

# **Tugas Besar 1 IF3170 Inteligensi Artifisial**

## **Pencarian Solusi Diagonal Magic Cube dengan Local Search**



Oleh:

13522125 - Satriadhikara Panji Yudhistira

13522128 - Mohammad Andhika Fadillah

13522148 - Auralea Alvinia Syaikha

13522162 - Pradipta Rafa Mahesa

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

## **DAFTAR ISI**

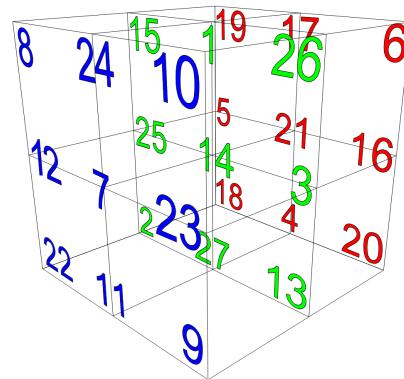
<b>DAFTAR ISI</b>	2
<b>Bab 1</b>	3
<b>    Deskripsi Persoalan</b>	3
<b>    Bab 2</b>	5
<b>        Pembahasan</b>	5
2.1 Pemilihan Objective Function	5
2.2 Penjelasan Algoritma Local Search	6
2.3 Hasil Eksperimen dan Analisis	27
<b>    Bab 3</b>	<b>39</b>
<b>        Kesimpulan</b>	<b>39</b>
3.1 Kesimpulan	39
3.2 Saran	40
<b>    Lampiran</b>	<b>41</b>
<b>    Referensi</b>	<b>42</b>

# 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:

<del>8 24 10</del>	<del>15 1 26</del>	<del>19 17 6</del>
<del>12 7 23</del>	<del>25 14 3</del>	<del>5 21 16</del>
<del>22 11 9</del>	<del>2 27 13</del>	<del>18 4 20</del>
<del>19 17 6</del>	<del>5 21 16</del>	<del>18 4 20</del>
<del>15 1 26</del>	<del>25 14 3</del>	<del>2 27 13</del>
<del>8 24 10</del>	<del>12 7 23</del>	<del>22 11 9</del>
<del>8 15 19</del>	<del>12 25 5</del>	<del>22 2 18</del>
<del>24 1 17</del>	<del>7 14 21</del>	<del>11 27 4</del>
<del>10 26 6</del>	<del>23 3 16</del>	<del>9 13 20</del>

- Diagonal yang dimaksud adalah yang dilingkari warna merah saja

Pada tugas kecil ini kita ditugasi untuk 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).

## Bab 2

### Pembahasan

#### 2.1 Pemilihan Objective Function

Objective function adalah suatu fungsi yang digunakan dalam optimasi untuk mengukur suatu solusi dalam memecahkan suatu masalah. Dalam konteks ini, objective function mengukur dan memberi nilai pada solusi-solusi yang diusulkan, ini bisa dipakai untuk membandingkan solusi-solusi lainnya dengan tujuan menemukan solusi terbaik.  $M$  Dalam menentukan objective function pada permasalahan ini, kita perlu mendapatkan magic number yaitu dengan setiap baris, kolom, tiang, diagonal bidang, dan diagonal ruang pada kubus memiliki jumlah yang sama. Maka dengan mendefinisikan magic number sebagai pada kubus berukuran  $n \times n \times n$ , kita dapat menghitung dengan rumus

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

Dalam kubus dengan  $n = 5$ , kita mendapatkan  $M$  sama dengan 315. Karena objective function dirancang untuk mengukur seberapa dekat state saat ini dengan solusi optimal. Fungsi ini memberikan nilai numeric yang menunjukkan total penyimpangan dari magic number pada semua garis yang harus dipenuhi. Objective function  $f$  dapat didefinisikan sebagai total penyimpangan absolut antara jumlah angka pada setiap garis dengan magic number .

$$f = \sum_{semua\ garis} |S_{garis} - M|$$

Di mana:

$S_{garis}$  adalah jumlah angka pada garis (baris, kolom, tiang, diagonal)

Alasan kita memilih fungsi ini adalah karena fungsi tersebut mencakup semua aspek penting dalam permasalahan, yaitu baris, kolom, tiang, dan diagonal, baik bidang maupun ruang. Dengan demikian, perhitungan penyimpangan tidak melewatkannya satupun garis yang harus memenuhi magic number. Selain itu, fungsi ini relatif mudah dihitung, karena hanya memerlukan operasi penjumlahan dan pengurangan sederhana.

Dengan menghitung penyimpangan absolut, kita dapat memperoleh ukuran seberapa jauh setiap garis dari kondisi ideal. Nilai absolut digunakan agar penyimpangan positif maupun negatif dihitung secara setara, karena yang penting adalah seberapa besar

perbedaannya, bukan arahnya. Misalnya, jika suatu garis memiliki jumlah angka 310, penyimpangannya adalah  $|310 - 315| = 5$ . Begitu juga kalau jumlahnya 320, penyimpangannya tetap  $|320 - 315| = 5$ .

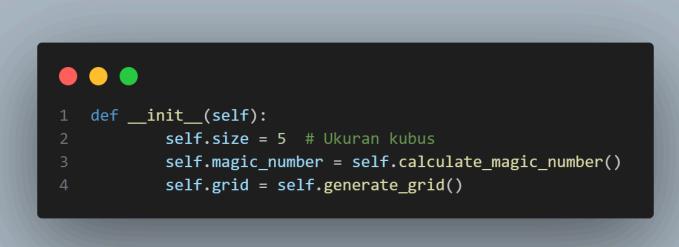
Oleh karena itu, fungsi ini memberikan panduan yang jelas bagi algoritma local search untuk bergerak menuju solusi yang lebih baik. Tujuan utama dari penggunaan objective function ini adalah meminimalkan nilai  $f$  atau mencapai kondisi optimal. Semakin kecil nilai sehingga konfigurasi kubus mendekati  $f$ , semakin dekat konfigurasi kubus dengan solusi yang diinginkan, yaitu Diagonal Magic Cube yang sempurna.

Dengan adanya objective function ini, algoritma pencarian lokal dapat melakukan evaluasi terhadap setiap perubahan atau langkah yang diambil. Misalnya, ketika algoritma menukar posisi dua angka dalam kubus, nilai  $f$  nilai  $f$  dihitung ulang untuk konfigurasi baru. Jika berkurang, artinya konfigurasi baru lebih baik daripada sebelumnya, dan algoritma dapat melanjutkan pencarian dari konfigurasi tersebut. Jika tidak, algoritma mungkin akan mencoba perubahan lain atau, dalam kasus algoritma seperti Simulated Annealing, mungkin tetap menerima konfigurasi yang kurang baik dengan probabilitas tertentu untuk menghindari jebakan local optimum.

## 2.2 Penjelasan Algoritma Local Search

### 2.2.1 Kelas Cube

Kelas Cube merupakan kelas untuk membuat dan mengevaluasi sebuah kubus ajaib (magic cube). Kubus ajaib adalah susunan angka dalam bentuk kubus tiga dimensi, di mana jumlah angka dalam setiap baris, kolom, dan diagonal harus sama. Berikut adalah penjelasan mendetail tentang komponen dan fungsionalitas dari kode ini:

No	Kode	Penjelasan
1	 <pre>1 def __init__(self): 2     self.size = 5 # Ukuran kubus 3     self.magic_number = self.calculate_magic_number() 4     self.grid = self.generate_grid()</pre>	Fungsi untuk menginisialisasi kelas

2	 <pre> 1 def generate_grid(self): 2     numbers = list( 3         range(1, self.size**3 + 1) 4     ) # Mengisi list dengan angka 1 sampai size^3 yaitu 125 kemudian diacak 5     random.shuffle(numbers) 6     grid = [ 7         [[0 for _ in range(self.size)] for _ in range(self.size)] 8         for _ in range(self.size) 9     ] 10    for x in range(self.size): 11        for y in range(self.size): 12            for z in range(self.size): 13                grid[x][y][z] = numbers.pop(0) 14 15    return grid </pre>	Fungsi untuk meng-generate grid 5x5x5 secara random untuk dijadikan state awal dari suatu magic cube
3	 <pre> 1 def generate_grid_from_input(self, input_grid): 2     grid = [ 3         [[0 for _ in range(self.size)] for _ in range(self.size)] 4         for _ in range(self.size) 5     ] 6     for x in range(self.size): 7         for y in range(self.size): 8             for z in range(self.size): 9                 grid[x][y][z] = input_grid[x][y][z] 10 11    return grid </pre>	
4	 <pre> 1 def calculate_magic_number(self): 2     return self.size * (self.size**3 + 1) // 2 </pre>	Fungsi untuk menghitung magic number dari suatu magic cube, yang pada kasus ini adalah 315 (kubus 5x5x5)
5	 <pre> 1 def print_grid(self): 2     for x in range(self.size): 3         for y in range(self.size): 4             for z in range(self.size): 5                 print(self.grid[x][y][z], end=" ") 6             print() 7         print() </pre>	Fungsi untuk menampilkan grid dari state kubus saat ini
6	 <pre> 1 def get_row(self, x, y): 2     return self.grid[x][y] 3 4 def get_column(self, x, z): 5     return [self.grid[i][x][z] for i in range(self.size)] 6 7 def get_pillar(self, y, z): 8     return [self.grid[y][i][z] for i in range(self.size)] </pre>	Fungsi yang secara berurutan mengembalikan suatu baris,kolom, dan tiang tertentu

7

```

● ● ●
1 def get_diagonals(self):
2     diagonals = []
3
4     for z in range(self.size):
5         diag1 = [self.grid[i][i][z] for i in range(self.size)]
6         diag2 = [self.grid[i][self.size - 1 - i][z] for i in range(self.size)]
7         diagonals.append((diag1, diag2))
8
9     for y in range(self.size):
10        diag1 = [self.grid[i][y][i] for i in range(self.size)]
11        diag2 = [self.grid[self.size - 1 - i][y][i] for i in range(self.size)]
12        diagonals.append((diag1, diag2))
13
14    for x in range(self.size):
15        diag1 = [self.grid[x][i][i] for i in range(self.size)]
16        diag2 = [self.grid[x][self.size - 1 - i][i] for i in range(self.size)]
17        diagonals.append((diag1, diag2))
18
19    return diagonals

```

Fungsi yang mengembalikan semua diagonal dari kubus

8

```

● ● ●
1 def evaluate_cube(self):
2     total_deviation = 0
3     magic_number = self.magic_number
4
5     # Menghitung penyimpangan untuk setiap baris
6     for x in range(self.size):
7         for y in range(self.size):
8             row = self.get_row(x, y)
9             sum_row = sum(row)
10            deviation = abs(sum_row - magic_number)
11            total_deviation += deviation
12
13     # Menghitung penyimpangan untuk setiap kolom
14     for y in range(self.size):
15         for z in range(self.size):
16             column = self.get_column(y, z)
17             sum_column = sum(column)
18             deviation = abs(sum_column - magic_number)
19             total_deviation += deviation
20
21     # Menghitung penyimpangan untuk setiap tiang
22     for x in range(self.size):
23         for z in range(self.size):
24             pillar = self.get_pillar(x, z)
25             sum_pillar = sum(pillar)
26             deviation = abs(sum_pillar - magic_number)
27             total_deviation += deviation
28
29     # Menghitung penyimpangan untuk setiap diagonal
30     diagonals = self.get_diagonals()
31     for diag_pair in diagonals:
32         diag1, diag2 = diag_pair
33         sum_diag1 = sum(diag1)
34         deviation1 = abs(sum_diag1 - magic_number)
35         total_deviation += deviation1
36         if diag2:
37             sum_diag2 = sum(diag2)
38             deviation2 = abs(sum_diag2 - magic_number)
39             total_deviation += deviation2
40
41     return total_deviation

