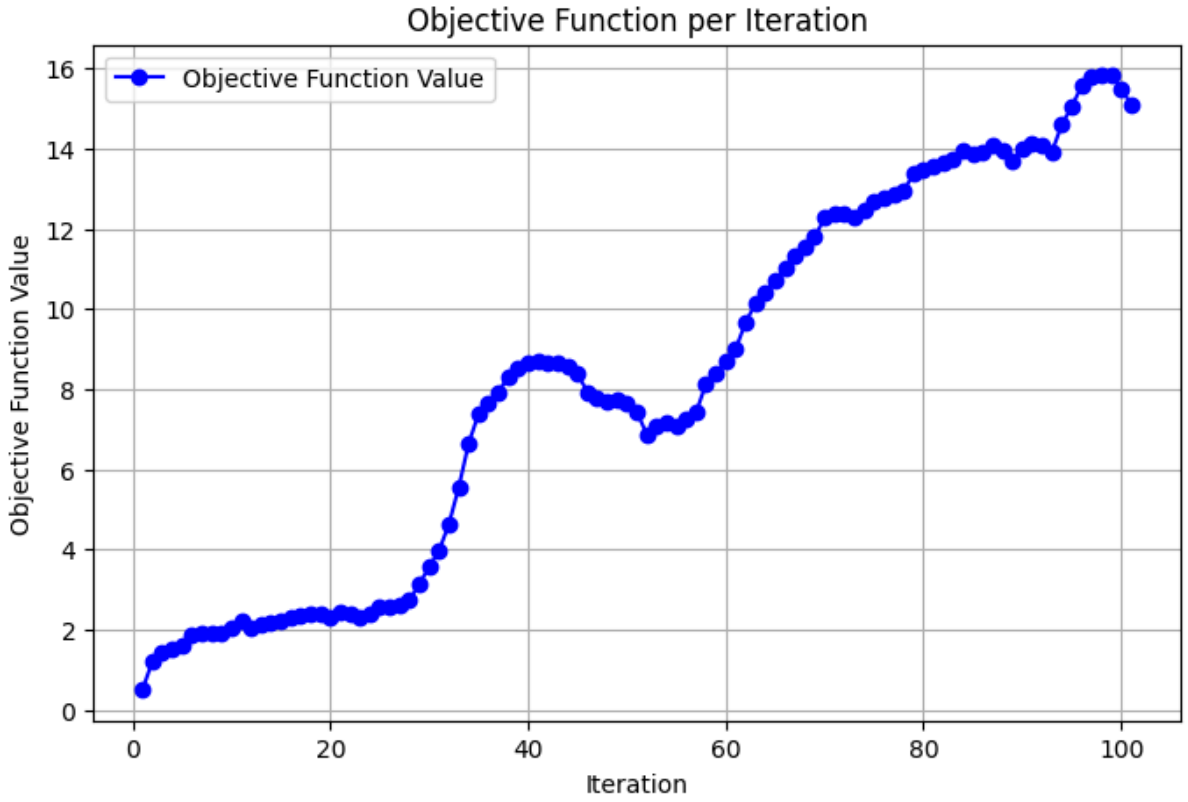


Nilai Objective Function: 17

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



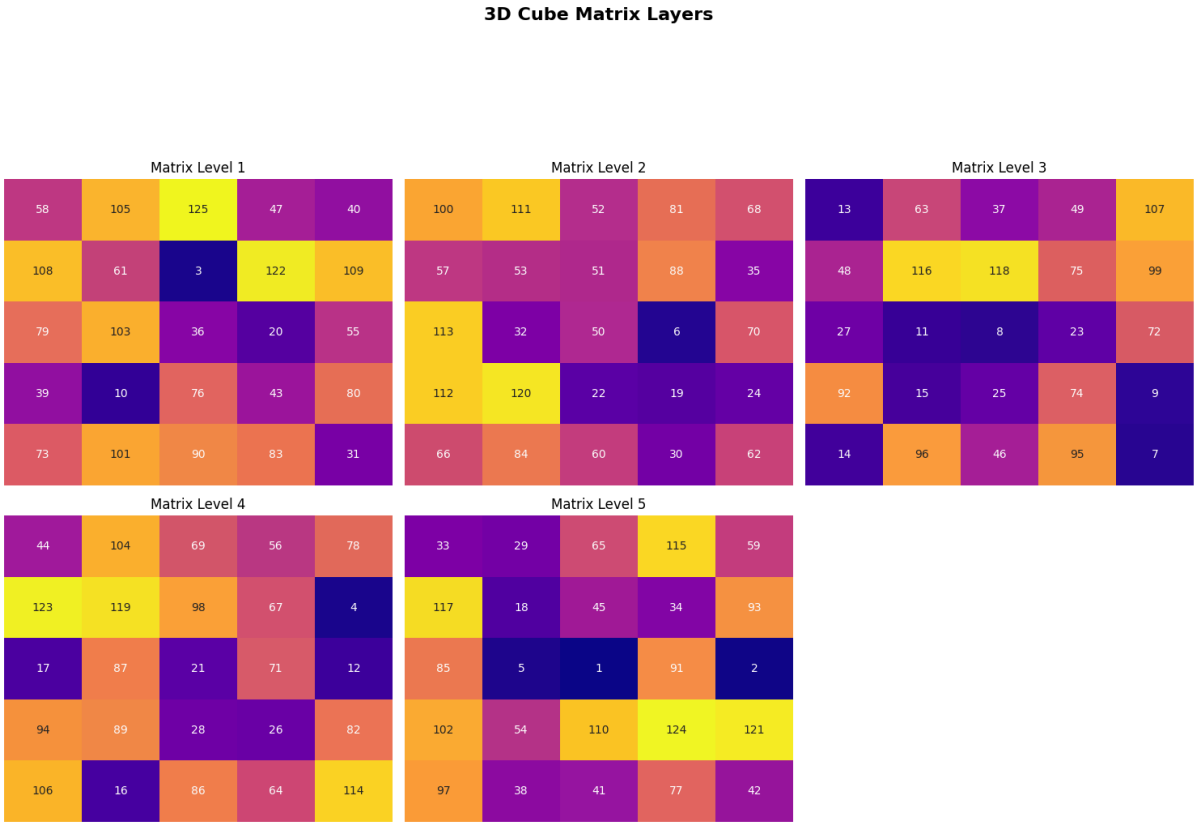
Jumlah Populasi: 300

Banyak Iterasi: 100

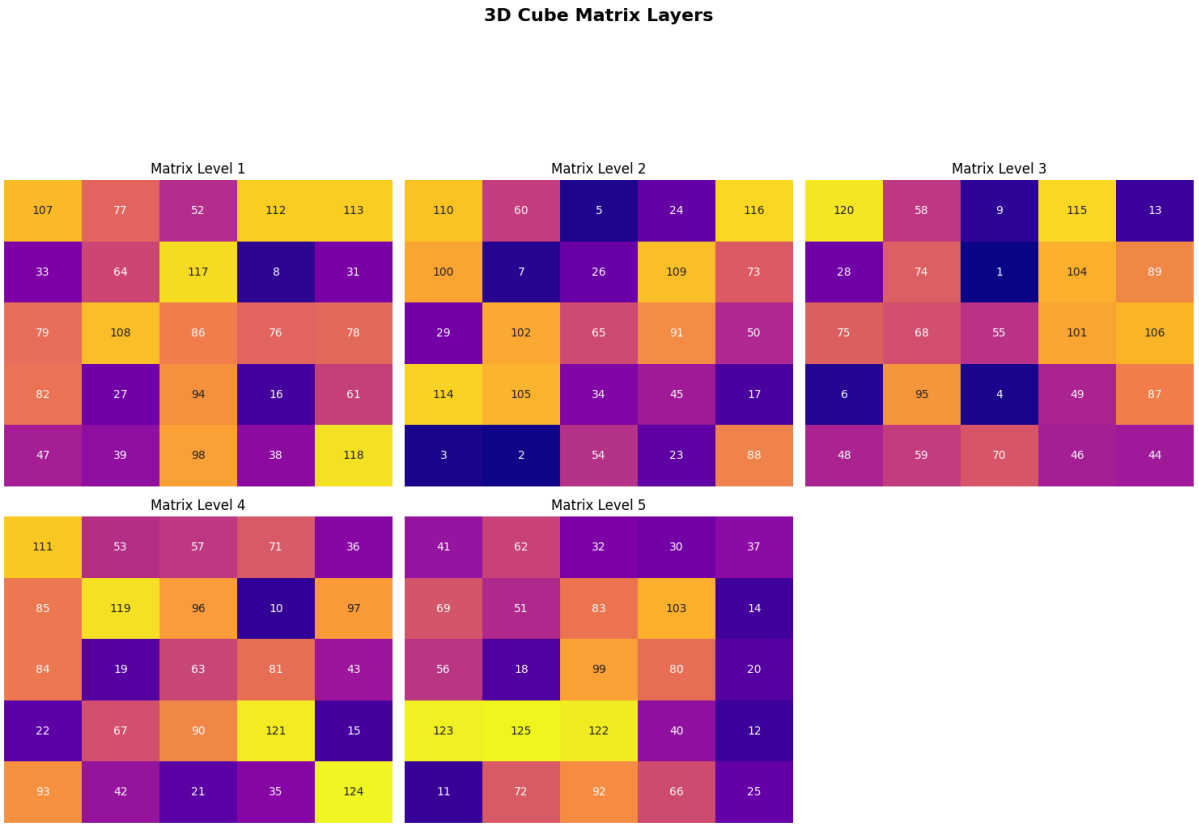
Durasi Proses Pencarian: 6 Second

2.2.6.12. Percobaan 12

State Awal

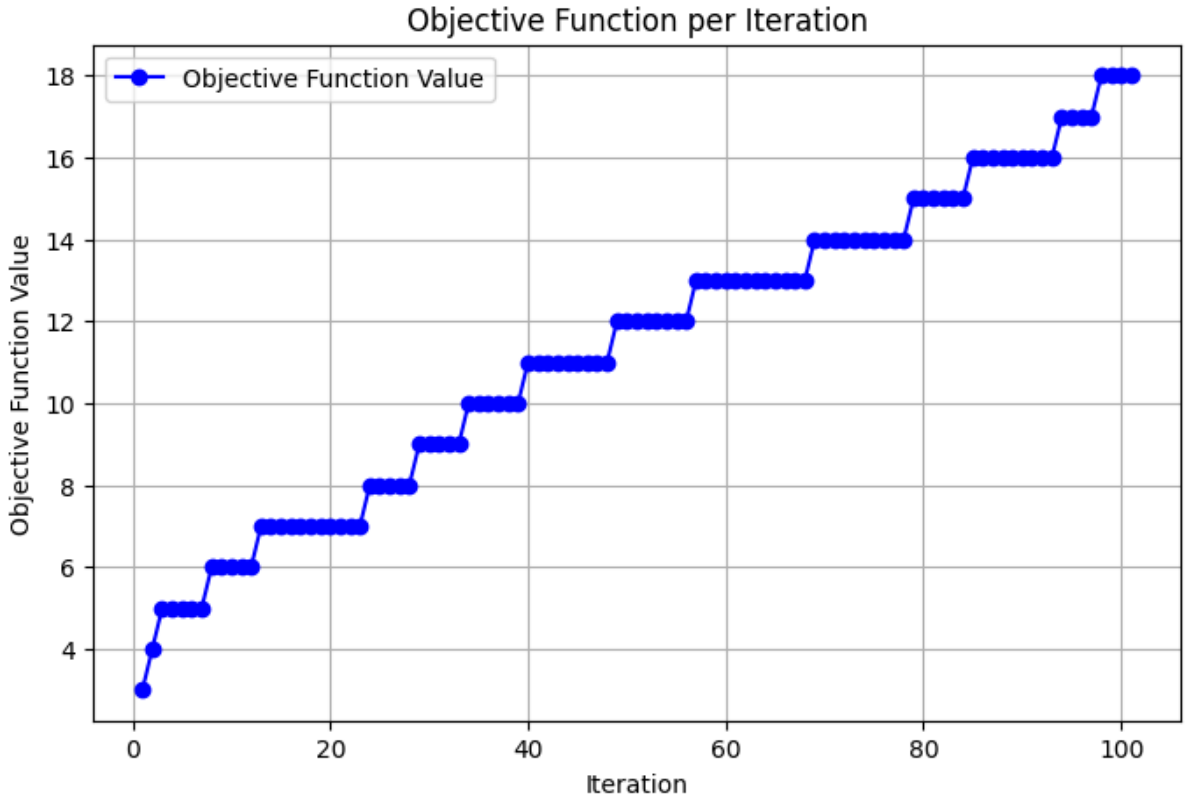


State Akhir