```

Fungsi untuk menghitung nilai objektif dari state kubus saat ini

9

```

● ● ●
1 def evaluate_cube_on_grid(self, grid):
2     total_deviation = 0
3     magic_number = self.magic_number
4     # Check rows
5     for x in range(self.size):
6         for y in range(self.size):
7             row = grid[x][y]
8             total_deviation += abs(sum(row) - magic_number)
9
10    # Check columns
11    for y in range(self.size):
12        for z in range(self.size):
13            column = [grid[i][y][z] for i in range(self.size)]
14            total_deviation += abs(sum(column) - magic_number)
15
16    # Check pillars
17    for x in range(self.size):
18        for z in range(self.size):
19            pillar = [grid[x][i][z] for i in range(self.size)]
20            total_deviation += abs(sum(pillar) - magic_number)
21
22    # Check diagonals
23    for z in range(self.size):
24        diag1 = [grid[i][i][z] for i in range(self.size)]
25        diag2 = [grid[i][self.size - 1 - i][z] for i in range(self.size)]
26        total_deviation += abs(sum(diag1) - magic_number)
27        total_deviation += abs(sum(diag2) - magic_number)
28
29    for y in range(self.size):
30        diag1 = [grid[i][y][i] for i in range(self.size)]
31        diag2 = [grid[self.size - 1 - i][y][i] for i in range(self.size)]
32        total_deviation += abs(sum(diag1) - magic_number)
33        total_deviation += abs(sum(diag2) - magic_number)
34
35    for x in range(self.size):
36        diag1 = [grid[x][i][i] for i in range(self.size)]
37        diag2 = [grid[x][self.size - 1 - i][i] for i in range(self.size)]
38        total_deviation += abs(sum(diag1) - magic_number)
39        total_deviation += abs(sum(diag2) - magic_number)
40
41    return total_deviation

```

Fungsi untuk menghitung nilai objektif dari kubus lain, digunakan untuk menghitung nilai objektif dari neighbor

### 2.2.1 Steepest Ascent Hill-climbing

Algoritma *steepest ascent hill climbing* adalah metode optimasi yang bertujuan menemukan solusi terbaik dengan terus bergerak menuju solusi yang memberikan nilai fungsi tujuan yang lebih baik (lebih optimal) pada setiap langkah. Algoritma ini dimulai dari suatu solusi awal dan mengevaluasi semua kemungkinan tetangga (solusi yang bisa dicapai dari solusi saat ini dengan satu perubahan kecil), lalu memilih tetangga yang memberikan peningkatan terbesar. Jika tetangga tersebut lebih baik dari solusi saat ini, algoritma bergerak ke solusi itu dan mengulangi proses. Algoritma berhenti ketika tidak ada lagi tetangga yang lebih baik dari solusi saat ini, menandakan telah tercapai titik optimum (yang bisa bersifat lokal atau global).

```
1  def steepest_ascent_hill_clim (self, max_iteration =1000):
2      burrent_deviatio = self.evaluatecsub ()
3      best_grid = copy.deepcopy(self.grid) # Keep track of the best grid foun
4          d
5      if current_deviatio == 0:
6          nprint("Already at global optimu ")
7          return self.grid, current_deviatio , 0
8          n
9      for iteration in range(max_iteration ):
10         best_deviatio = cuurent_deviatio
11         found_improvemen n= False
12         t
13         # Generate all unique neighbors by swapping each unique pair of element
14         for x1 in range(self.size):
15             for y1 in range(self.size):
16                 for z1 in range(self.size):
17                     for x2 in range(self.size):
18                         for y2 in range(self.size):
19                             for z2 in range(self.size):
20                                 if (x1, y1, z1) >= (
21                                     x2,
22                                     y2,
23                                     z2,
24                                 ): # Avoid duplicate and self-swap
25                                     continue
26
27         # Create a copy of the grid and perform the swa
28         neighbor = copy.deepcopy(self.grid)
29         neighbor[x1][y1][z1], neighbor[x2][y2][z2] = (
30             neighbor[x2][y2][z2],
31             neighbor[x1][y1][z1],
32
33     )
34         # Evaluate the neighbo
35         deviation = self.evaluate_cube_on_gri (neighbor)
36             d
37         # Update the best neighbor foun
38         if deviation < best_deviatio :
39             best_deviatio = deviation
40             best_grid = neighbor
41             found_improvemen = True
42             t
43         # Move to the best neighbor if it's better than the curren
44         if found_improvemen and best_deviatio < current_deviatio :
45             tself.grid = best_gridn
46             current_deviatio = best_deviatio
47             print(f"Iteration {iteration + 1}: Deviation = {current_deviatio }")
48         else:
49             # No improvement found, we may have reached a local optimu
50             if current_deviatio == 0:
51                 nprint("Reached global optimu ")
52             else:
53                 print("Reached local optimu ")
54             break
55
56         print(self.grid, current_deviatio , iteration + 1)
57         return self.grid,ncurrent_deviatio , (iteration + 1)
58
59
```

Pada implementasi algoritma *steepest ascent hill climbing* diatas, diterapkan tahapan-tahapan berikut:

1. Evaluasi awal:

- Algoritma ini memulai dengan mengevaluasi deviasi awal dari konfigurasi kubus (`self.grid`).
- Jika deviasi awal adalah 0, maka algoritma langsung berhenti karena sudah mencapai solusi optimal global.

2. Proses iterasi:

- Untuk setiap iterasi, algoritma mencoba semua kemungkinan pasangan elemen dalam kubus dan melakukan *swap* (pertukaran posisi) satu per satu.
- Setiap kali elemen-elemen dalam kubus ditukar, algoritma mengevaluasi konfigurasi kubus baru tersebut untuk menghitung deviasi.
- Jika deviasi dari hasil swap lebih rendah dari deviasi saat ini, algoritma memperbarui konfigurasi kubus yang terbaik dan mencatatnya sebagai *found improvement*.

3. Pembaruan dan penghentian:

- Jika ada perbaikan dari *swap* terbaik, algoritma memperbarui konfigurasi kubus menjadi *state* yang terbaik.
- Jika tidak ada perbaikan lebih lanjut yang ditemukan dalam satu iterasi, algoritma berhenti dan mengembalikan hasilnya.
- Jika algoritma berhasil mencapai deviasi 0, maka dianggap sebagai solusi optimal global.
- Jika algoritma berhenti karena tidak ada perbaikan lagi, maka dianggap sebagai solusi optimal lokal.

4. Pengembalian hasil:

- Algoritma mengembalikan konfigurasi kubus terbaik yang ditemukan.

### 2.2.2 Hill-climbing with Sideways Move

Algoritma sideways hill climbing adalah varian dari algoritma hill climbing yang dirancang untuk mengatasi masalah ketika algoritma menemui sebuah flat, yaitu

area pencarian di mana semua tetangganya memiliki nilai evaluasi yang sama. Dalam algoritma ini, pencarian dapat tetap bergerak ke tetangga dengan nilai evaluasi yang sama (sideways move) agar tidak terjebak di dataran tinggi dan dapat melanjutkan pencarian menuju solusi yang lebih baik. Namun, langkah sideways ini biasanya dibatasi oleh jumlah iterasi tertentu untuk mencegah pencarian berlarut-larut tanpa hasil. Tujuan akhirnya adalah mencapai puncak lokal atau global di ruang solusi dengan meminimalkan atau memaksimalkan nilai fungsi evaluasi. Untuk Algoritmanya adalah sebagai berikut:

#### 2.2.2.1 Fungsi Utama

```

● ● ●
1 def sideways(self, max_iterations=1000, max_sideways=100):
2     past_cubes = []
3     current_deviation = self.evaluate_cube()
4     best_grid = copy.deepcopy(self.grid)
5     message = ""
6     iterations_history = [{"iteration": 0, "obj_value": current_deviation}]
7     n_sideways = 0
8     if current_deviation == 0:
9         print("Already at global optimum")
10        return self.grid, current_deviation, 0
11
12    for iteration in range(max_iterations):
13        best_deviation = current_deviation
14        found_improvement = False
15        past_cubes.append(self.grid)
16        # Generate all unique neighbors by swapping each unique pair of elements
17        for x1 in range(self.size):
18            for y1 in range(self.size):
19                for z1 in range(self.size):
20                    for x2 in range(self.size):
21                        for y2 in range(self.size):
22                            for z2 in range(self.size):
23                                if (x1, y1, z1) >= (x2,
24                                         y2,
25                                         z2,
26                                         ): # Avoid duplicate and self-swaps
27                                    continue
28
29                                # Create a copy of the grid and perform the swap
30                                neighbor = copy.deepcopy(self.grid)
31                                neighbor[x1][y1][z1], neighbor[x2][y2][z2] = (
32                                         neighbor[x2][y2][z2],
33                                         neighbor[x1][y1][z1],
34                                         )
35
36

```

```

37             # Evaluate the neighbor
38             deviation = self.evaluate_cube_on_grid(neighbor)
39
40             # Update the best neighbor found
41             if (
42                 deviation <= best_development
43                 and not past_cubes.isIn(neighbor)
44             ):
45                 best_development = deviation
46                 best_grid = neighbor
47                 found_improvement = True
48
49             # Move to the best neighbor if it's better than the current
50             if found_improvement and best_development <= current_development and n_sideways < max_sideways:
51                 if best_development < current_development:
52                     n_sideways=0
53                     past_cubes.reset()
54                 else:
55                     n_sideways+=1
56                     self.grid = best_grid
57                     current_development = best_development
58                     print(
59                         f"Iteration {iteration + 1}: Deviation = {current_development}, Visited Cubes : {past_cubes.size}"
60                     )
61                     iterations_history.append(
62                         {"iteration": iteration + 1, "obj_value": current_development}
63                     )
64             else:
65                 if current_development == 0:
66                     message = "Reached global optimum"
67                 else:
68                     message = "Reached local optimum"
69                 break
70
71             print(self.grid, current_development, iteration + 1)
72         return (
73             self.grid,
74             current_development,
75             (iteration + 1),
76             message,
77             iterations_history,
78         )

```

Fungsi ini adalah fungsi utama dari Algoritma Hill Climbing with Sideways Move yang cara kerjanya adalah sebagai berikut.

1. Inisialisasi:

Proses dimulai dengan menginisialisasi parameter masukan dan variabel yang diperlukan untuk jalannya algoritma:

- a. Parameter Masukan:

- i. max\_iterations: Menentukan jumlah maksimum iterasi yang dapat dilakukan oleh algoritma sebelum berhenti. Nilai default yang sering digunakan adalah 1000 iterasi.
- ii. max\_sideways: Menentukan batas jumlah pergerakan sideways yang dapat dilakukan secara berturut-turut. Defaultnya adalah 100, yang memastikan algoritma tidak terjebak dalam siklus sideways terlalu lama.

- b. Variabel Utama:

- i. past\_cubes: Array atau daftar yang digunakan untuk melacak semua state kubus yang pernah dikunjungi selama proses

- pencarian. Hal ini bertujuan untuk menghindari looping pada state yang sama berulang kali.
- ii. **current\_deviation**: Variabel yang merepresentasikan nilai fungsi objektif dari state kubus saat ini. Ini digunakan untuk mengevaluasi apakah state tersebut lebih baik, sama, atau lebih buruk dibandingkan tetangga-tetangganya.
  - iii. **best\_grid**: Salinan grid dari status kubus saat ini yang digunakan sebagai acuan untuk membandingkan state-state tetangga. Salinan ini sangat penting untuk mempermudah pemilihan tetangga terbaik tanpa memodifikasi state asli.
  - iv. **n\_sideways**: Variabel penghitung yang digunakan untuk melacak jumlah gerakan sideways yang telah dilakukan. Pada awal inisialisasi, nilainya diatur ke 0. Ini membantu algoritma memastikan bahwa jumlah sideways move yang diizinkan tidak terlampaui.
2. Evaluasi Tetangga:
- Pada tahap ini, algoritma akan mengevaluasi semua tetangga dari state saat ini dengan tujuan menemukan tetangga dengan nilai fungsi objektif tertinggi. Proses evaluasi ini melibatkan:
- a. **Mencoba Semua Kombinasi Pertukaran 2 Angka**: Algoritma melakukan pertukaran antara dua angka dalam grid untuk menghasilkan state-state tetangga. Setiap state tetangga dievaluasi untuk mengukur seberapa baik mereka berdasarkan fungsi objektif yang digunakan.
  - b. **Pencatatan State Terbaik**: Jika algoritma menemukan tetangga dengan nilai fungsi objektif yang lebih tinggi daripada **current\_deviation**, state dan nilai tersebut dicatat sebagai kandidat untuk langkah berikutnya.
3. Pemilihan Solusi Berikutnya:

Setelah mengevaluasi semua tetangga, algoritma memutuskan langkah selanjutnya berdasarkan hasil evaluasi:

- a. **Pindah ke Tetangga dengan Nilai Lebih Tinggi**: Jika terdapat tetangga dengan nilai fungsi objektif yang lebih tinggi dari **current\_deviation**, algoritma akan bergerak ke state tersebut,

memperbarui nilai `current_deviation`, `best_grid`, dan menyimpan state di `past_cubes`.