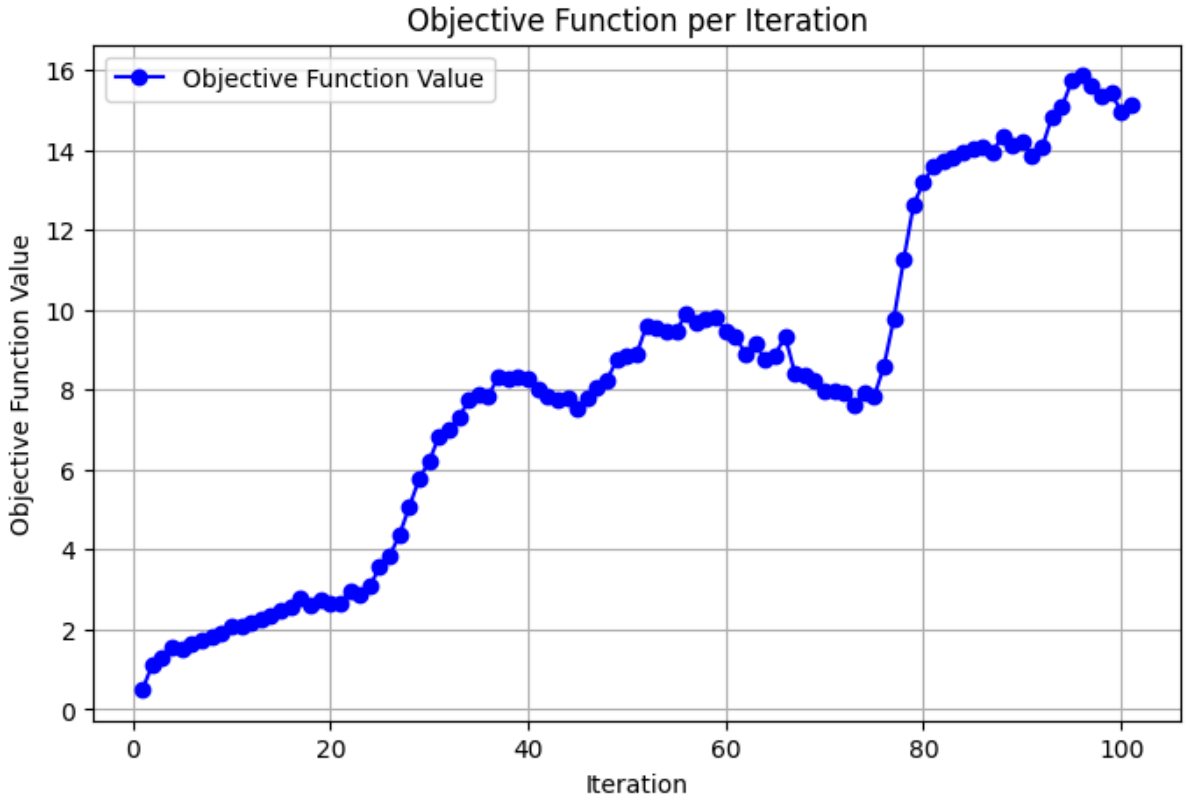


Nilai Objective Function: 18

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



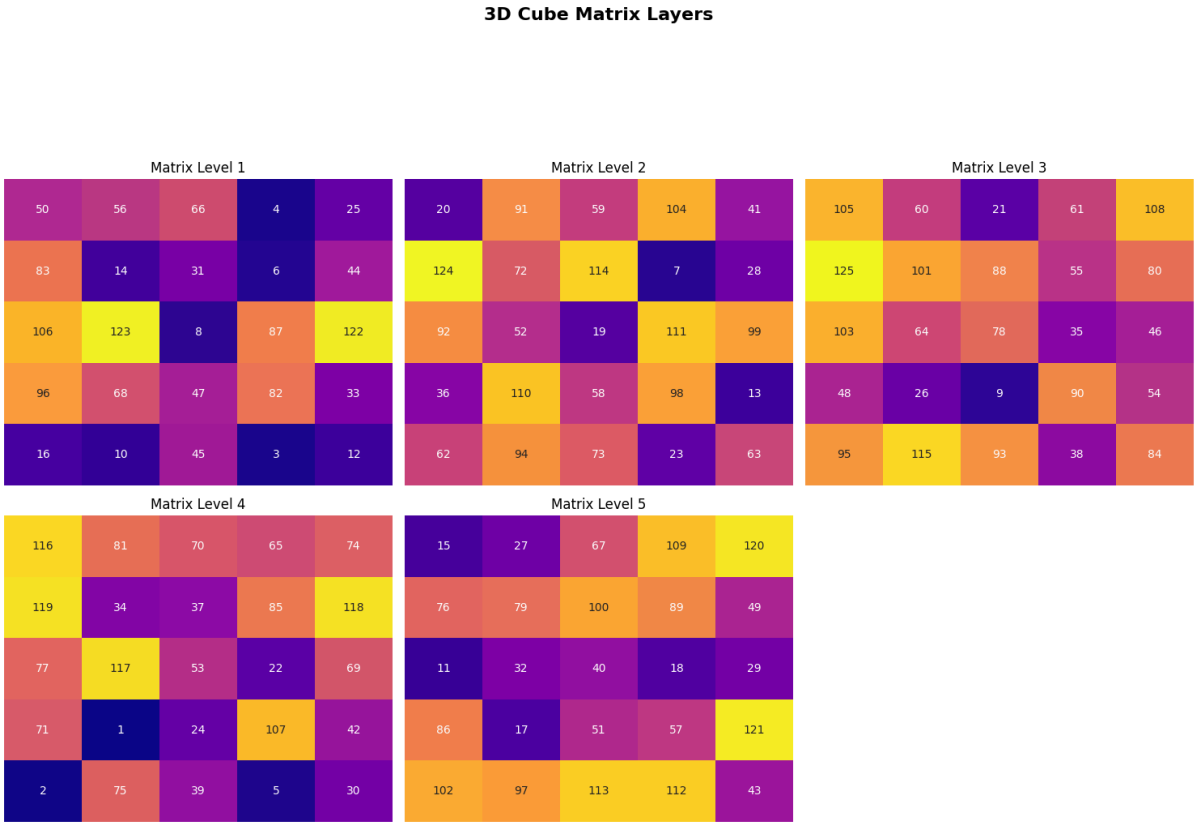
Jumlah Populasi: 300

Banyak Iterasi: 100

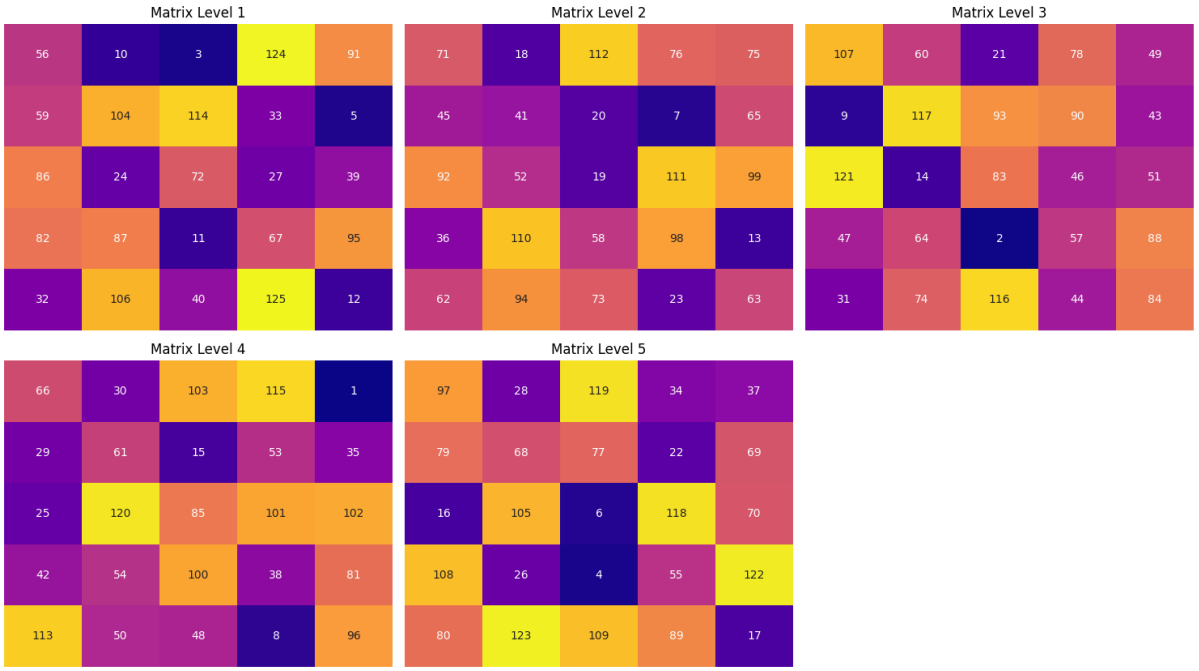
Durasi Proses Pencarian: 6 Second

2.2.6.13. Percobaan 13

State Awal

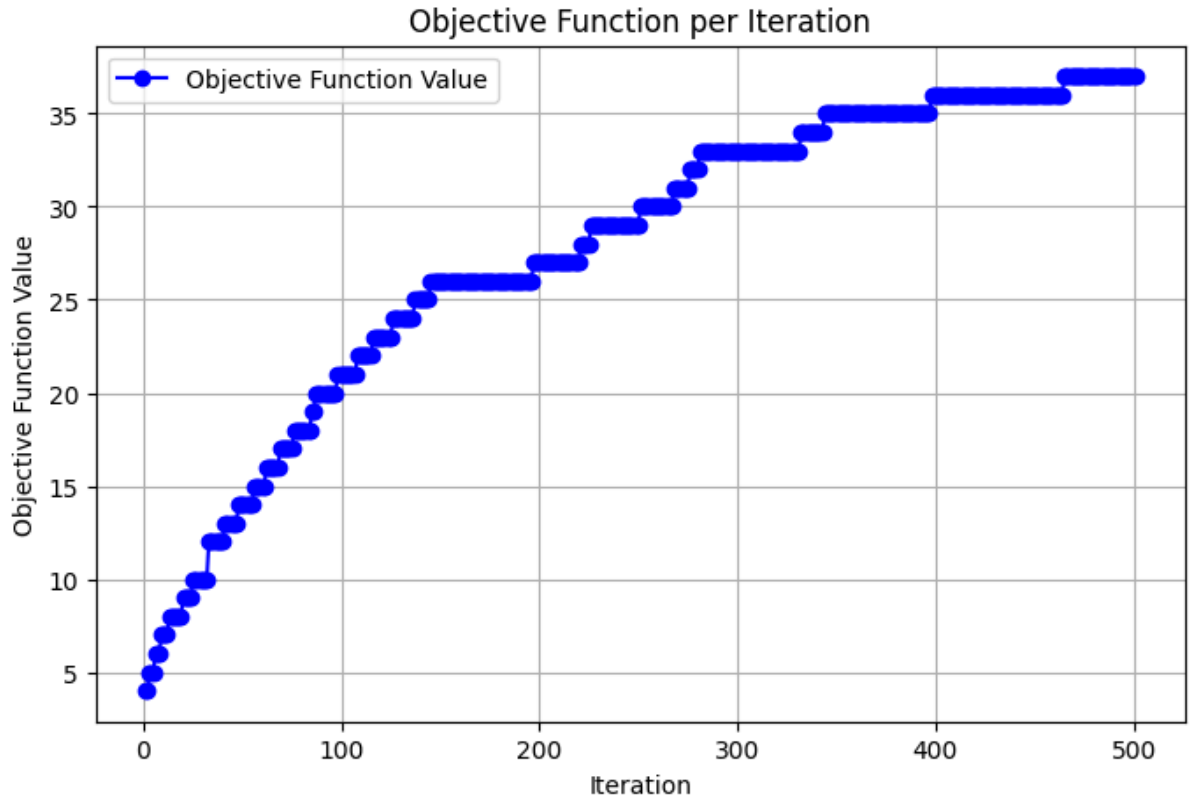


State Akhir

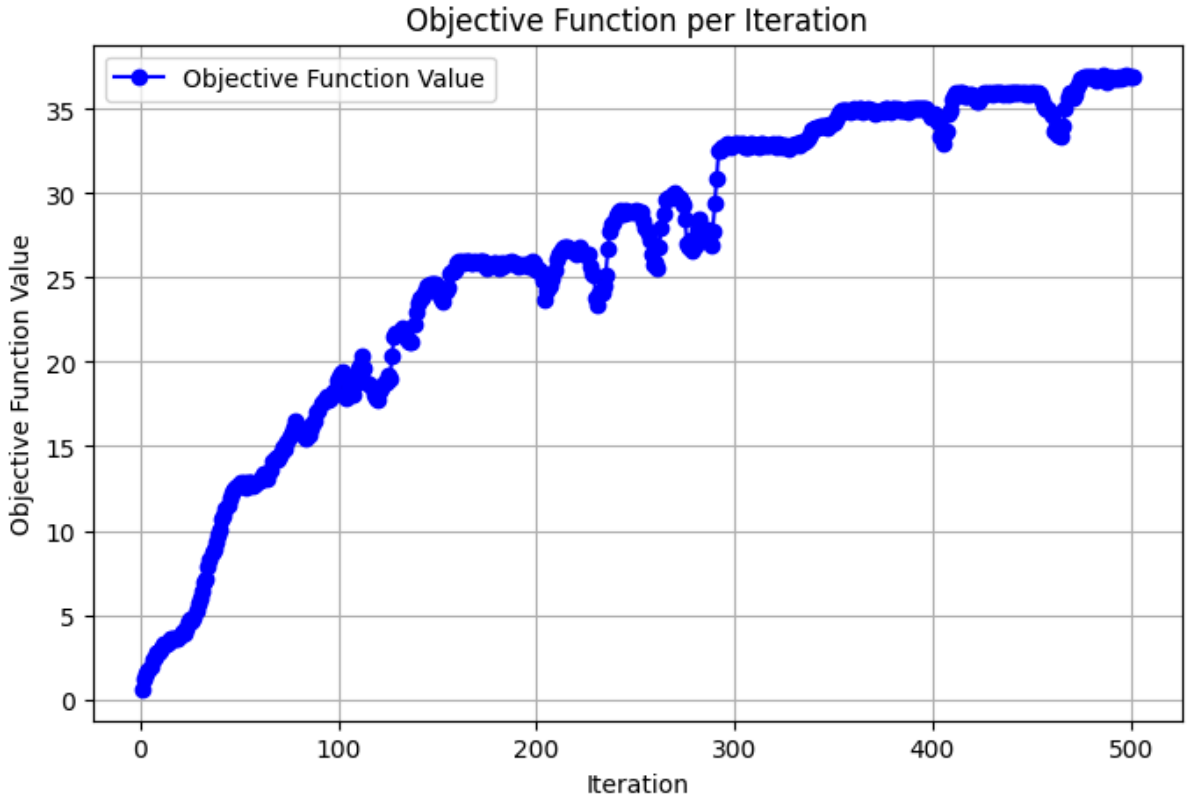


Nilai Objective Function: 37

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



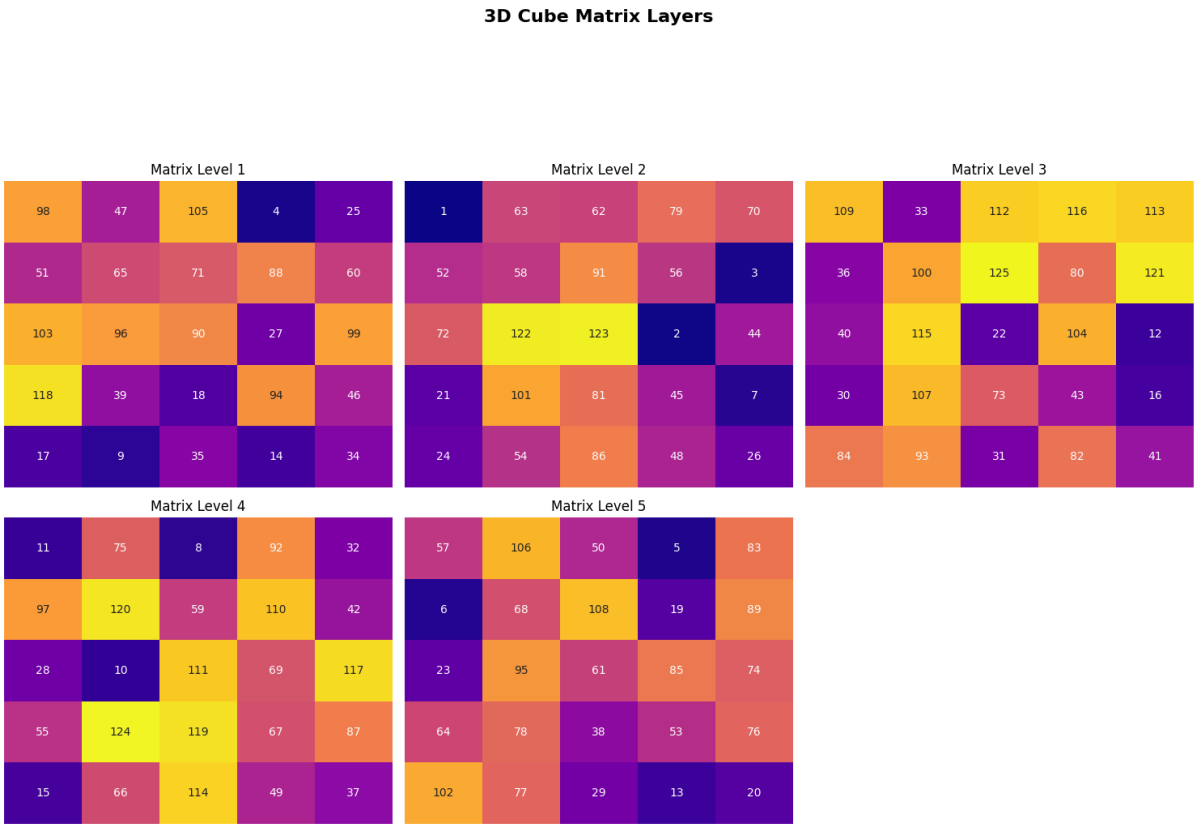
Jumlah Populasi: 300

Banyak Iterasi: 500

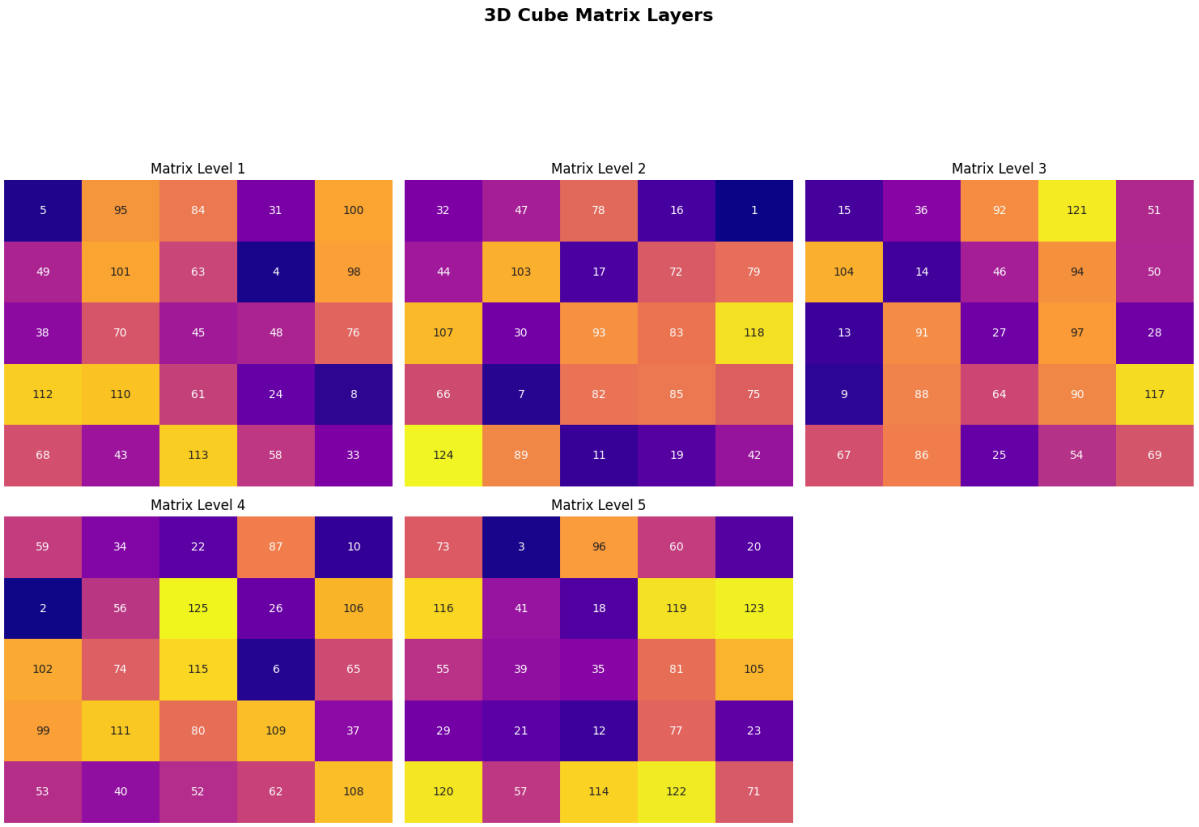
Durasi Proses Pencarian: 1.6 Second

2.2.6.14. Percobaan 14

State Awal

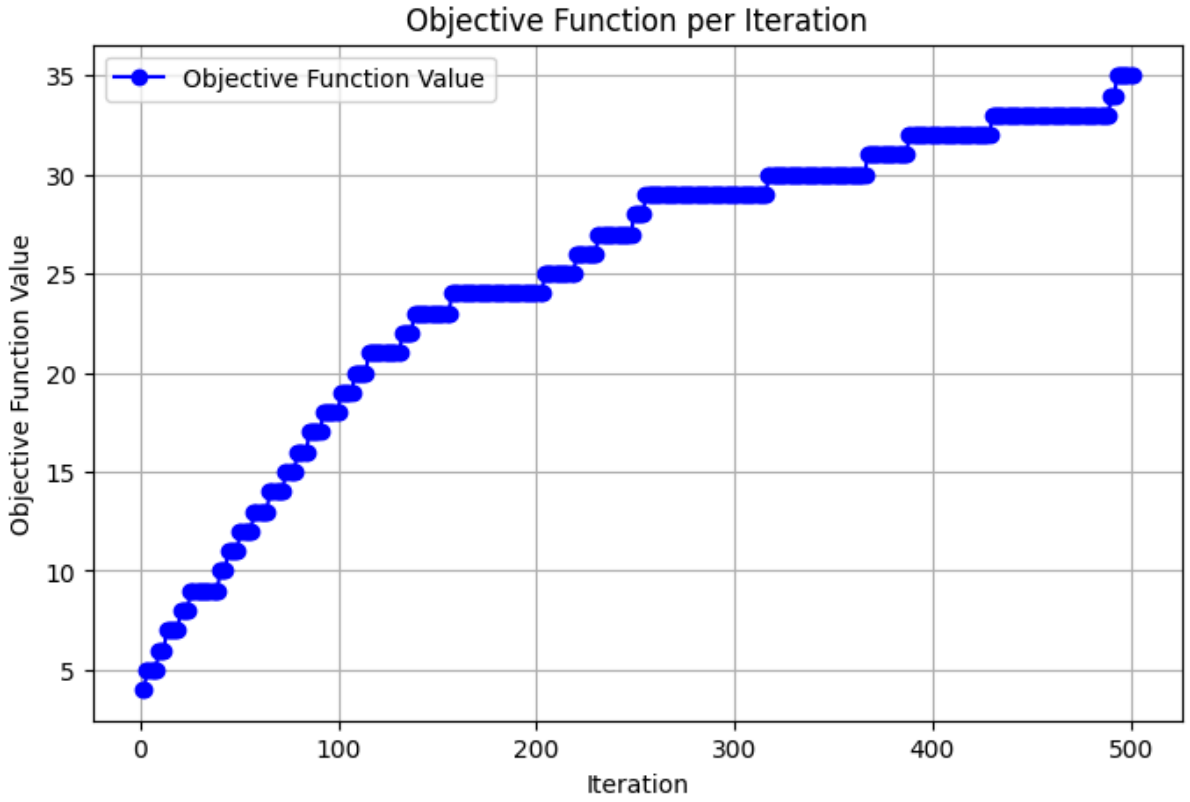


State Akhir

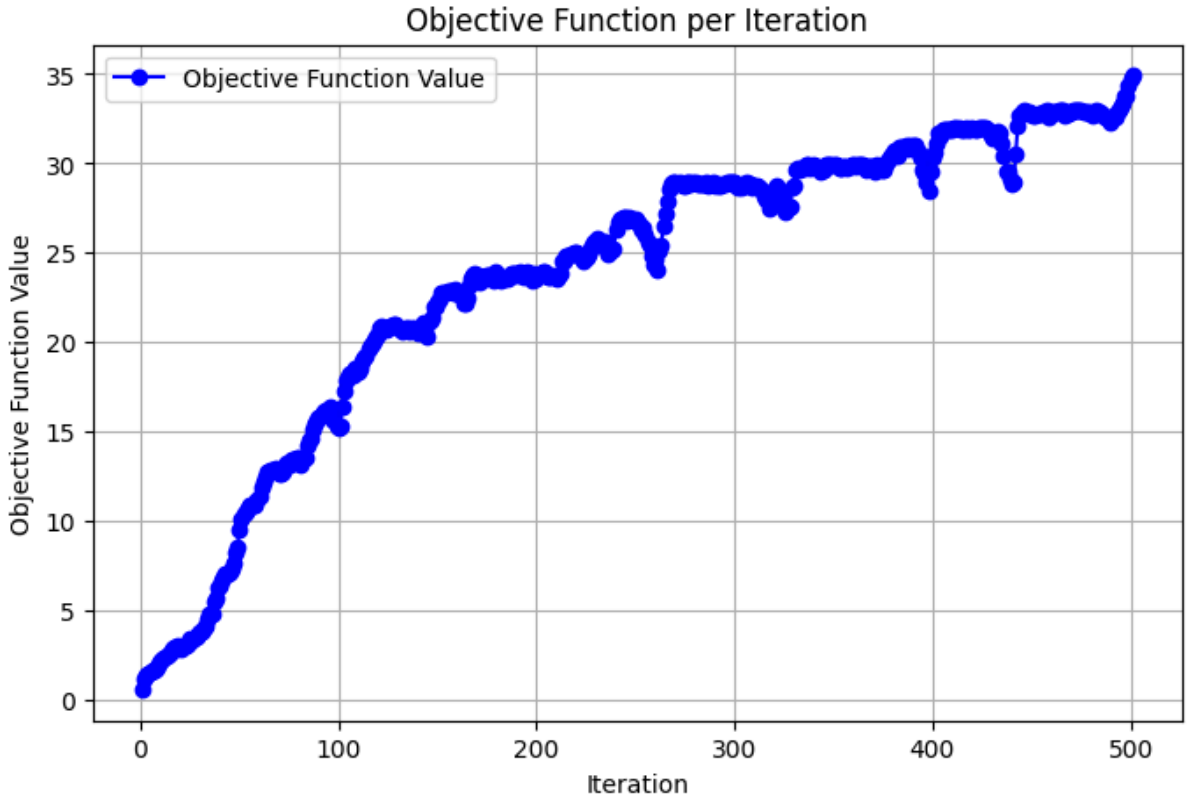