- b. Gerakan Sideways: Jika tidak ditemukan tetangga dengan nilai lebih tinggi tetapi ada tetangga dengan nilai yang sama, algoritma melakukan sideways move ke tetangga tersebut. Namun, ini hanya dilakukan jika jumlah pergerakan sideways yang telah dilakukan (`n_sideways`) masih di bawah `max_sideways`. Setelah melakukan sideways move, variabel `n_sideways` diincrement untuk mencatat pergerakan tersebut.
4. Periksa Kondisi Berhenti:
- Algoritma memiliki beberapa kondisi berhenti yang memastikan algoritma tidak berjalan tanpa henti:
- a. Tidak Ada Tetangga yang Lebih Baik atau Sama: Jika semua tetangga memiliki nilai yang lebih rendah dari `current_deviation`, algoritma berhenti karena sudah mencapai puncak (peak) lokal.
  - b. Maksimal Sideways Move Tercapai: Jika jumlah sideways move telah mencapai `max_sideways`, algoritma berhenti, menganggap bahwa solusi tidak dapat diperbaiki lebih jauh dengan gerakan sideways.

### 2.2.2.1 Kelas CustomList



```
1  from concurrent.futures import ThreadPoolExecutor
2
3  class CustomList:
4      def __init__(self):
5          self.size = 0 # Ukuran kubus
6          self.arr = []
7
8      def add(self,element):
9          self.arr.append(element)
10     self.size += 1
11
12     def reset(self):
13         self.arr = []
14         self.size = 0
15
16
17     def isIn(self,element):
18         if self.size <= 5:
19             return self.__checkElement(element)
20         else:
21             return self.__checkElementConcurrent(element)
22
23     def __checkElement(self,element):
24         for el in self.arr:
25             if element == el:
26                 return True
27
28
29     def __checkElementConcurrent(self,element):
30         # Ensure we always divide into 4 chunks
31         chunk_size = max(1, len(self.arr) // 4)
32         chunks = [self.arr[i:i + chunk_size] for i in range(0, len(self.arr), chunk_size)]
33
34         with ThreadPoolExecutor() as executor:
35             # Submit tasks for each chunk
36             futures = [executor.submit(self.__check_in_chunk, chunk, element) for chunk in chunks]
37
38             # Wait for any future to complete successfully
39             for future in futures:
40                 if future.result(): # If any future returns True
41                     return True
42
43             return False
44
45     def __check_in_chunk(self, chunk, element):
46         return element in chunk
```

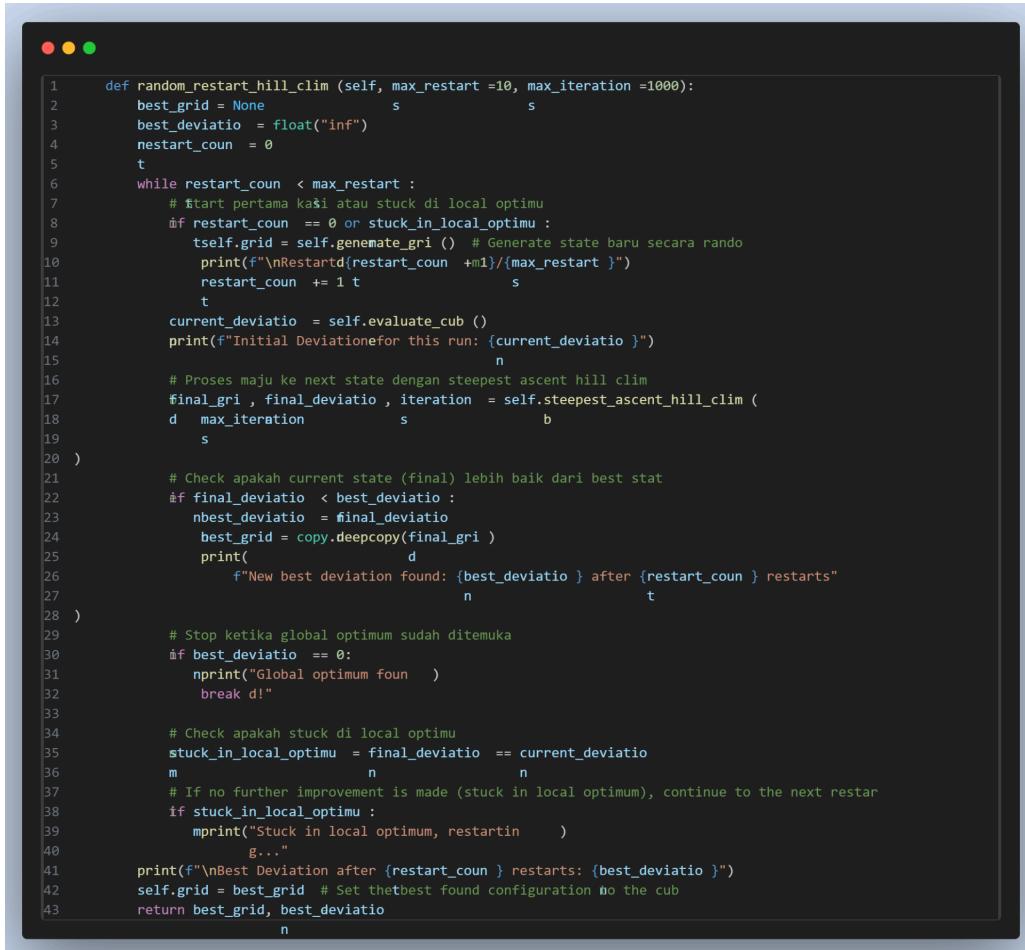
Kelas CustomList berfungsi sebagai implementasi kustom dari list yang memiliki kemampuan untuk menambahkan elemen, mereset list, dan mengecek apakah elemen tertentu ada di dalam list tersebut. Kelas ini digunakan untuk variabel past\_cubes pada fungsi utama hill climbing with sideways move.

1. Inisialisasi Kelas: Kelas ini memiliki dua atribut utama: size, yang melacak jumlah elemen saat ini di dalam list, dan arr, yang berfungsi sebagai tempat penyimpanan elemen-elemen tersebut.

2. Metode Penambahan Elemen: Metode `add` memungkinkan penambahan elemen baru ke dalam list, sekaligus memperbarui nilai `size` untuk merefleksikan jumlah elemen terkini.
3. Metode Reset: Metode `reset` digunakan untuk mengosongkan arr dan mengatur `size` kembali ke 0, mengembalikan list ke kondisi awal.
4. Metode Pengecekan Elemen: Metode `isIn` mengevaluasi apakah elemen tertentu ada di dalam list. Jika ukuran arr kurang dari atau sama dengan 5, pencarian dilakukan secara sekuensial. Untuk list yang lebih besar, pencarian dilakukan secara paralel menggunakan multi-threading, yaitu metode `_checkElementConcurrent`.
5. Pengecekan Linear: Pencarian elemen secara linear dilakukan dengan iterasi satu per satu melalui list. Metode ini efisien untuk list kecil.
6. Pengecekan Paralel: Untuk list besar, `_checkElementConcurrent` membagi arr menjadi beberapa bagian (chunk). Setiap chunk diproses dalam thread terpisah menggunakan `ThreadPoolExecutor`. Jika salah satu thread menemukan elemen yang dicari, hasil pencarian segera dikembalikan, dan pencarian di thread lain dihentikan.
7. Pencarian dalam Chunk: Setiap chunk diperiksa secara individual, dan elemen dicari di dalamnya. Jika elemen ditemukan di dalam chunk, hasil pencarian dikembalikan sebagai `True`.
8. Manfaat Multi-threading: Penggunaan `ThreadPoolExecutor` memungkinkan pencarian paralel yang meningkatkan kinerja ketika list sangat besar. Metode ini membagi pekerjaan pencarian ke beberapa thread, yang kemudian berjalan secara bersamaan. Pendekatan ini membuat pencarian lebih cepat dengan memanfaatkan prosesor multi-core.

### 2.2.3 Random Restart Hill-climbing

Algoritma *Random Restart Hill-climbing* memperluas pendekatan hill climbing dengan melakukan beberapa kali restart dari titik acak ketika solusi lokal optimum ditemukan. Dengan demikian, metode ini berusaha menemukan solusi optimal global dengan menghindari jebakan solusi lokal.



```

1  def random_restart_hill_clim (self, max_restart =10, max_iteration =1000):
2      best_grid = None
3      best_deviatio = float("inf")
4      restart_coun = 0
5      t
6      while restart_coun < max_restart :
7          # Start pertama kali atau stuck di local optimu
8          if restart_coun == 0 or stuck_in_local_optimu :
9              tself.grid = self.genemate_gri () # Generate state baru secara rando
10             print(f"\nRestart{restart_coun +1}/{max_restart }")
11             restart_coun += 1 t
12             t
13             current_deviatio = self.evaluate_cub ()
14             print(f"Initial Deviation for this run: {current_deviatio }")
15             n
16             # Proses maju ke next state dengan steepest ascent hill clim
17             final_gri , final_deviatio , iteration = self.steepest_ascent_hill_clim (
18                 max_iteration
19                 s
20             )
21             # Check apakah current state (final) lebih baik dari best stat
22             if final_deviatio < best_deviatio :
23                 best_deviatio = final_deviatio
24                 best_grid = copy.deepcopy(final_gri )
25                 print(
26                     f"New best deviation found: {best_deviatio } after {restart_coun } restarts"
27                     n
28             )
29             # Stop ketika global optimum sudah ditemuka
30             if best_deviatio == 0:
31                 nprint("Global optimum foun ")
32                 break d!
33
34             # Check apakah stuck di local optimu
35             stuck_in_local_optimu = final_deviatio == current_deviatio
36             m
37             n
38             # If no further improvement is made (stuck in local optimum), continue to the next restart
39             if stuck_in_local_optimu :
40                 mprint("Stuck in local optimum, restartin ")
41                 g...
42             print(f"\nBest Deviation after {restart_coun } restarts: {best_deviatio }")
43             self.grid = best_grid # Set the best found configuration to the cub
44             return best_grid, best_deviatio
45             n

```

Pada implementasi algoritma *random restart* diatas, diterapkan tahapan-tahapan berikut:

### 1. Inisialisasi

- Mulai dengan inisialisasi variabel *best\_grid* (untuk menyimpan konfigurasi kubus terbaik) dan *best\_deviation* (untuk menyimpan deviasi terendah yang ditemukan sejauh ini).
- *restart\_count* juga diinisialisasi untuk mencatat berapa kali *restart* dilakukan.

### 2. Proses random restart

- Algoritma akan terus berjalan sampai mencapai jumlah restart maksimum yang ditentukan oleh *max\_restarts*.
- Pada setiap restart atau jika algoritma terjebak di solusi lokal, algoritma menghasilkan konfigurasi acak baru dengan memanggil *self.generate\_grid()* dan menghitung deviasinya.

### 3. Penerapan steepest ascent hill climbing

- Setelah menghasilkan konfigurasi acak, algoritma menjalankan metode steepest ascent hill climbing untuk mencapai solusi optimal lokal dari titik awal tersebut.
- Hasil steepest ascent hill climb adalah konfigurasi terbaik yang dicapai dalam iterasi tersebut (final\_grid), deviasinya (final\_deviation), dan jumlah iterasi yang dijalankan.