Nilai Objective Function: 35

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



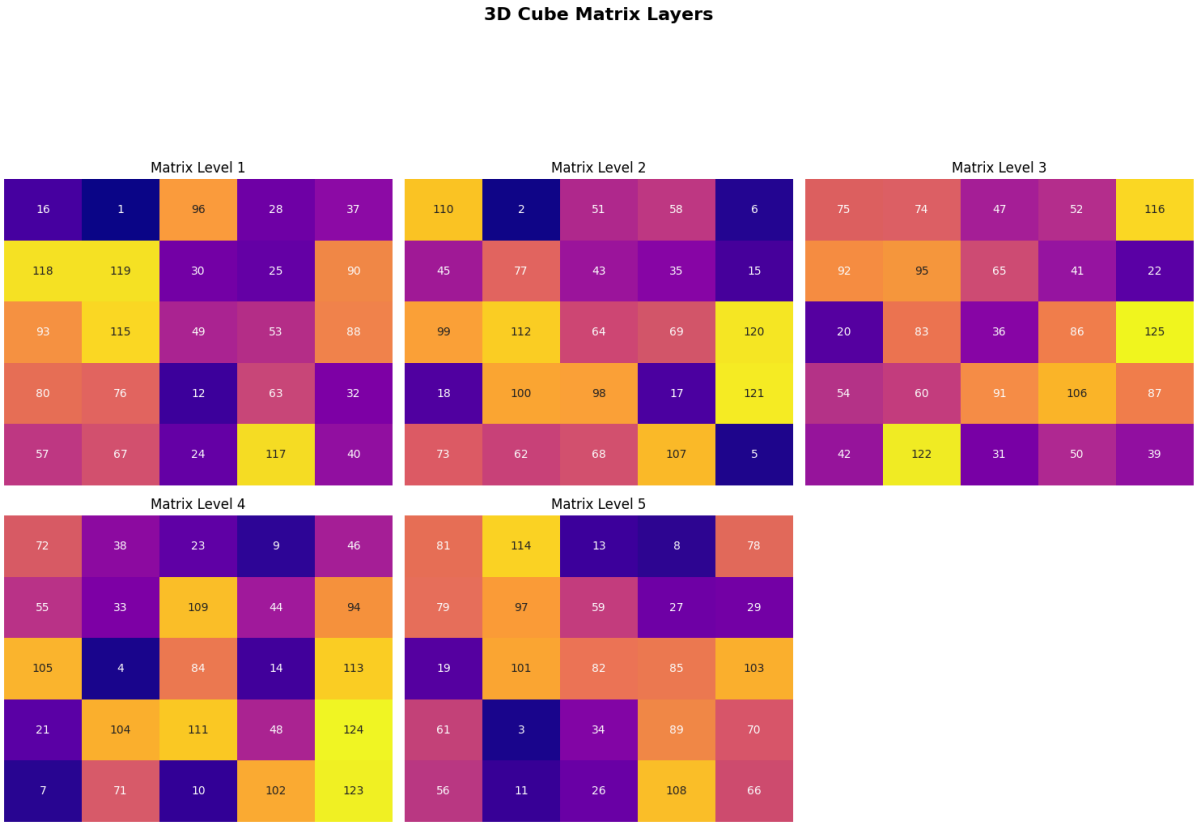
Jumlah Populasi: 300

Banyak Iterasi: 500

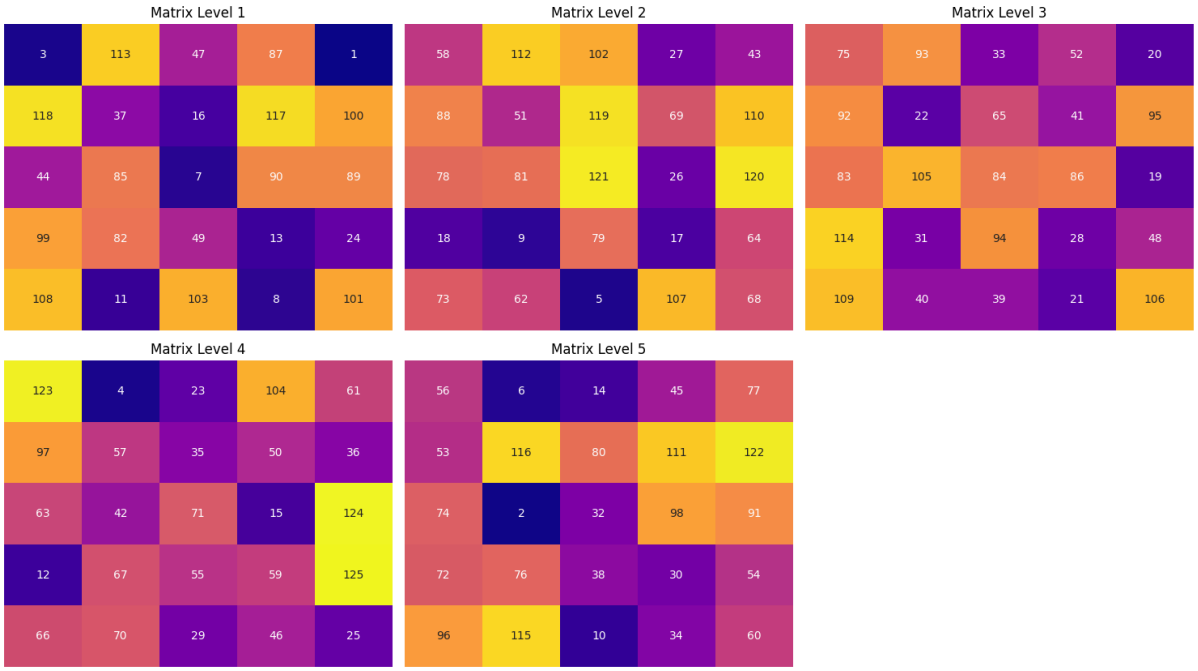
Durasi Proses Pencarian: 30 Second

2.2.6.15. Percobaan 15

State Awal

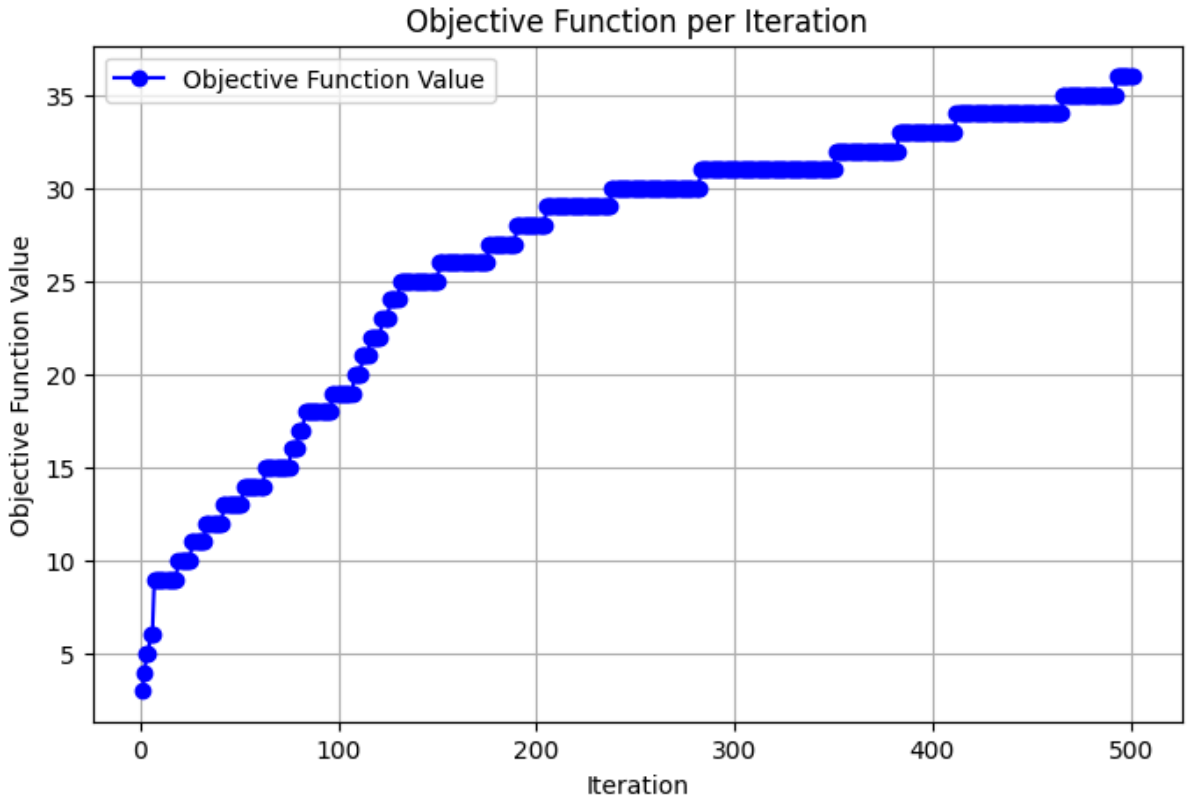


State Akhir

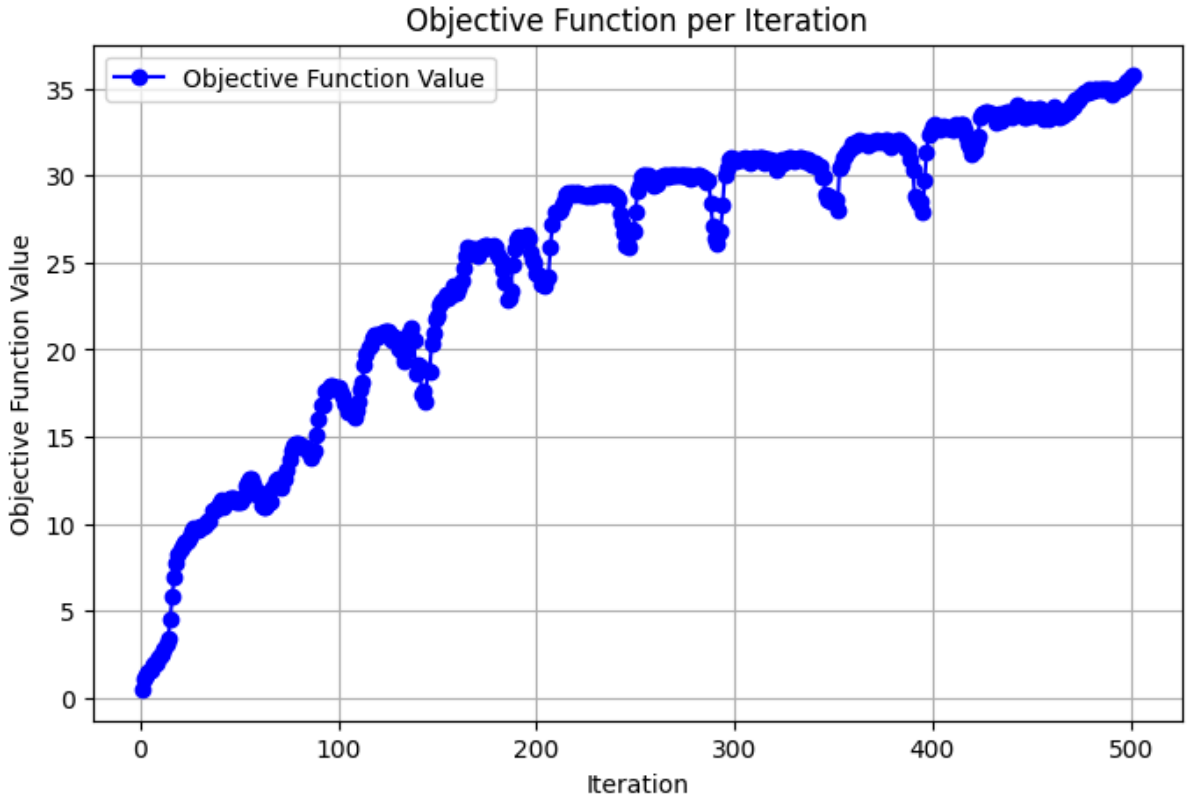