#### 4. Penyimpanan solusi terbaik

- Jika deviasi dari konfigurasi hasil lebih baik dari best\_deviation yang disimpan, algoritma memperbarui best\_grid dan best\_deviation dengan hasil ini.
- Jika deviasi mencapai 0 (global optimum), algoritma langsung berhenti karena solusi terbaik sudah ditemukan.

#### 5. Deteksi optimum lokal

- Jika final\_deviation sama dengan current\_deviation, artinya tidak ada perbaikan lagi yang bisa dilakukan dan algoritma dianggap terjebak di solusi lokal.
- Dalam kasus ini, restart berikutnya akan dimulai dari titik acak yang baru.

#### 6. Pengembalian hasil

- Setelah semua restart selesai atau ketika global optimum ditemukan, algoritma mengembalikan best\_grid (konfigurasi terbaik) dan best\_deviation yang telah ditemukan.

#### 2.2.4 Stochastic Hill-climbing

Algoritma *Stochastic Hill-climbing* merupakan varian dari hill climbing dengan pendekatan acak untuk memilih solusi tetangga. Alih-alih memeriksa semua tetangga seperti pada steepest ascent hill climbing, algoritma ini hanya memilih tetangga secara acak dan bergerak ke arah yang lebih baik jika ditemukan perbaikan.



```

1  def stochastic_hill_clim (self, max_iteration =1000, max_attempt =100):
2      burrent_deviatio_ = self.evaluate_cub ()           s
3      # best_grid = copy.deepcopy(self.grid)
4      message = ""
5      iterations_histoir = [{"iteratio_ : 0, "obj_valu_ : current_deviatio_}"]
6      y
7      n
8      if current_deviatio_ == 0:
9          nmessage = "Already at global optimu"
10         return self.grid, current_deviatio_ , 0, message, iterations_histoir
11         n
12         y
13     for iteration in range(max_iteration ):
14         found_improvemen_ =False
15         t
16         for attempt in range(max_attempt ):
17             # Random point selection for swa
18             x1, y1, z1 = (
19                 random.randint(0, self.size - 1),
20                 random.randint(0, self.size - 1),
21                 random.randint(0, self.size - 1),
22             )
23             x2, y2, z2 = (
24                 random.randint(0, self.size - 1),
25                 random.randint(0, self.size - 1),
26                 random.randint(0, self.size - 1),
27             )
28             # Avoid self-swap
29             while (x1, y1, z1) == (x2, y2, z2):
30                 x2, y2, z2 = (
31                     random.randint(0, self.size - 1),
32                     random.randint(0, self.size - 1),
33                     random.randint(0, self.size - 1),
34             )
35             neighbor = copy.deepcopy(self.grid)
36             neighbor[x1][y1][z1], neighbor[x2][y2][z2] = (
37                 neighbor[x2][y2][z2],
38                 neighbor[x1][y1][z1],
39             )
40             deviation = self.evaluate_cube_on_gri (neighbor)
41             d
42             if deviation < current_deviatio_ :
43                 self.grid =neighbor
44                 current_deviatio_ = deviation
45                 found_improvemen_ = True
46                 iterations_histoir.append(
47                     {"iteratio_ : iteration + 1, "obj_valu_ : current_deviatio_}
48                     n
49                 )
50             print(
51                 f"Iteration {iteration + 1}, Attempt {attempt + 1}: Deviation = {current_deviatio_}"
52                 n
53             )
54             break
55             if not found_improvemen_ :
56                 message = "Reached local optimu"
57                 break
58             if current_deviatio_ == 0:
59                 nmessage = "Reached global optimu"
60                 break
61                 m"
62
63             return (
64                 self.grid,
65                 current_deviatio_ ,
66                 niteration + 1),
67                 message,
68                 iterations_histoir ,
69             ) y

```

Pada implementasi algoritma *stochastic hill climbing* diatas, diterapkan tahapan-tahapan berikut:

1. Inisialisasi

- Algoritma dimulai dengan mengevaluasi deviasi dari konfigurasi kubus awal. Jika deviasi awal sudah nol, algoritma berhenti karena solusi optimal global sudah ditemukan.
- Algoritma juga menyiapkan riwayat iterasi untuk mencatat deviasi selama proses optimasi.

## 2. Loop iterasi utama

- Algoritma berjalan hingga mencapai batas max\_iterations atau menemukan solusi optimal (deviasi = 0).
- Pada setiap iterasi, variabel found\_improvement disetel ke False sebagai penanda apakah perbaikan ditemukan pada iterasi tersebut.

## 3. Loop percobaan (attempts)

- Dalam setiap iterasi, algoritma melakukan hingga max\_attempts percobaan untuk menemukan tetangga yang lebih baik.
- Pada setiap percobaan, algoritma memilih dua titik acak dalam kubus (dengan menghindari memilih titik yang sama) dan menukar elemen pada titik-titik tersebut untuk menghasilkan konfigurasi tetangga.

## 4. Evaluasi tetangga

- Setelah konfigurasi tetangga dihasilkan, algoritma mengevaluasi deviasi dari konfigurasi tersebut.
- Jika deviasi tetangga lebih kecil dari deviasi konfigurasi saat ini, algoritma menerima konfigurasi tetangga sebagai konfigurasi baru, memperbarui deviasi, dan mencatat perbaikan.
- Jika perbaikan ditemukan, algoritma keluar dari loop attempts dan lanjut ke iterasi berikutnya.

## 5. Penghentian dan penanganan optimum lokal

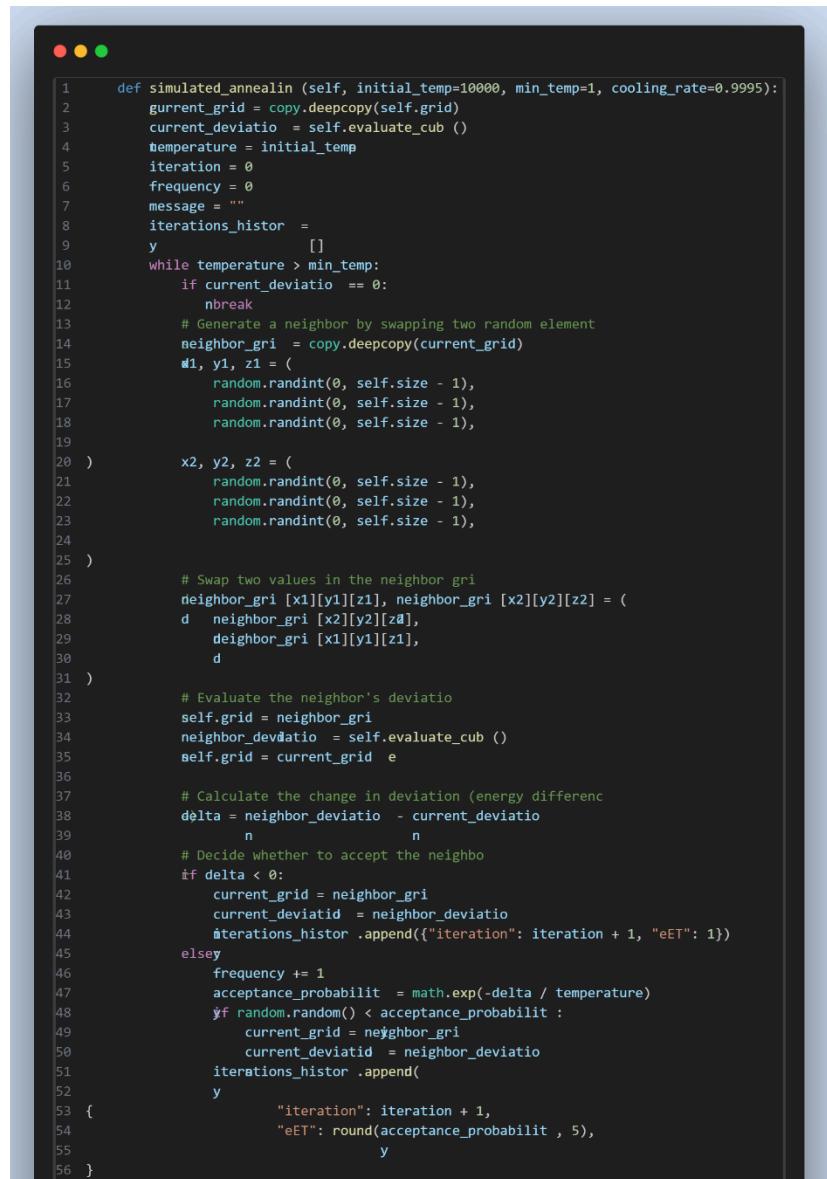
- Jika tidak ada perbaikan yang ditemukan dalam semua percobaan di satu iterasi, artinya algoritma mencapai solusi lokal optimum, dan loop utama berakhir dengan pesan local optimum.
- Jika deviasi mencapai nol, algoritma berhenti lebih awal dengan pesan bahwa solusi optimal global telah tercapai.

## 6. Pengembalian hasil

- Algoritma mengembalikan konfigurasi terbaik yang ditemukan, deviasi akhir, jumlah iterasi yang dilakukan, pesan hasil, dan riwayat deviasi tiap iterasi.

## 2.2.5 Simulated Annealing

Algoritma *Simulated Annealing* adalah metode optimasi yang meniru proses pendinginan dalam fisika untuk menemukan solusi optimal. Tujuannya adalah untuk meminimalkan nilai deviasi dari konfigurasi kubus 3D dengan memungkinkan eksplorasi solusi suboptimal sementara, sehingga algoritma dapat menghindari terjebak pada solusi lokal dan lebih berpeluang mencapai solusi global.



```
 1 def simulated_annealin (self, initial_temp=10000, min_temp=1, cooling_rate=0.9995):
 2     current_grid = copy.deepcopy(self.grid)
 3     current_deviatio = self.evaluate_cub ()
 4     temperature = initial_temp
 5     iteration = 0
 6     frequency = 0
 7     message = ""
 8     iterations_histor =
 9     y
10     while temperature > min_temp:
11         if current_deviatio == 0:
12             nbreak
13             # Generate a neighbor by swapping two random element
14             neighbor_gri = copy.deepcopy(current_grid)
15             x1, y1, z1 = (
16                 random.randint(0, self.size - 1),
17                 random.randint(0, self.size - 1),
18                 random.randint(0, self.size - 1),
19             )
20             x2, y2, z2 = (
21                 random.randint(0, self.size - 1),
22                 random.randint(0, self.size - 1),
23                 random.randint(0, self.size - 1),
24             )
25             # Swap two values in the neighbor gri
26             neighbor_gri [x1][y1][z1], neighbor_gri [x2][y2][z2] = (
27                 neighbor_gri [x2][y2][z1],
28                 neighbor_gri [x1][y1][z1],
29                 d
30             )
31             # Evaluate the neighbor's deviation
32             self.grid = neighbor_gri
33             neighbor_deviatio = self.evaluate_cub ()
34             self.grid = current_grid
35
36             # Calculate the change in deviation (energy difference)
37             delta = neighbor_deviatio - current_deviatio
38             n
39             n
40             # Decide whether to accept the neighbor
41             if delta < 0:
42                 current_grid = neighbor_gri
43                 current_deviatid = neighbor_deviatio
44                 iterations_histor .append({"iteration": iteration + 1, "eET": 1})
45             else:
46                 frequency += 1
47                 acceptance_probabilit = math.exp(-delta / temperature)
48                 if random.random() < acceptance_probabilit :
49                     current_grid = neighbor_gri
50                     current_deviatid = neighbor_deviatio
51                     iterations_histor .append(
52                         y
53                         "iteration": iteration + 1,
54                         "eET": round(acceptance_probabilit , 5),
55                         y
56                     }
```

```

57     )
58     temperature *= cooling_rate
59
60     print(
61         f"Iteration {iteration + 1}, Temperature = {temperature:.2f}, "
62         f"Deviation = {current_deviatio }"
63         n
64     )
65     iteration += 1
66
67     if current_deviatio == 0:
68         message = "Reached global optimu
69     else:
70         m"
71         message = "Reached minimum temperatur
72         e"
73 # Update the grid to the best configuration foun
74 self.grid = current_grid
75
76     return (
77         copy.deepcopy(current_grid),
78         current_deviatio ,
79         iteration,
80         frequency,
81         message,
82         iterations_histoir ,
83     ) y

```

Pada implementasi algoritma *simulated annealing* diatas, diterapkan tahapan-tahapan berikut:

### 1. Inisialisasi

- Algoritma ini memulai dengan menyalin konfigurasi awal kubus (`current_grid`) dan menghitung deviasinya.
- Kemudian, suhu awal (`initial_temp`) dan kondisi suhu minimum (`min_temp`) ditetapkan. Iterasi dan variabel lain untuk pencatatan juga diinisialisasi.

### 2. Looping utama (pendinginan)

- Selama `temperature` masih lebih tinggi dari suhu minimum (`min_temp`), algoritma akan terus berjalan.
- Pada setiap iterasi, dihasilkan konfigurasi tetangga dengan melakukan pertukaran dua elemen secara acak dalam kubus.
- Setelah itu, algoritma mengevaluasi deviasi dari konfigurasi tetangga dan menghitung perbedaan deviasi (`delta`) antara konfigurasi saat ini dan tetangga tersebut.

### 3. Pengambilan keputusan

- Jika deviasi konfigurasi tetangga lebih kecil dari konfigurasi saat ini, maka konfigurasi tetangga diterima sebagai konfigurasi baru.
- Jika tidak, konfigurasi tetangga tetap mungkin diterima dengan probabilitas tetentu berdasarkan `acceptance_probability`, yang dihitung dengan formula probabilitas Boltzmann:  $\exp(-\Delta/\text{temperature})$ .

#### 4. Pendinginan suhu

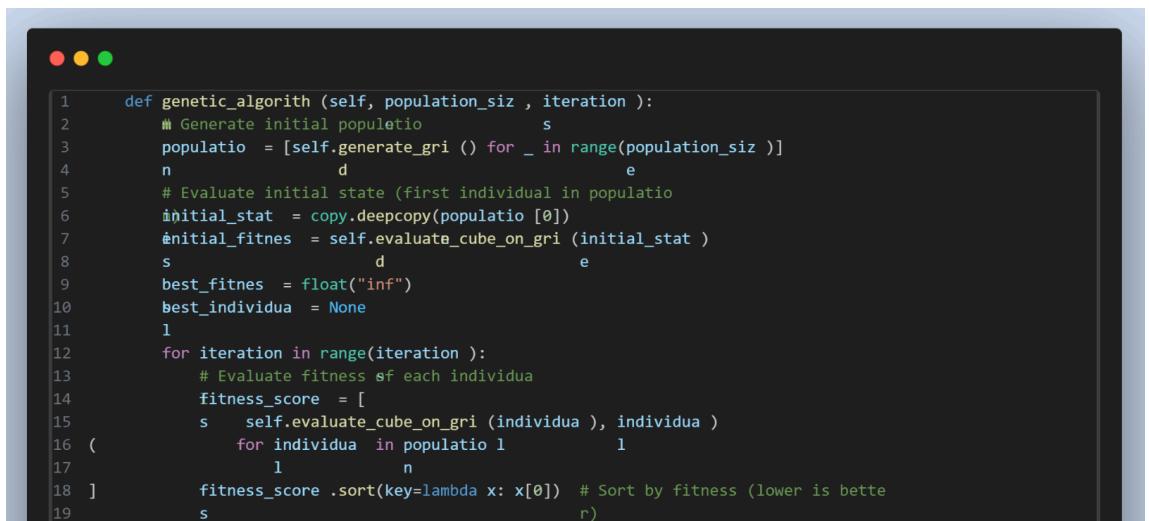
- Suhu dikurangi secara bertahap sesuai dengan cooling\_rate di setiap iterasi, sehingga algoritma lebih jarang menerima solusi suboptimal seiring penurunan suhu.

#### 5. Penghentian dan pengembalian hasil

- Jika deviasi mencapai nol, algoritma berhenti lebih awal dengan pesan bahwa solusi optimal global telah tercapai.
- Jika suhu mencapai nilai minimum sebelum solusi optimal tercapai, algoritma berhenti dengan menyatakan telah mencapai suhu minimum.
- Hasil akhir adalah konfigurasi terbaik yang ditemukan, deviasi akhir, jumlah iterasi yang dilakukan, frekuensi solusi yang tidak diterima, pesan status, dan riwayat iterasi dengan probabilitas penerimaan di setiap langkah.

#### 2.2.6 Genetic Algorithm

Algoritma *Genetic Algorithm* merupakan metode optimasi berbasis evolusi yang meniru proses seleksi alam. Algoritma ini bekerja dengan menghasilkan populasi solusi, memilih solusi terbaik, dan menggunakan proses crossover dan mutation untuk menghasilkan generasi baru yang lebih baik.



```
1  def genetic_algorithm (self, population_size , iteration ):
2      # Generate initial population
3      population = [self.generate_gri () for _ in range(population_size )]
4      n           d           e
5      # Evaluate initial state (first individual in population)
6      initial_state = copy.deepcopy(population [0])
7      initial_fitness = self.evaluate_cube_on_gri (initial_state )
8      s           d           e
9      best_fitness = float("inf")
10     best_individual = None
11     l
12     for iteration in range(iteration ):
13         # Evaluate fitness of each individual
14         fitness_scores = [
15             self.evaluate_cube_on_gri (individual ), individual
16         for individual in population ]
17         l           n
18     fitness_scores .sort(key=lambda x: x[0]) # Sort by fitness (lower is better)
19     s           r
```

```

20         # Keep track of the best solution
21         if fitness_score[0][0] < best_fitness :
22             best_fitness , best_individual = fitness_score[0]
23             s           l           s
24         # Selection: Take the top half of the population
25         selected_population = [
26             n   individual for _ , individual in fitness_score [: population_size // 2]
27             l           l           s           e
28         ]
29         # Crossover and mutation to produce the next generation
30         next_population = []
31         while len(next_population) < population_size :
32             parent1 = random.choice(selected_population)
33             parent2 = random.choice(selected_population)
34             child = self.crossover(parent1, parent2)
35             child = self.mutate(child)
36             next_population.append(child)
37             n
38         # Update population
39         population = next_population
40         n           n
41         # Final output
42         final_stats = best_individual
43         final_fitness = best_fitness
44         s           s
45         # Print initial and final states and fitness value
46         print("Initial State")
47         self.output_grid(initial_state)
48         print(f"Initial Fitness (Objective Function) , {initial_fitness}")
49         print(f"\nFinal State")
50         self.output_grid(final_state)
51         print(f"Final Fitness (Objective Function) , {final_fitness}")
52         n):"
53         # Return outputs as requested
54         return initial_stats , final_stats , final_fitness
e           e           s

```

Pada implementasi algoritma *genetic algorithm* diatas, diterapkan tahapan-tahapan berikut:

1. Inisialisasi populasi
  - Algoritma dimulai dengan menghasilkan populasi awal secara acak, di mana setiap individu dalam populasi merupakan konfigurasi kubus (atau grid).
  - Algoritma juga menghitung fitness dari individu pertama dalam populasi sebagai referensi awal.
2. Evaluasi fitness
  - Pada setiap iterasi, fitness (atau deviasi) dari setiap individu dihitung dengan fungsi evaluate\_cube\_on\_grid.
  - Populasi kemudian diurutkan berdasarkan nilai fitness, dengan nilai lebih rendah dianggap lebih baik.
  - Jika individu terbaik dalam populasi memiliki fitness lebih baik dari solusi terbaik yang ditemukan sejauh ini, maka individu dan fitness tersebut disimpan sebagai solusi terbaik saat ini.

### 3. Seleksi

- Setelah populasi diurutkan berdasarkan fitness, algoritma memilih setengah populasi terbaik sebagai `selected_population`. Individu-individu ini akan menjadi *parent* yang digunakan untuk menghasilkan generasi berikutnya.

### 4. Crossover dan mutasi

- Algoritma kemudian melakukan crossover (kombinasi dua individu) dan mutasi (modifikasi acak) untuk menghasilkan individu baru.
- Dua individu dari populasi terpilih dipilih secara acak sebagai pasangan untuk menghasilkan keturunan (child), dengan proses crossover dilakukan melalui fungsi crossover.
- Setelah itu, individu keturunan mengalami mutasi melalui fungsi `mutate`, yang bertujuan untuk memperkenalkan keragaman dan membantu menghindari solusi lokal.
- Individu baru ini ditambahkan ke dalam `next_population`, dan proses ini berlanjut hingga jumlah individu pada populasi baru sama dengan ukuran populasi awal.

### 5. Pembaruan populasi

- Populasi lama digantikan oleh populasi baru yang dihasilkan melalui crossover dan mutation. Proses ini diulang selama jumlah iterasi yang ditentukan.

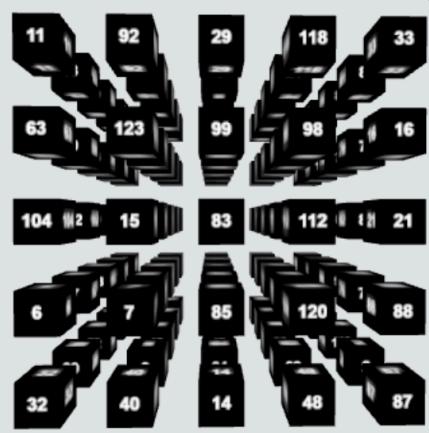
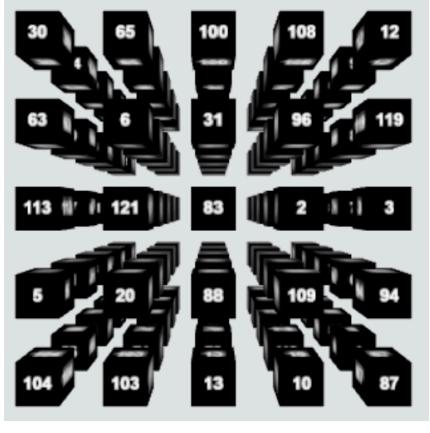
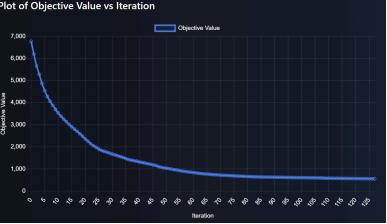
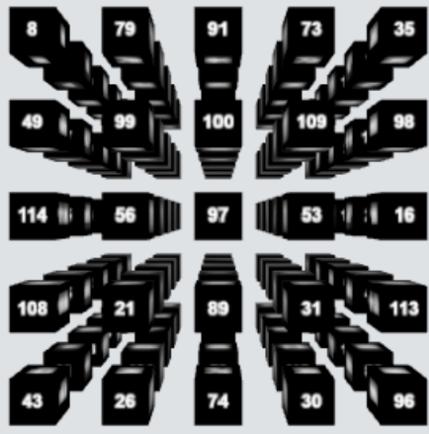
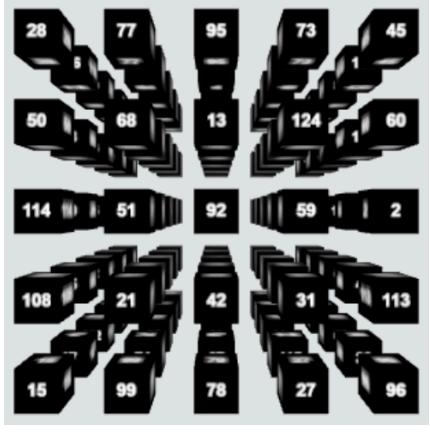
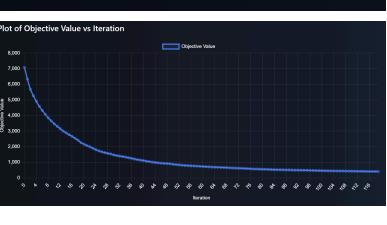
### 6. Output akhir

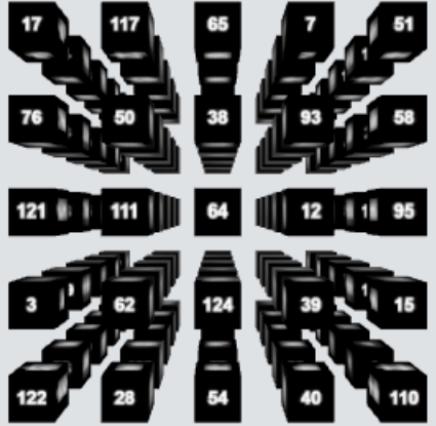
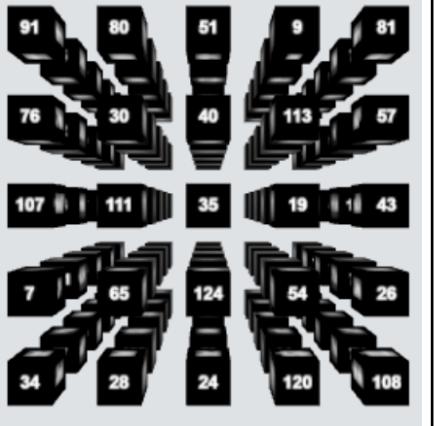
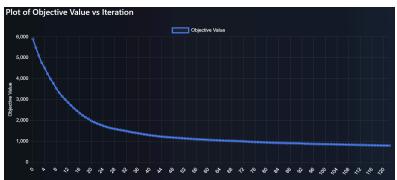
- Setelah iterasi selesai, algoritma mengeluarkan hasil berupa solusi awal, solusi terbaik yang ditemukan, dan nilai fitness dari solusi terbaik.
- Algoritma juga menampilkan grid awal, grid akhir, dan nilai fitness masing-masing.