Nilai Objective Function: 36

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



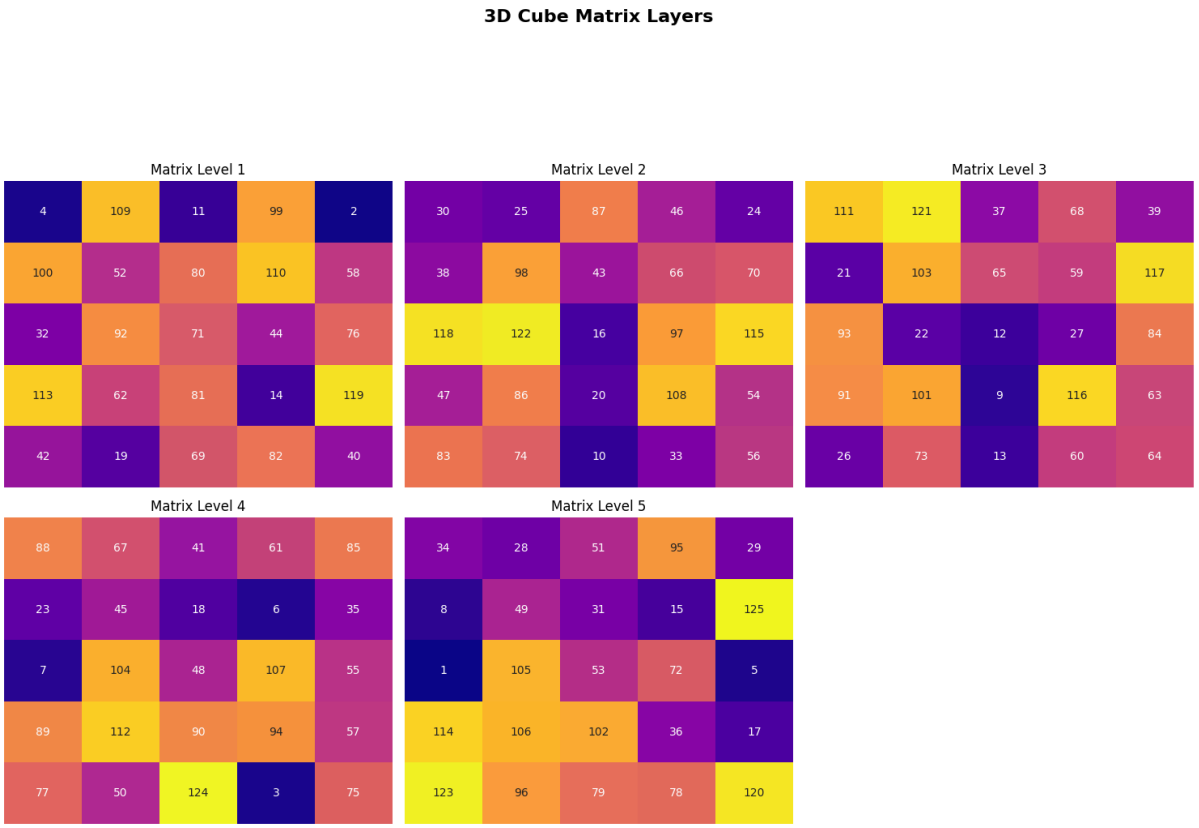
Jumlah Populasi: 300

Banyak Iterasi: 500

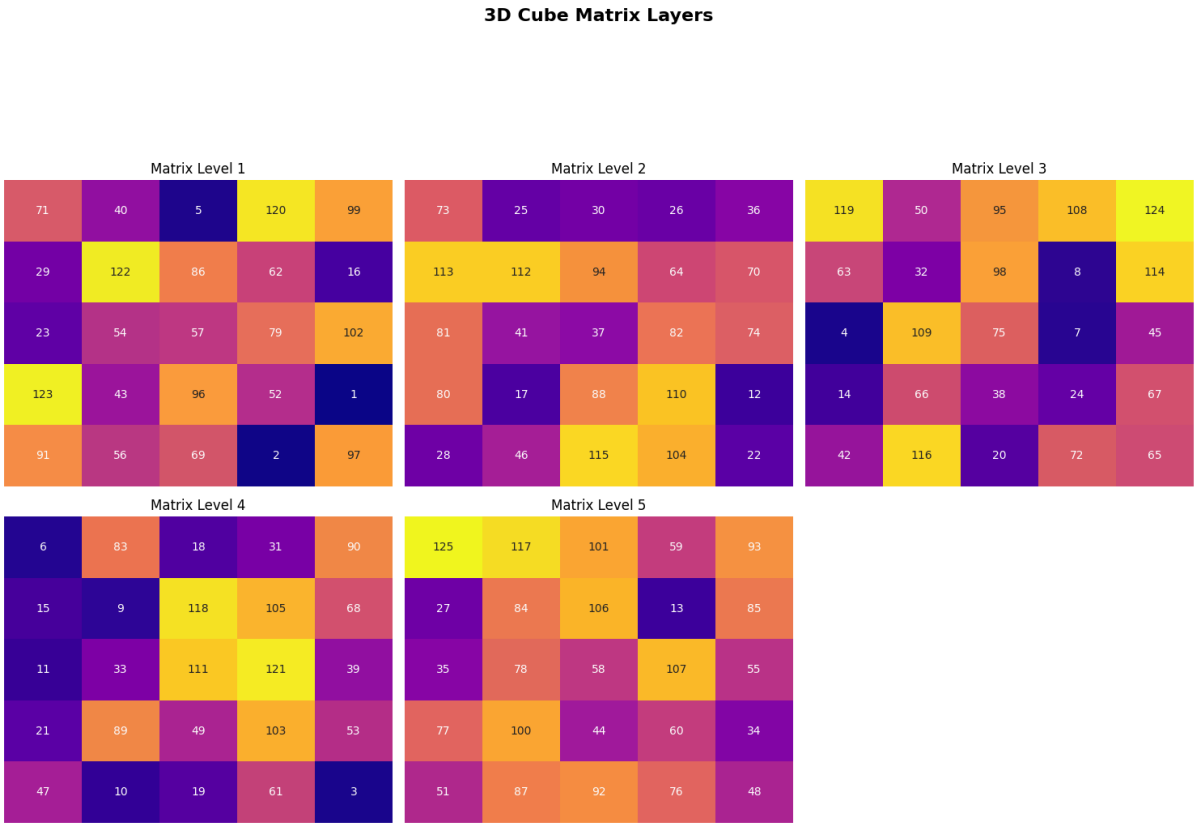
Durasi Proses Pencarian: 30 Second

2.2.6.16. Percobaan 16

State Awal

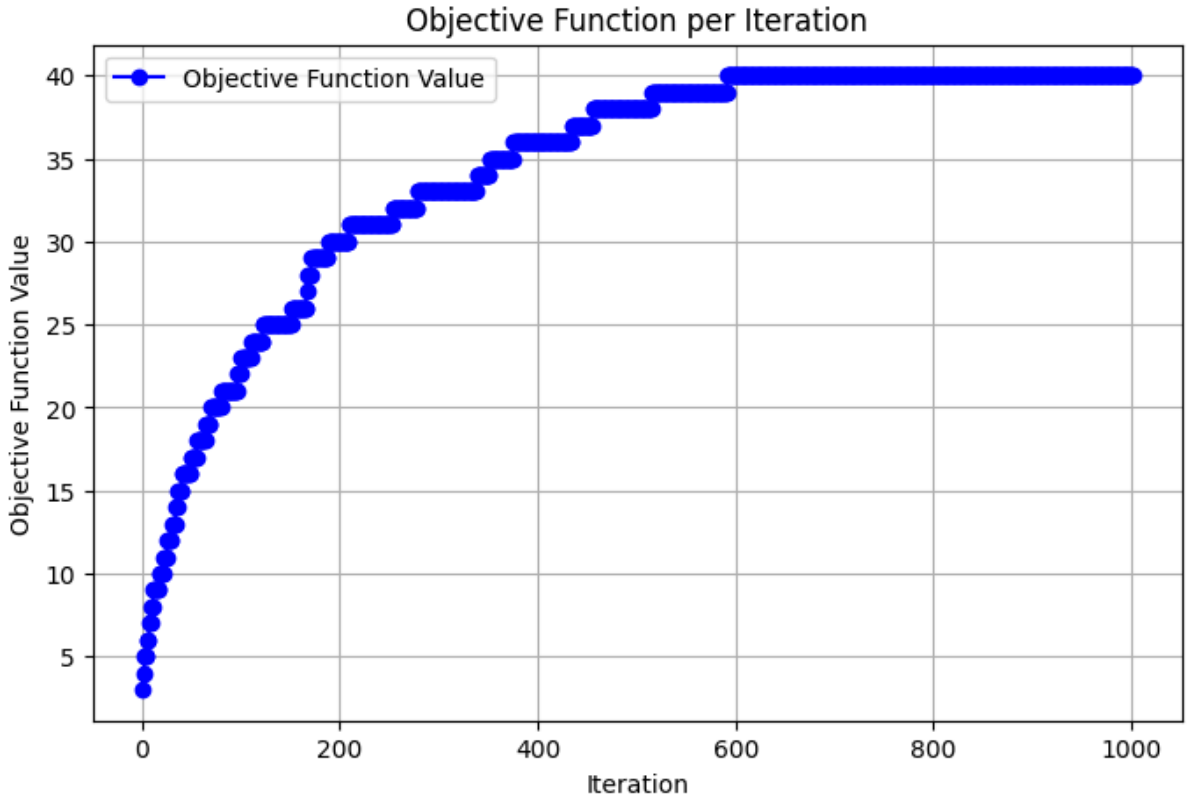


State Akhir

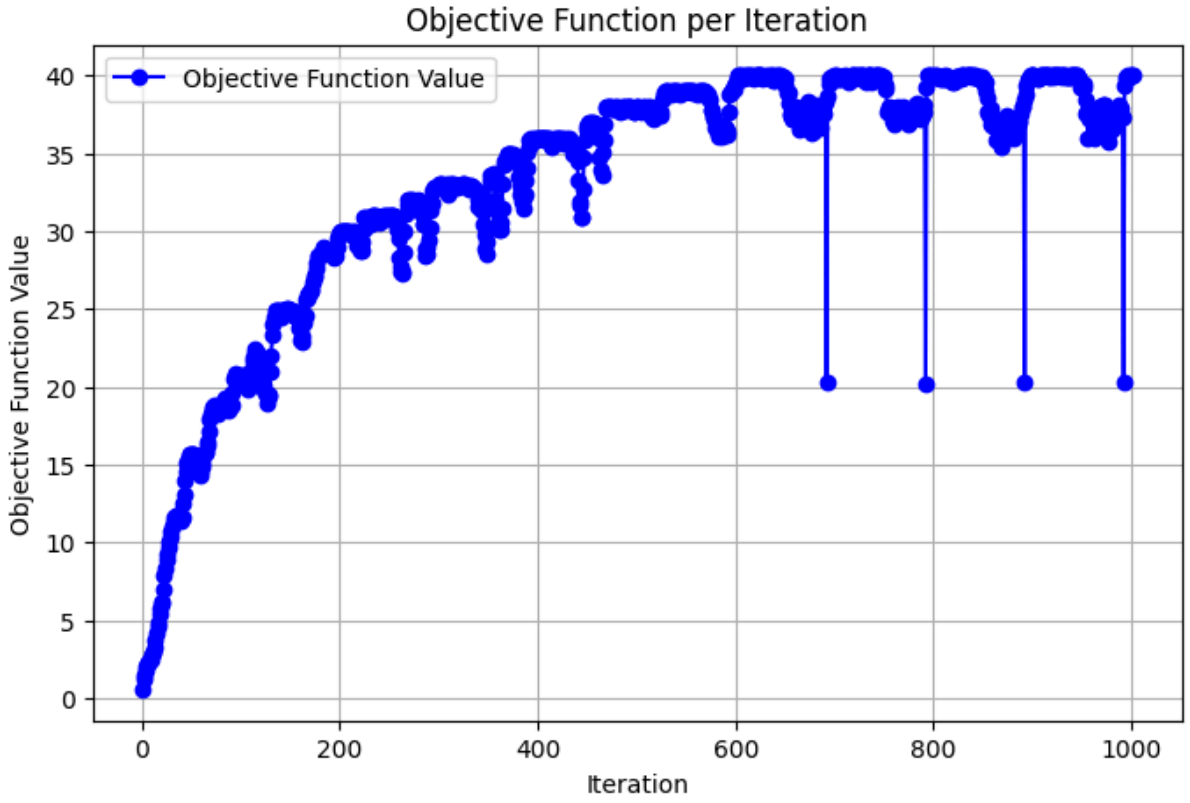