## 2.3 Hasil Eksperimen dan Analisis

### 2.3.1 Steepest Ascent Hill-climbing

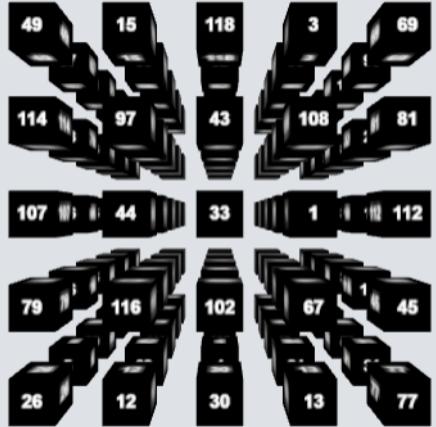
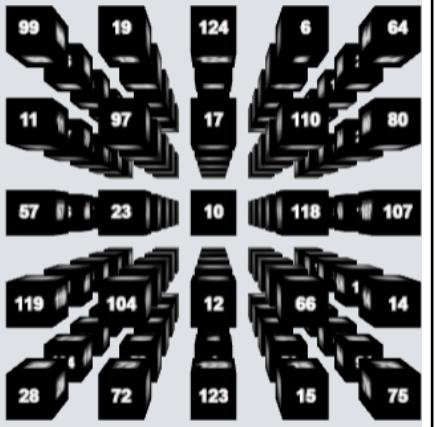
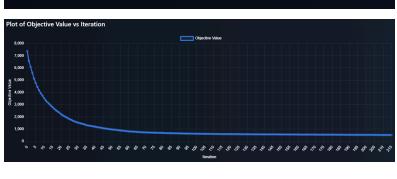
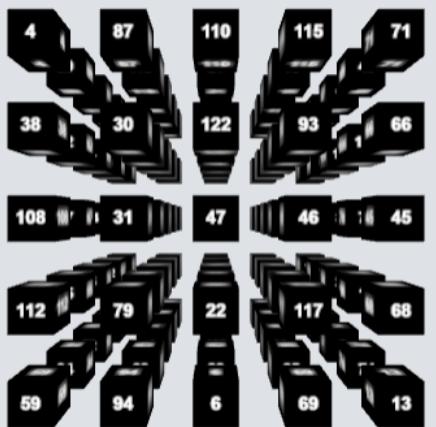
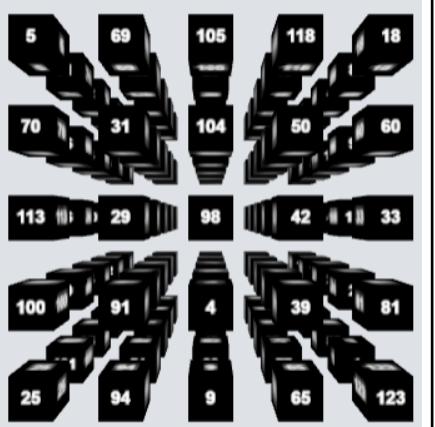
#### 2.3.1.1 Hasil Eksperimen

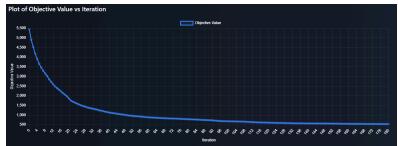
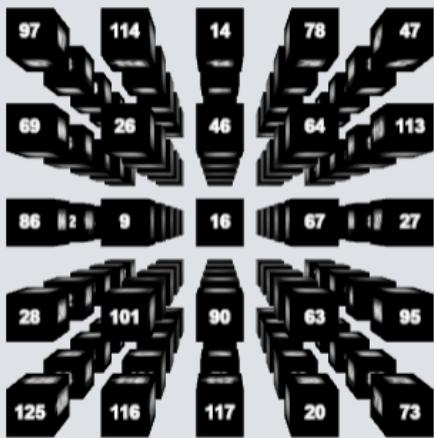
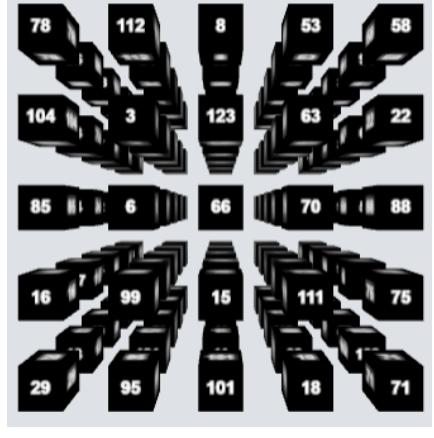
No	State Awal	State Akhir	Data
1			<p>Initial State Objective Function: 6773</p> <p>Computation Time: 88.54s</p> <p>Final State Objective Function: 558</p> <p>Number of Iterations: 128</p> <p>Plot of Objective Value vs Iteration</p> 
2			<p>Initial State Objective Function: 7088</p> <p>Computation Time: 84.06s</p> <p>Final State Objective Function: 410</p> <p>Number of Iterations: 120</p> <p>Plot of Objective Value vs Iteration</p> 

3	 	<p>Initial State Objective Function: 5904</p> <p>Computation Time: 86.03s</p> <p>Final State Objective Function: 793</p> <p>Number of Iterations: 123</p> 
---	--	---

### 2.3.2 Sideways Move Hill-climbing

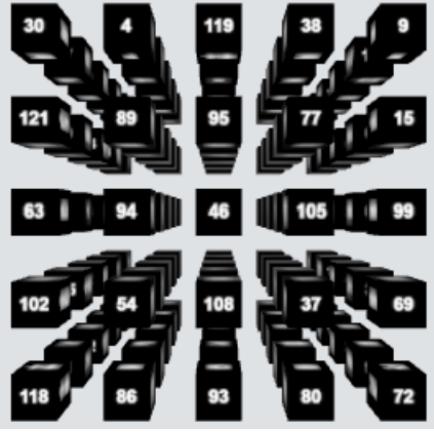
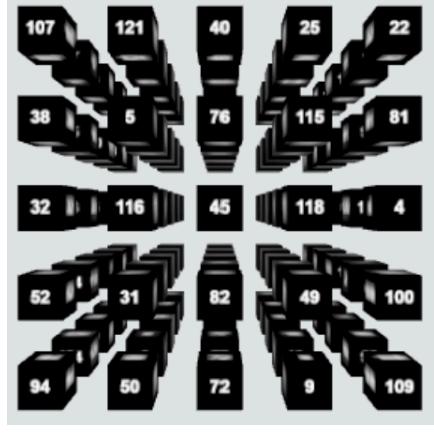
#### 2.3.2.1 Hasil Eksperimen

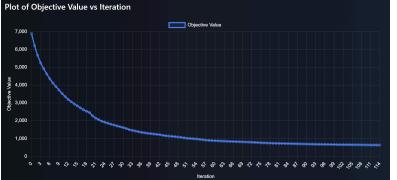
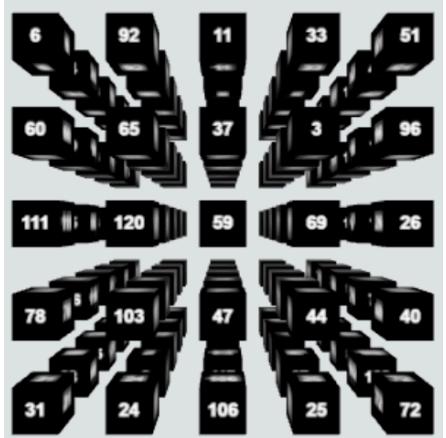
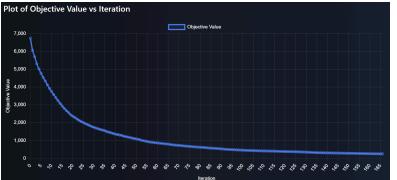
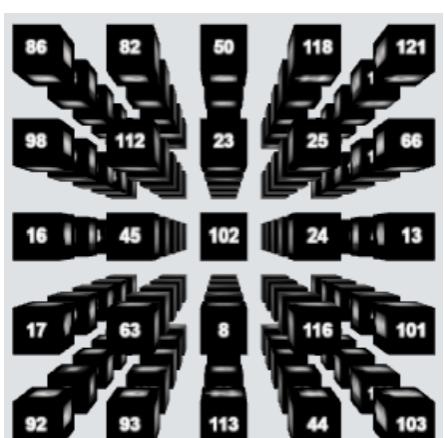
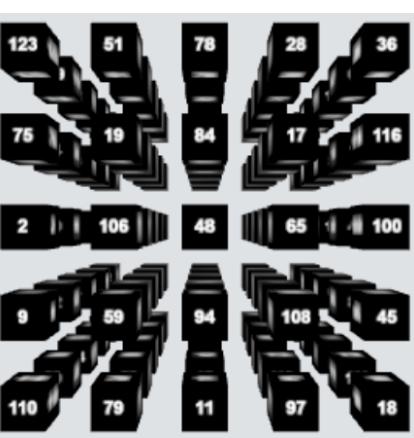
No	State Awal	State Akhir	Data
1			<p>Initial State Objective Function: 7388</p> <p>Computation Time: 127.82s</p> <p>Final State Objective Function: 515</p> <p>Number of Iterations: 216</p> 
2			<p>Initial State Objective Function: 5449</p> <p>Computation Time: 105.49s</p> <p>Final State Objective Function: 531</p> <p>Number of Iterations: 181</p>

		
3	 	<p>Initial State Objective Function: 6527</p> <p>Computation Time: 113.14s</p> <p>Final State Objective Function: 522</p> <p>Number of Iterations: 194</p> 

### 2.3.3 Random Restart Hill-climbing

#### 2.3.3.1 Hasil Eksperimen

No	State Awal	State Akhir	Data
1			<p>Initial State Objective Function: 6891</p> <p>Computation Time: 166.61s</p> <p>Number of Restarts: 2</p> <p>Iterations in Restart 2: 128</p> <p>Final State Objective Function: 636</p> <p>Number of Iterations: 115</p> <p>Iterations in Restart 1: 115</p>

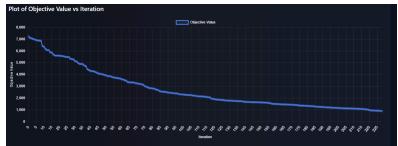
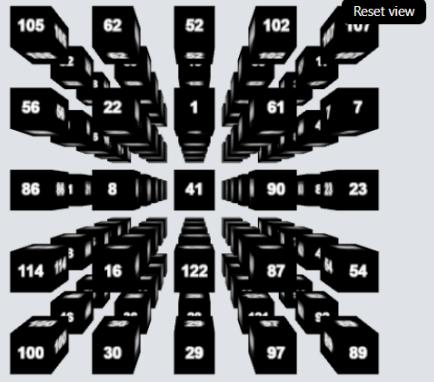
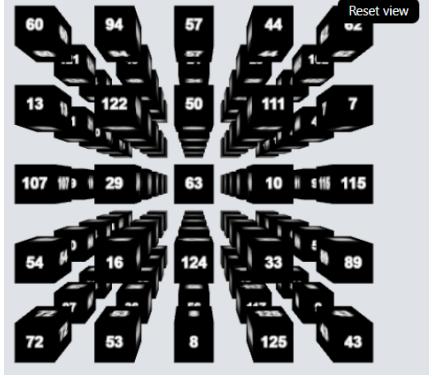
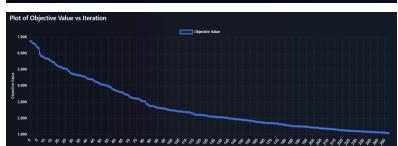
		
2	 	<p>Initial State Objective Function: 6743</p> <p>Computation Time: 189.14s</p> <p>Number of Restarts: 2</p> <p>Iterations in Restart 2: 167</p> <p>Final State Objective Function: 253</p> <p>Number of Iterations: 167</p> <p>Iterations in Restart 1: 105</p> 
3	 	<p>Initial State Objective Function: 7148</p> <p>Computation Time: 142.95s</p> <p>Number of Restarts: 2</p> <p>Iterations in Restart 2: 106</p>

		<p>Final State Objective Function: 654</p> <p>Number of Iterations: 101</p> <p>Iterations in Restart 1: 101</p>
--	--	---

### 2.3.4 Stochastic

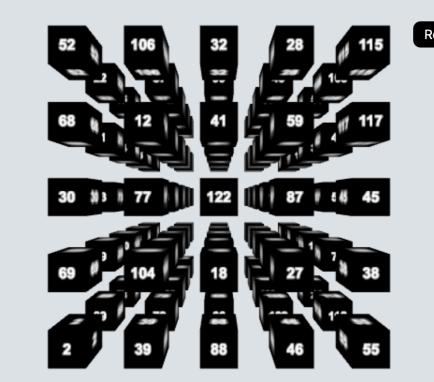
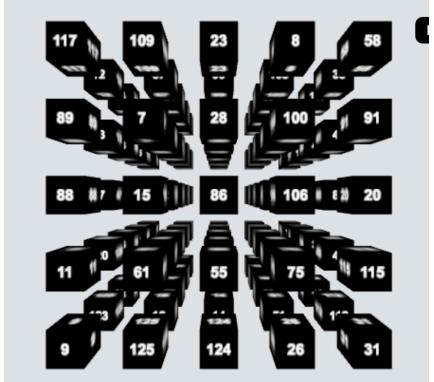
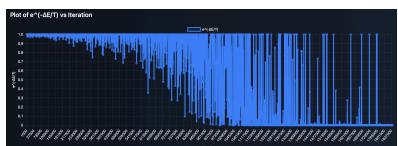
#### 2.3.4.1 Hasil Eksperimen

No	State Awal	State Akhir	Data
1			<p>Initial State Objective Function: 6909</p> <p>Computation Time: 0.70s</p> <p>Final State Objective Function: 1689</p> <p>Number of Iterations: 187</p>
2			<p>Initial State Objective Function: 7235</p> <p>Computation Time: 1.28s</p> <p>Final State Objective Function: 906</p> <p>Number of Iterations: 229</p>

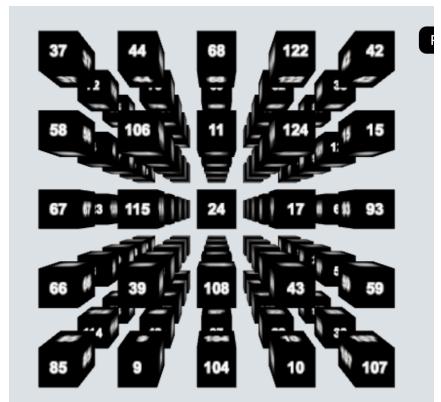
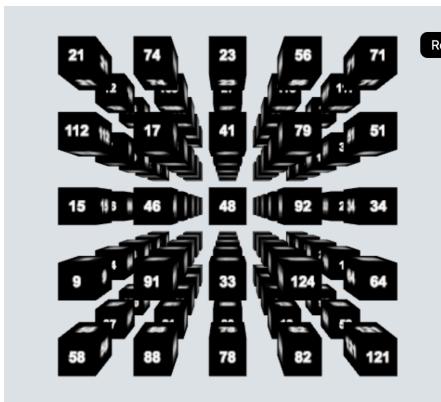
		
3	 	<p>Initial State Objective Function: 6749</p> <p>Computation Time: 1.56s</p> <p>Final State Objective Function: 1084</p> <p>Number of Iterations: 254</p> 

### 2.3.5 Simulated Annealing

#### 2.3.5.1 Hasil Eksperimen

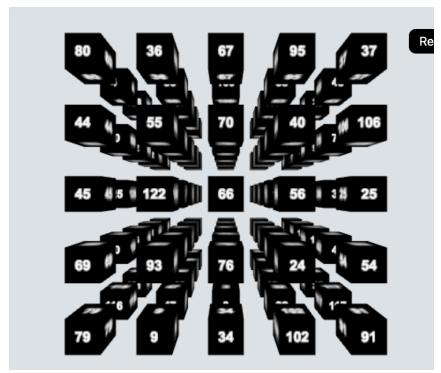
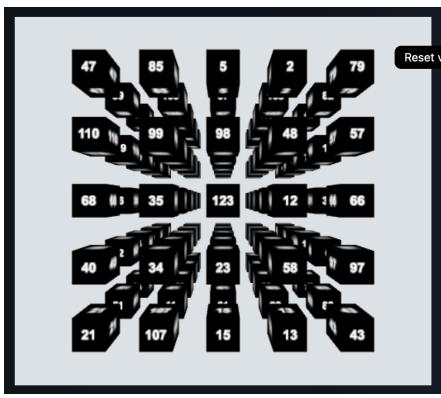
No	State Awal	State Akhir	Data
1			<p>Initial State Objective Function: 6111</p> <p>Computation Time: 123.73s</p> <p>Frequency: 1377316</p> <p>Final State Objective Function: 92</p> <p>Number of Iterations: 1842064</p> 

2



Initial State Objective Function: 6807  
Computation Time: 124.14s  
Frequency: 1376954  
Final State Objective Function: 103  
Number of Iterations: 1842064  
Plot of  $\epsilon^{-1}(\Delta f(t))$  vs Iteration

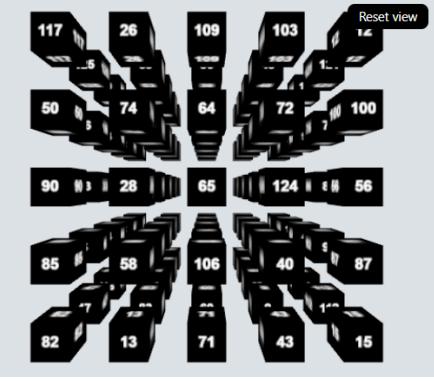
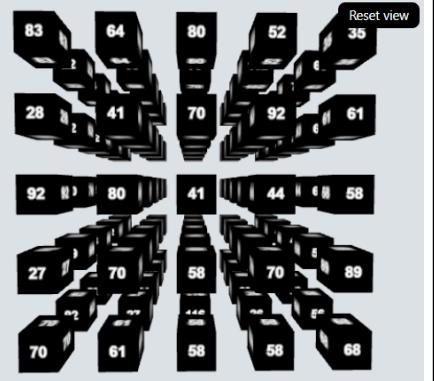
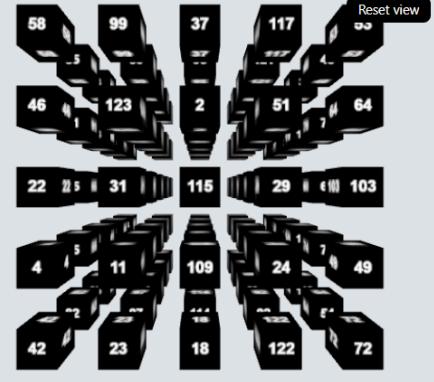
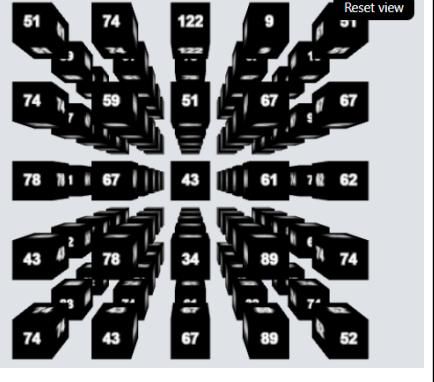
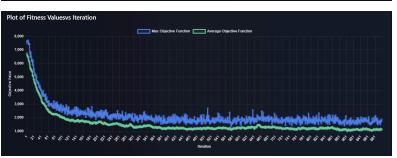
3



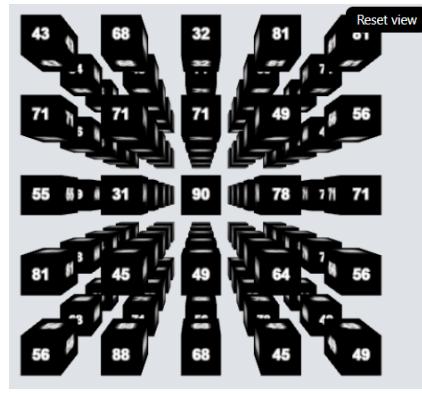
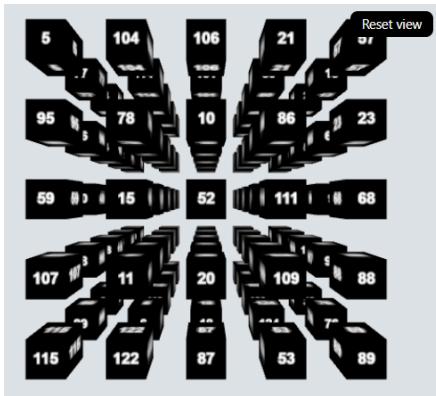
Initial State Objective Function: 7081  
Computation Time: 123.83s  
Frequency: 1376591  
Final State Objective Function: 81  
Number of Iterations: 1842064  
Plot of  $\epsilon^{-1}(\Delta f(t))$  vs Iteration

## 2.3.6 Genetic Algorithm

### 2.3.6.1 Hasil Eksperimen

No	State Awal	State Akhir	Data
1			<p>Initial State Objective Function: 6643</p> <p>Computation Time: 16.10s</p> <p>Population Size: 50</p> <p>Final State Objective Function: 706</p> <p>Number of Iterations: 1000</p> 
2			<p>Initial State Objective Function: 6626</p> <p>Computation Time: 16.53s</p> <p>Population Size: 50</p> <p>Final State Objective Function: 731</p> <p>Number of Iterations: 1000</p> 

3



Initial State Objective Function: 6551
Computation Time: 16.04s
Population Size: 50
Final State Objective Function: 669
Number of Iterations: 1000

Plot of Fitness Value vs Iteration

## Analisis

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

HC Steepest Ascent lumayan mendekati global optima karena algoritma tersebut hanya memilih yang terbaik diantara tetangganya yang sangat memungkinkan untuk terjebak di local optima.
  - b. HC Sideways Move
 

HC Sideways Move lebih mendekati global optima dibandingkan dengan Steepest Ascent karena dapat mengatasi masalah saat bertemu dengan *flat*, sehingga bisa lebih mendekati global optima.
  - c. HC Random Restart
 

HC Random Restart memiliki kesempatan yang lebih baik untuk mendekati global optima, karena algoritma ini memanfaatkan peluang yang lebih besar kalau frekuensi restart-nya lebih banyak.
  - d. HC Stochastic
 

HC Stochastic tidak mendekati global optima, hal tersebut bisa dilihat dari nilai akhir objective functionnya yang lebih besar daripada nilai akhir objective function algoritma lain. Algoritma ini mudah terjebak di *local optimum*, terutama tanpa adanya mekanisme restart atau mutasi yang bisa membantu keluar dari kondisi tersebut.
  - e. Genetic Algorithm
 

Genetic Algorithm menunjukkan hasil yang lebih mendekati global optima dibandingkan Stochastic Hill Climbing, terlihat dari nilai akhir *objective function*

yang lebih rendah. Hal ini disebabkan oleh kemampuan algoritma untuk mengeksplorasi solusi melalui proses seleksi, mutasi, dan crossover yang memungkinkan eksplorasi ruang solusi yang lebih luas.

f. Simulated Annealing

Simulated Annealing mempunyai kemungkinan untuk paling dekat dengan global optima, memanfaatkan perhitungan untuk memilih yang lebih ‘buruk’, algoritma ini sangat bagus untuk mendekati global optima. Semakin lambat temperaturnya berkurang, semakin bagus untuk mendekati global optima.