Nilai Objective Function: 40

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



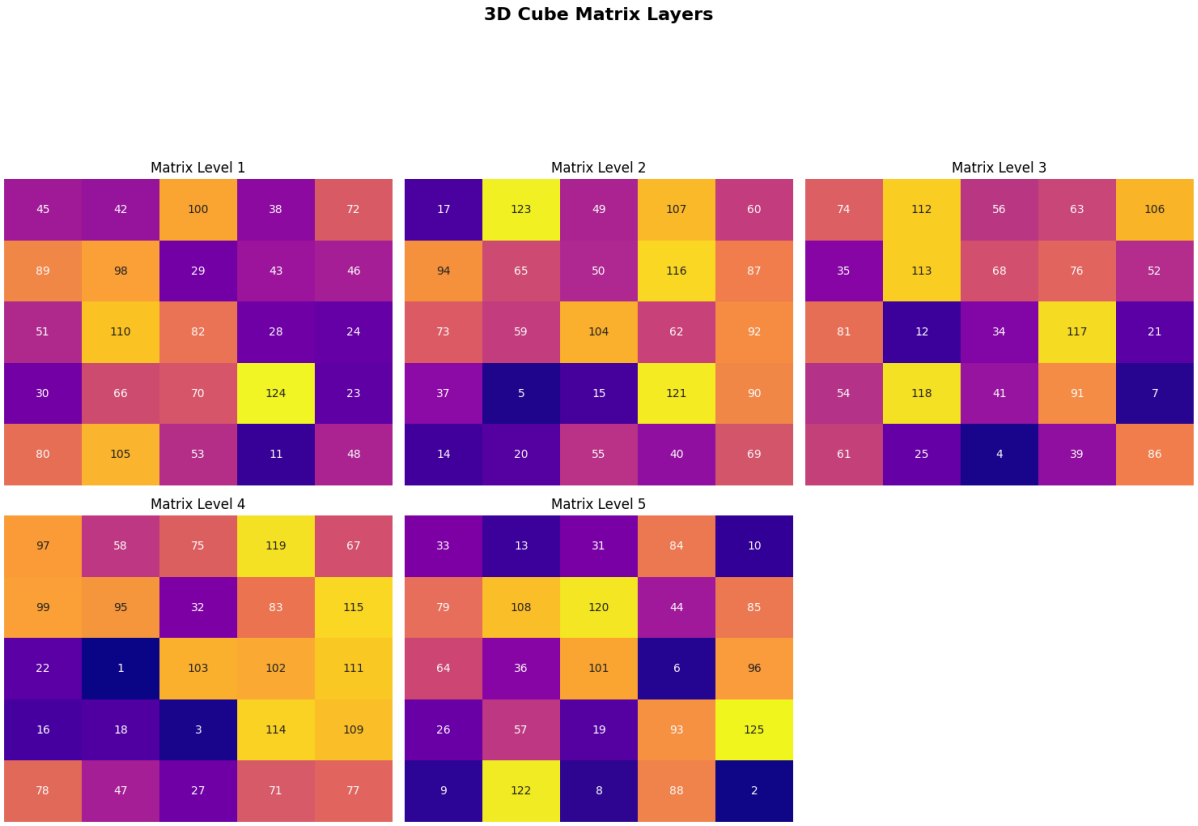
Jumlah Populasi: 300

Banyak Iterasi: 1000

Durasi Proses Pencarian: 63 Second

2.2.6.17. Percobaan 17

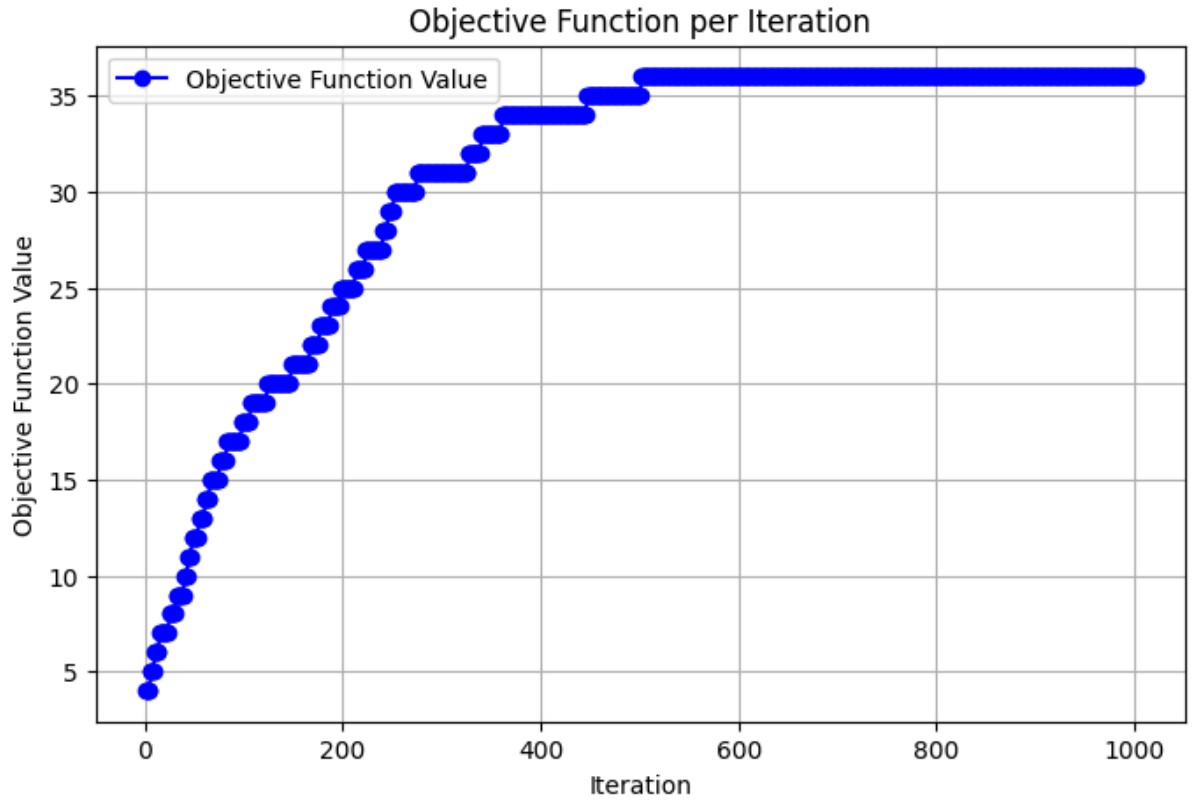
State Awal



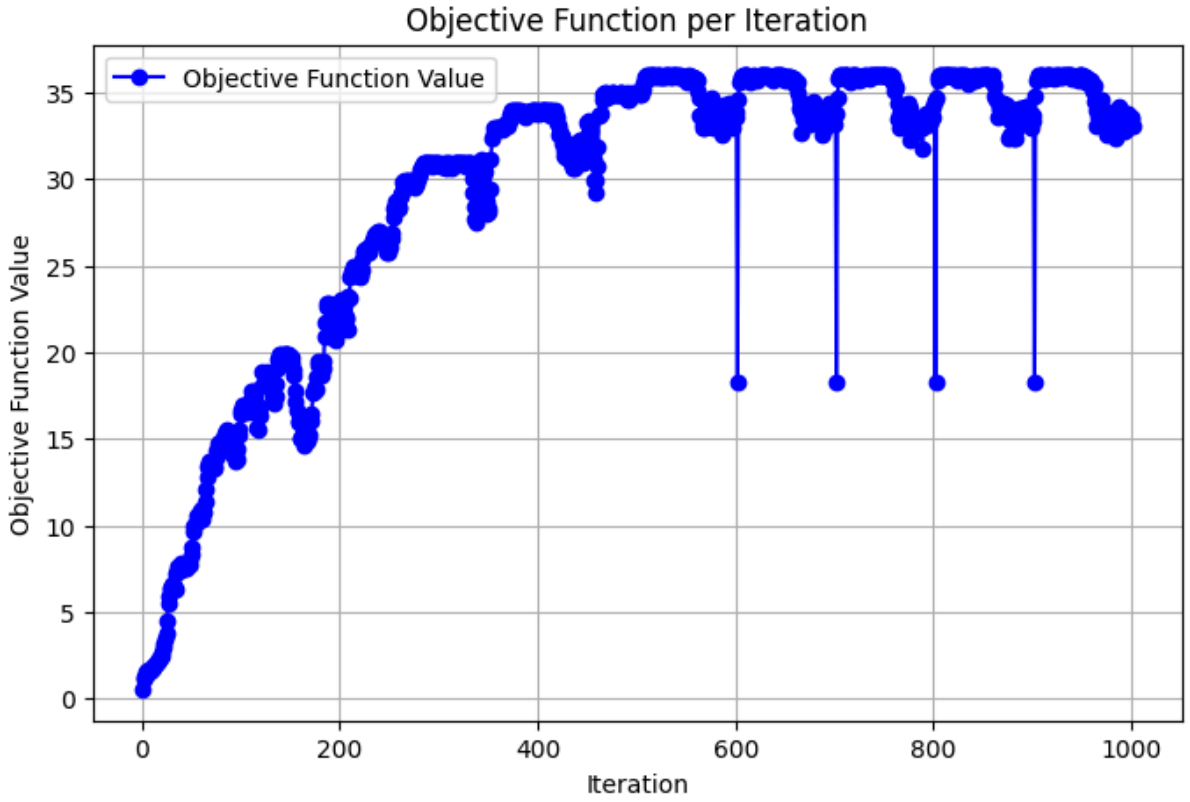
State Akhir

Nilai Objective Function: 36

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