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

a. HC Steepest Ascent

Hasil Steepest kurang bagus dibandingkan algoritma lainnya karena masih lebih rentan untuk terjebak di optimum lokal. HC Steepest Ascent masih memiliki keterbatasan ketika bertemu dengan kondisi *flat*.

b. HC Sideways Move

Hasil Sideways Move lebih bagus dibandingkan HC Steepest Ascent karena tidak memiliki batasan ketika bertemu dengan kondisi *flat*.

c. HC Random Restart

Hasil HC Random Restart sedikit lebih bagus dibandingkan dengan algoritma HC Steepest Ascent dan juga HC Sideways Move karena mengatasi keterbatasan dari kondisi *flat* dan ketergantungan pada initial state dengan membangkitkan initial state baru ketika terjebak pada optimum lokal.

d. HC Stochastic

Dibandingkan dengan Genetic Algorithm, Stochastic Hill Climbing cenderung lebih cepat mencapai solusi tetapi kurang mampu mengeksplorasi ruang pencarian secara menyeluruh. Genetic Algorithm lebih unggul dalam menemukan solusi yang lebih dekat dengan global optima karena adanya variasi genetik dalam populasi, meskipun memerlukan waktu lebih lama.

e. Genetic Algorithm

Hasil Genetic Algorithm kurang bagus dibandingkan algoritma lainnya, namun dibandingkan dengan Stochastic Hill Climbing, Genetic Algorithm lebih unggul dalam kualitas hasil akhir karena mampu menemukan solusi yang lebih optimal melalui kombinasi solusi dalam populasi. Genetic Algorithm tidak mudah terjebak dalam *local optimum* karena adanya variasi genetik yang memperluas ruang pencarian.

f. Simulated Annealing

Dibandingkan dengan semua algoritma lainnya, algoritma ini memiliki nilai objektif akhir yang paling mendekati global optima tetapi harus tergantung denganberapa lambatnya temperatur berkurang.

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

a. HC Steepest Ascent

Durasi proses pada algoritma Hill Climbing (HC) Steepest Ascent lebih cepat dibandingkan HC Sideways Move karena Steepest Ascent selalu memilih tetangga dengan nilai evaluasi terbaik, sehingga mengurangi waktu pencarian solusi. Namun, jika dibandingkan dengan Simulated Annealing, algoritma ini sedikit lebih lama karena Simulated Annealing menggunakan pendekatan probabilistik yang memungkinkan eksplorasi lebih luas dan menghindari lokal optimum, meskipun memerlukan lebih banyak iterasi.

b. HC Sideways Move

Durasi proses pada algoritma Hill Climbing (HC) Sideways Move lebih cepat dibandingkan HC Random Restart karena Sideways Move tidak memerlukan inisialisasi ulang grid, sehingga menghemat waktu. Namun, jika dibandingkan dengan HC Steepest Ascent, algoritma ini sedikit lebih lama karena Sideways Move tidak selalu memilih tetangga terbaik, sehingga mungkin memerlukan lebih banyak iterasi untuk menemukan solusi yang optimal.

c. HC Random Restart

Random Restart memiliki durasi yang paling lama dibandingkan algoritma lainnya. Hal ini dikarenakan random restart melakukan sejumlah HC Steepest yang pada sendirinya sudah cukup lama.

d. HC Stochastic

Durasi proses pada Stochastic Hill Climbing relatif lebih cepat dibandingkan Genetic Algorithm. Hal ini karena Stochastic Hill Climbing tidak perlu memelihara populasi, melainkan hanya memodifikasi solusi tunggal. Namun, kecepatan ini terkadang mengorbankan kualitas hasil akhir yang mana tidak seefisien solusi dari Genetic Algorithm.

e. Genetic Algorithm

Durasi proses Genetic Algorithm lebih lama dibandingkan Stochastic Hill Climbing karena harus memproses populasi, melakukan seleksi, crossover, dan mutasi di setiap iterasi. Meski membutuhkan waktu yang lebih panjang, Genetic Algorithm cenderung menghasilkan solusi yang lebih baik dibanding Stochastic Hill Climbing.

f. Simulated Annealing

Durasi proses algoritma Simulated Annealing tergantung pada cooling rate nya. Pada kasus ini karena cooling rate nya sangat kecil, kecepatannya cukup lama dibandingkan algoritma lainnya tetapi masih lebih cepat dibandingkan Random Restart. Hal tersebut karena jumlah dari iterasi yang sangat banyak dibandingkan algoritma lainnya bisa mencapai jutaan sedangkan algoritma lainnya hanya mencapai kisaran ratusan hingga seribu iterasi.

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

a. HC Steepest Ascent

Hasil dari Steepest Ascent tidak terlalu konsisten karena state awal sangat mempengaruhi keberjalanannya pencarian dan adanya keterbatasan saat menemui *flat*.

b. HC Sideways Move

Hasil dari HC Sideways Move juga tidak terlalu konsisten tetapi masih lebih konsisten dibandingkan HC Steepest, hal ini karena HC Sideways tidak ada keterbatasan dalam melewati *flat*.

c. HC Random Restart

Hasil dari HC Random Restart mirip seperti HC Steepest Ascent yang masih kurang konsisten. Hasil sangat bergantung pada initial state yang dibangkitkan walaupun probabilitas mendapatkan initial state yang bagus lebih tinggi karena initial state dibangkitkan pada setiap restart.

d. HC Stochastic

Hasil Stochastic Hill Climbing cenderung cukup konsisten, terutama karena algoritma ini tidak bergantung pada variabilitas populasi. Namun, konsistensi ini sering kali tetap berada dalam *local optimum* tertentu yang mungkin berbeda dengan *global optimum*.

e. Genetic Algorithm

Hasil dari Genetic Algorithm cukup konsisten jika parameter yang sama digunakan, meskipun variasi hasil masih dapat terjadi tergantung pada variasi awal populasi dan operasi genetik. Namun, nilai akhir *objective function* dari eksperimen yang sama cenderung berada dalam rentang yang mirip.

f. Simulated Annealing

Hasil Simulated Annealing merupakan algoritma yang paling konsisten karena *cooling\_rate* (temperatur) nya sama. Dengan mengurangi suhu secara bertahap, algoritma dapat menghindari terjebak dalam optimum lokal, memberikan kesempatan untuk menemukan solusi yang lebih baik seiring waktu.

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

Semakin banyak iterasi dan ukuran populasi, semakin baik kualitas solusi yang dapat dicapai oleh Genetic Algorithm, karena ruang pencarian yang lebih besar dan kesempatan lebih banyak untuk perbaikan melalui crossover dan mutasi. Meningkatkan jumlah iterasi memberikan Genetic Algorithm waktu yang lebih lama untuk mendekati solusi optimal, dan ukuran populasi yang lebih besar memberikan keragaman genetik yang lebih besar untuk dieksplorasi.

## Bab 3

### Kesimpulan

#### 3.1 Kesimpulan

Analisis berbagai algoritma pencarian menunjukkan bahwa setiap metode memiliki kelebihan dan kekurangan dalam mendekati solusi optimal. HC Steepest Ascent cenderung lebih cepat tetapi rentan terhadap local optimum, sementara HC Sideways Move menawarkan hasil yang lebih baik dengan kemampuan untuk mengatasi kondisi flat. HC Random Restart meningkatkan peluang menemukan global optima meskipun dengan durasi yang lebih lama, sedangkan HC Stochastic meskipun lebih cepat, sering terjebak di local optimum. Genetic Algorithm menunjukkan kinerja yang kuat berkat mekanisme seleksi dan mutasi yang mendalam, dan Simulated Annealing memiliki potensi terbaik untuk mencapai solusi optimal melalui eksplorasi yang lebih luas.

Konsistensi hasil juga bervariasi antar algoritma; Simulated Annealing terbukti paling konsisten karena pengaturan temperatur yang stabil, sedangkan HC Steepest Ascent dan HC Random Restart lebih dipengaruhi oleh kondisi awal. Genetic Algorithm memberikan hasil yang baik dalam hal konsistensi jika parameter yang sama digunakan, meskipun tetap ada variasi. Dengan semakin banyak iterasi dan ukuran populasi yang lebih besar, kualitas solusi yang dicapai oleh Genetic Algorithm cenderung meningkat, menunjukkan bahwa eksplorasi yang lebih luas dan perbaikan berkelanjutan sangat penting dalam pencarian solusi optimal.

Alhasil algoritma terbaik untuk kasus 5x5x5 diagonal magic cube ini adalah Algoritma Simulated Annealing karena konsistensinya dalam mendapatkan nilai objektif yang paling mendekati global optimum dan durasinya yang masih cukup baik. Dan algoritma terburuk untuk kasus ini adalah Stochastic yang walaupun durasi eksekusinya sangat cepat, hasilnya lebih buruk dan tidak konsisten dibandingkan dengan algoritma lainnya.

### **3.2 Saran**

Berdasarkan analisis algoritma pencarian untuk kasus 5x5x5 diagonal magic cube, disarankan untuk menggunakan Algoritma Simulated Annealing sebagai metode utama karena konsistensinya dalam mendekati solusi optimal dan durasinya yang relatif efisien. Selain itu, eksplorasi lebih lanjut dengan parameter yang tepat dapat meningkatkan hasil.

Sebaiknya hindari penggunaan algoritma Stochastic dalam konteks ini, mengingat hasilnya yang tidak konsisten meskipun memiliki waktu eksekusi yang cepat. Untuk algoritma lainnya, seperti HC Steepest Ascent dan HC Random Restart, perlu diperhatikan pengaturan kondisi awal untuk mengoptimalkan performa. Penelitian lebih lanjut dapat dilakukan untuk mengeksplorasi kombinasi algoritma atau penyesuaian parameter guna meningkatkan kualitas solusi di masa depan.

Penggunaan bahasa pemrograman lain yang lebih cepat dibandingkan dengan python juga dapat dipertimbangkan untuk mengoptimalkan durasi dari pencarian masing-masing algoritma dan memaksimalkan hasil yang didapatkan.

## **Lampiran**

Pembagian Tugas :

NIM	Tugas
13522125	Simulated Annealing, Steepest Ascent Hill-climbing Algorithm, API, UI
13522128	Genetic Algorithm, Stochastic Algorithm, Laporan
13522148	Random Restart Hill-climbing, Cube Visualization, Laporan
13522162	Sideways Move Hill-climbing, Cube Visualization, Laporan

## **Referensi**

Edunex. (n.d.). *Berbagai algoritma local search*. Retrieved from  
<https://edunex.itb.ac.id/courses/64464/preview/278735>

Magisch Vierkant. (n.d.). *Features of magic cube*. Retrieved from  
<https://www.magischvierkant.com/three-dimensional-eng/magic-features/>

Trump, G. (n.d.). *Perfect magic cube*. Retrieved from  
<https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>

Wikipedia. (n.d.). *Magic cube*. Retrieved from  
[https://en.wikipedia.org/wiki/Magic\\_cube](https://en.wikipedia.org/wiki/Magic_cube)