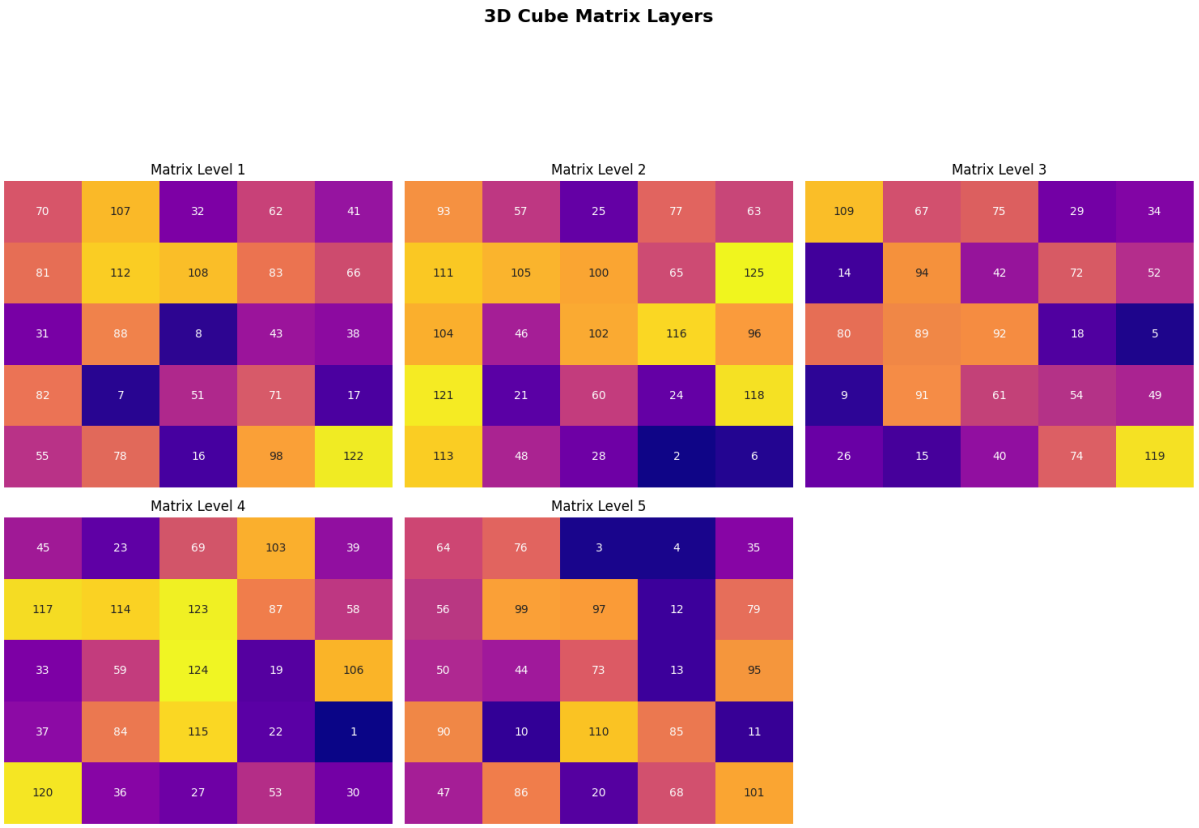
Jumlah Populasi: 300

Banyak Iterasi: 1000

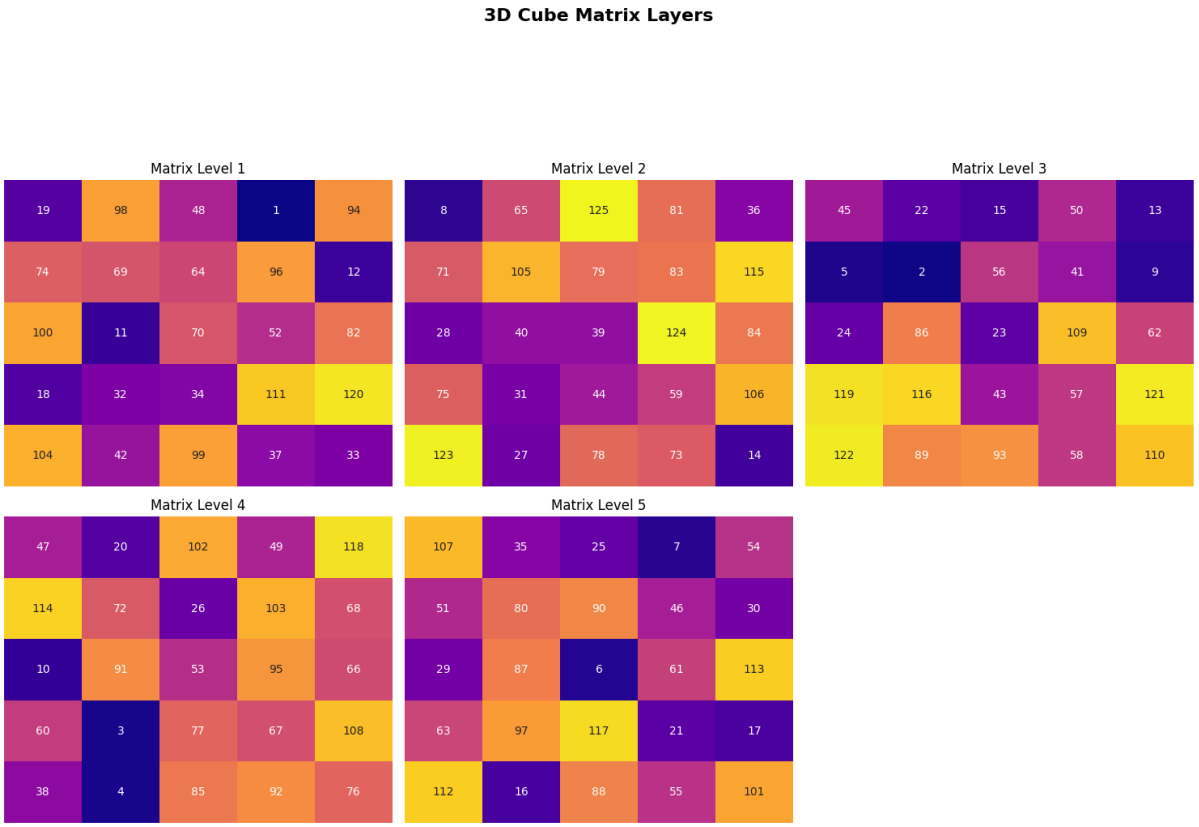
Durasi Proses Pencarian: 63 Second

2.2.6.18. Percobaan 18

State Awal

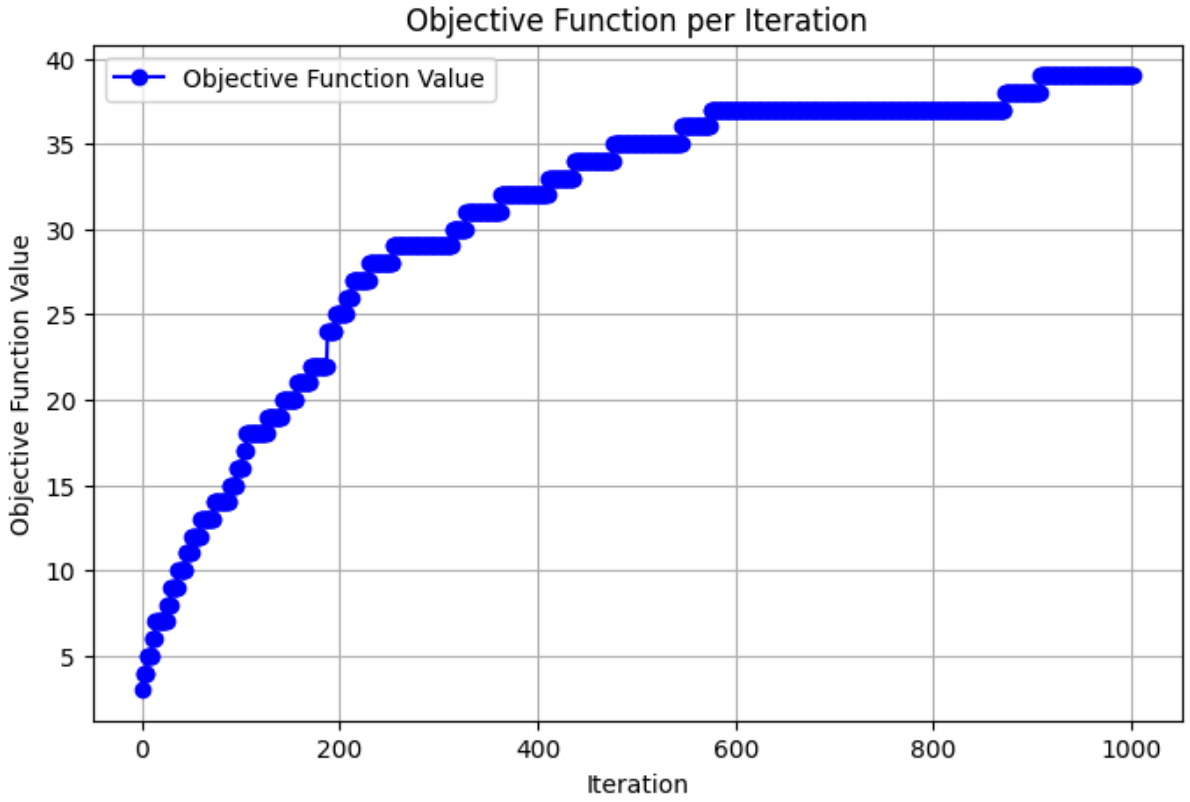


State Akhir

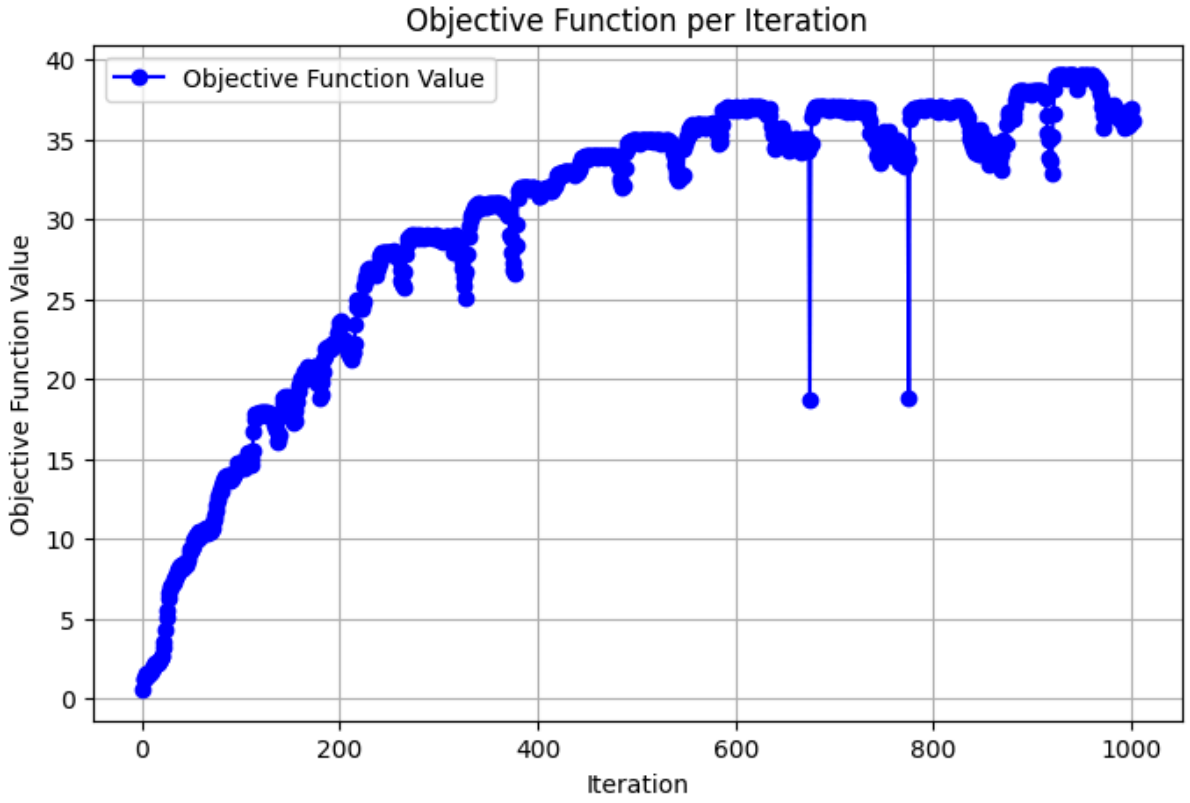


Nilai Objective Function: 39

Plot Nilai Objective Function Max:



Plot Nilai Objective Function Avg:



Jumlah Populasi: 300

Banyak Iterasi: 1000

Durasi Proses Pencarian: 62 Second

BAB 3

Analisis

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

Masing - masing algoritma memiliki kemampuan mendekati global optima yang berbeda - beda. Perlu diketahui bahwa global optima memiliki nilai *objective function*/*fitness value* yakni 109. Berikut merupakan ringkasan hasil yang didapatkan :

1. *Genetic Algorithm* : berdasarkan hasil analisis penulis, algoritma ini cenderung memiliki nilai *fitness value* yang kecil dibandingkan dengan algoritma lainnya, tetapi algoritma ini mampu menghindari solusi lokal dengan baik dengan menggunakan metode *crossover* dan *mutation*. Alasan algoritma ini memiliki *fitness value* yang sedikit jauh dari global optima adalah maksimum iterasinya. Algoritma ini memiliki parameter maksimum iterasi yang menyebabkan tidak dicapainya global optima walaupun mampu menghindari solusi lokal. Alasan lain yang menghambat algoritma ini adalah kebutuhan komputasionalnya yang berat dikarenakan perlunya membuat populasi baru untuk setiap iterasi.
2. *Simulated Annealing* : Dari antara algoritma lain yang penulis analisis, algoritma ini merupakan algoritma yang paling mungkin mendekati global optima. Algoritma *simulated annealing* memiliki kemampuan untuk menghindari solusi lokal karena algoritma ini memperbolehkan pemilihan *state* yang lebih buruk berdasarkan *temperature* dan *schedule*. Algoritma ini juga tidak berat secara komputasional sehingga dapat mencari solusi dengan cepat dan cukup dekat dengan global optima.
3. *Steepest Ascent Hill-climbing* : Berdasarkan analisis penulis, algoritma ini menghasilkan solusi yang cukup dekat dengan global optima dengan *fitness value* yang besar, hanya saja algoritma ini cenderung terjebak dalam solusi lokal dibandingkan dengan algoritma lainnya.
4. *Hill-climbing with sideways move* : Berdasarkan analisis penulis, algoritma ini menghasilkan solusi yang cukup dekat dengan global optima dibandingkan dengan *Steepest Ascent Hill-climbing* karena algoritma ini menghindari solusi lokal dengan memperbolehkan pemilihan suksesor dengan nilai yang sama dengan *current cube*. Hal ini memberikan algoritma sedikit lebih fleksibilitas dalam memilih suksesor dibandingkan dengan *Steepest Ascent Hill-climbing.*

5. *Random Restart Hill-climbing* : Berdasarkan analisis penulis, algoritma ini menghasilkan solusi yang cukup dekat dengan global optima dengan *fitness value* yang cukup besar serta algoritma ini dapat menghindari solusi lokal dengan cara mengulangi algoritma *hill climbing* dengan *initial state* yang berbeda, hanya saja algoritma ini tidak sebaik *Simulated Annealing* dalam proses menghindari solusi lokal tersebut.
6. *Stochastic Hill-climbing* : Berdasarkan analisis penulis, algoritma ini menghasilkan solusi yang cukup **jauh** dengan global optima dengan *fitness value* yang cukup rendah. Hal ini disebabkan cara algoritma memilih *cube* berikutnya sebagai suksesor yakni secara *random*.

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

Jika kita bandingkan algoritma pencarian local search yang kita pakai, dapat disimpulkan beberapa hal

1. Steepest Ascent Hill-climbing : Menghasilkan solusi dengan *fitness value* yang relatif lebih besar dibandingkan *Stochastic Hill-climbing* dan *genetic algorithm*.
2. Sideways Move Hill Climbing : Menghasilkan solusi dengan *fitness value* yang relatif lebih besar dibandingkan *Stochastic Hill-climbing*, *genetic algorithm, Steepest Ascent Hill-climbing,* dan *Random Restart Hill-climbing.*
3. Random Restart Hill Climbing : Menghasilkan solusi dengan *fitness value* yang relatif lebih besar dibandingkan *Stochastic Hill-climbing*, *genetic algorithm,* dan *Steepest Ascent Hill-climbing.*
4. Stochastic Hill Climbing : Menghasilkan solusi dengan nilai *fitness value* yang paling rendah dibandingkan dengan algoritma yang lain.
5. Simulated Annealing : Menghasilkan solusi dengan nilai *fitness value* yang paling besar dibandingkan dengan algoritma yang lain.
6. Genetic Algorithm : Menghasilkan solusi dengan *fitness value* yang relatif lebih besar dibandingkan *Stochastic Hill-climbing*.

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

1. Steepest Ascent Hill Climbing : Algoritma ini memiliki durasi yang lumayan cepat dibandingkan yang lain yaitu dari hasil rata-rata memiliki durasi 30 detik
2. Sideways Move Hill Climbing : Algoritma ini memiliki durasi yang lumayan cepat dibandingkan yang lain yaitu dari hasil rata-rata memiliki durasi 30 detik
3. Random Restart Hill Climbing :Algoritma ini memiliki durasi yang lumayan lama dibandingkan yang lain yaitu dari hasil rata-rata memiliki durasi 110 detik.

4. Stochastic Hill Climbing : Algoritma ini memiliki durasi paling cepat dengan hasil dari eksperimen mendapatkan durasi rata-rata 0.3 detik.
5. Simulated Annealing : Algoritma ini memiliki durasi yang lumayan cepat dibandingkan yang lain yaitu dari hasil rata-rata memiliki durasi 16 detik.
6. Genetic Algorithm : Algoritma ini memiliki durasi yang cukup bervariatif tergantung parameter yang dimasukkan, mulai dari 1 detik hingga 60 detik.

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

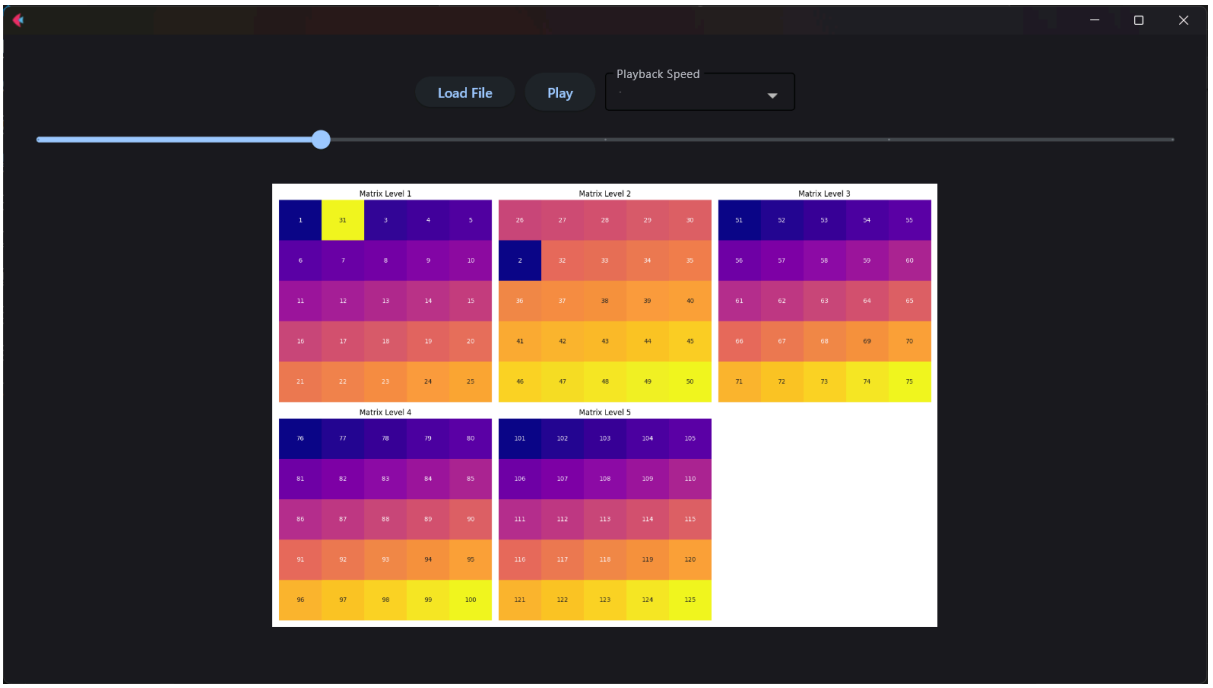
1. Steepest Ascent Hill Climbing : Berdasarkan eksperimen algoritma ini menghasilkan hasil yang cukup konsisten dengan state akhir yang memiliki value 40-44.
2. Sideways Move Hill Climbing : Berdasarkan eksperimen algoritma ini menghasilkan hasil yang cukup konsisten dengan state akhir yang memiliki value 41-45.
3. Random Restart Hill Climbing : Berdasarkan eksperimen algoritma ini menghasilkan hasil yang sedikit tidak konsisten dengan state akhir yang memiliki value 35-44.
4. Stochastic Hill Climbing : Berdasarkan eksperimen algoritma ini menghasilkan hasil yang tidak konsisten dengan state akhir yang memiliki value 8-19.
5. Simulated Annealing : Berdasarkan eksperimen algoritma ini menghasilkan hasil yang cukup konsisten dengan state akhir yang memiliki value 50-53.
6. Genetic Algorithm : Berdasarkan eksperimen algoritma ini menghasilkan hasil yang cukup konsisten dengan state akhir yang memiliki value tergantung parameter namun dari hasil eksperimen cukup konsisten dalam parameter yang sama.

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

Berdasarkan hasil percobaan, didapatkan bahwa semakin banyak jumlah iterasi maka nilai *objective function* yang dihasilkan juga semakin besar. Sedangkan untuk jumlah populasi yang digunakan tidak begitu berpengaruh terlalu besar, tetapi tetap terdapat korelasi antara nilai *objective function* dengan jumlah populasi.

BAB 4

GUI



Fitur:

- Load File: membaca file json berisi state hasil eksperimen.
- Play and Pause: memulai dan memberhentikan progress secara otomatis
- Drag and Drop: menampilkan visualisasi state yang dituju dengan meng-*drag and drop* progress bar
- Playback speed: dapat mengubah speed dengan pilihan 0.25, 0.5, 0.75, 1.0, 1.25. 1.5, 1.75, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, dan 128.0 dengan pendefinisian speed sebagai 1/speed terpilih.

BAB 5

Kesimpulan

Berdasarkan percobaan dari kode yang telah dibuat, dapat disimpulkan bahwa pencarian dengan algoritma *local search* yang bervariasi mulai dari *Hill Climbing* hingga *Genetic Algorithm* memiliki kelebihan dan kekurangannya masing-masing. Pemilihan algoritma akan mempengaruhi durasi yang didapatkan dan juga state value yang dihasilkan. Selain itu juga parameter yang digunakan untuk setiap algoritma juga berpengaruh terhadap nilai *objective function* yang dihasilkan. Berdasarkan eksperimen yang kita lakukan juga, dapat disimpulkan juga bahwa algoritma *local search* dengan hasil terbaik dan waktu yang tidak terlalu lama adalah ***Simulated Annealing***.

LAMPIRAN

Repository: https://github.com/Zechtro/Tubes1_AI

Pembagian Tugas

NIM	Responsibilities
13522061	Steepest Ascent HC, HC with Sideways Move, Random Restart HC, GUI, Report
13522075	Steepest Ascent HC, HC with Sideways Move, Random Restart HC, Stochastic HC, Simulated Annealing, Genetic Algorithm, Report
13522083	Steepest Ascent HC, HC with Sideways Move, Random Restart HC, GUI (bonus), Report
13522103	Visualisation, Genetic Algorithm, Report
13521154	Simulated Annealing, Stochastic HC, Report

DAFTAR PUSTAKA

Institut Teknologi Bandung. (n.d.). LMS Indonesia. https://edunex.itb.ac.id/courses/44315/preview/137704

Russell, Stuart J., and Peter Norvig. 2021. *Artificial Intelligence: A Modern Approach - Global Edition*. 4th ed. Harlow, United Kingdom: Pearson.

"Features of the Magic Cube." n.d. Magisch Vierkant. https://www.magischvierkant.com/three-dimensional-eng/magic-features/.

"Perfect Magic Cubes." n.d. https://www.trump.de/magic-squares/magic-cubes/cubes-1.html.

Wikipedia contributors. 2024. "Magic Cube." Wikipedia. May 27, 2024. https://en.wikipedia.org/wiki/Magic_cube.