

Tugas Besar 1
IF3170 Intelelegensi Artifisial

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



Disusun oleh :

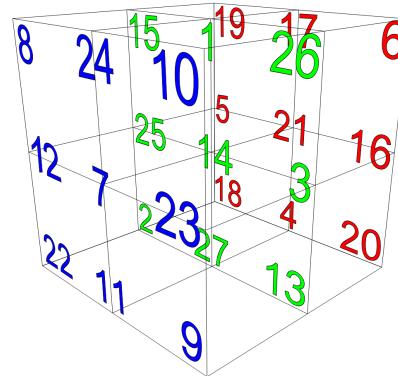
Mohammad Nugraha Eka Prawira	13522001/K01
Ahmad Farid Mudrika	13522008/K01
Muhammad Yusuf Rafi	13522009/K01
Muhammad Rifki Virzadeili	13522120/K02

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

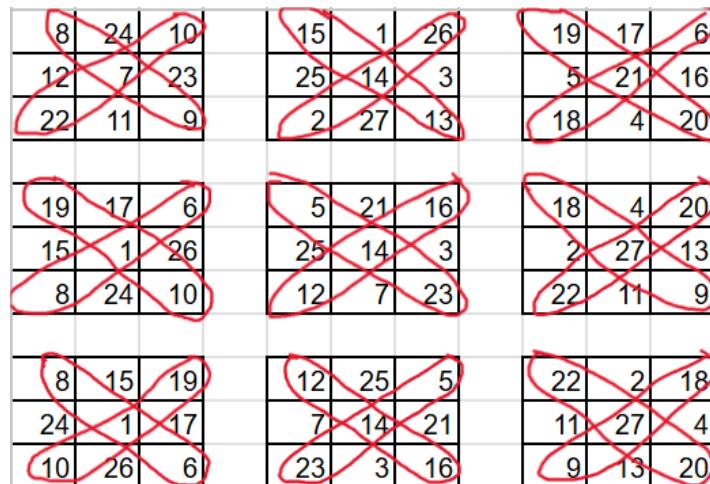
I. Deskripsi Persoalan

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).

II. Pembahasan

Magic Number dari Magic Cube dengan ukuran 5x5x5 dapat ditentukan dengan rumus $S = \frac{m(m^3+1)}{2}$, dengan m sebagai panjang cube. Terdapat $3m^2+6m+4$ garis lurus, m^2 baris, m^2 kolom, m^2 tiang, 6m diagonal dan 4 triagonal. Dengan persoalan ini, dimana $m = 5$, maka didapatkan magic number bernilai 315, dengan
Magic cube memiliki syarat seperti yang telah disebutkan sebelumnya:

1. Jumlah setiap baris selalu memiliki nilai yang sama dengan S
Terdapat 5^2 baris = 25 baris
 2. Jumlah setiap kolom selalu memiliki nilai yang sama dengan S
Terdapat 5^2 kolom = 25 kolom
 3. Jumlah setiap tiang selalu memiliki nilai yang sama dengan S
Terdapat 5^2 tiang = 25 tiang
 4. Jumlah setiap diagonal ruang memiliki nilai yang sama dengan S
Terdapat 4 diagonal ruang
 5. Jumlah setiap diagonal pada setiap bidang memiliki nilai yang sama dengan S
Terdapat $6*5$ diagonal yang terbentuk = 30 diagonal bidang
- Terdapat $25+25+25+4+30 = 109$ garis yang terbentuk dari magic cube berukuran 5x5.

Objective Function

Objective function yang kami usulkan adalah total deviasi absolut dari magic number untuk semua properti yang harus dipenuhi:

$$f(x) = \sum |S_i - M|$$

Dimana x adalah layout dari cube saat ini, S_i adalah jumlah dari properti ke-i (baris, kolom, tiang, diagonal), dan M adalah magic number.

Alasan kami memilih ini sebagai objective function kami adalah

- a. Mencakup semua properti: function ini memperhitungkan semua aspek yang harus dipenuhi seperti baris, kolom, tiang, diagonal ruang, dan diagonal potongan bidang.
- b. Normalisasi: penggunaan nilai deviasi yang absolut memastikan bahwa setiap properti diperlakukan sama pentingnya, terlepas dari lebih besar atau lebih kecil nilainya dari magic number, ini juga menandakan bahwa nilai pembanding yang kita lihat hanyalah nilai jarak deviasi ke nilai magic number
- c. Minimization problem: function ini mengubah masalah menjadi minimization problem, dimana solusi optimal akan memiliki nilai 0 (semua properti bernilai sama dengan magic number)

Penjelasan Setiap Algoritma Local Search

Proses pencarian solusi untuk permasalahan magic cube 5x5 ini dapat dilakukan dengan memecah permasalahan menjadi lebih mudah terlebih dahulu. Anggaplah kita memiliki list of number sebanyak 5^3 yang merepresentasikan posisi angka pada magic number. List of number (L) tersebut akan menjadi state selama pencarian dilakukan. Misalkan kita mempunyai cube seperti berikut

25	16	80	104	90	90
115	98	4	1	97	97
42	111	85	2	75	97
66	72	27	102	48	75
67	18	119	106	5	48
67	18	119	106	5	48
116	17	14	73	95	14
40	50	81	65	79	37
56	120	55	49	35	94
36	110	46	22	101	60
36	110	46	22	101	101

Kemudian, akan dipecah menjadi beberapa level untuk mempermudah penyimpanan angka.

121	103	7	20	59	59
29	28	122	125	11	99
51	15	41	124	84	11
78	54	99	24	60	54
36	110	46	22	101	60
36	110	46	22	101	101

Level 1

31	53	112	109	10	10
12	82	34	87	100	100
103	3	105	8	96	100
113	57	9	62	74	96
56	120	55	49	35	74
56	120	55	49	35	35

Level 2

47	61	45	76	86	86
107	43	38	33	94	94
89	68	63	58	37	94
32	93	88	83	19	37
40	50	81	65	79	19
40	50	81	65	79	79

Level 3

91	77	71	6	70	70
52	64	117	69	13	13
30	118	21	123	23	23
26	39	92	44	114	114
116	17	14	73	95	95
116	17	14	73	95	95

Level 4

25	16	80	104	90	90
115	98	4	1	97	97
42	111	85	2	75	97
66	72	27	102	48	75
67	18	119	106	5	48
67	18	119	106	5	48
116	17	14	73	95	95
40	50	81	65	79	79
40	50	81	65	79	79

Level 5

Dalam proses pencarian menggunakan local search, kita hanya bisa menukar 2 posisi elemen L untuk membuat successor. Posisi dari elemen L dapat direpresentasikan menjadi indeks dari L.

Kami membuat sebuah kelas yang berisi fungsi-fungsi cube yang dibutuhkan pada berbagai algoritma:

```
● ● ●
1 import random
2 import numpy as np
3 import time
4 from numba import jit
5
6 @jit(nopython=True)
7 def calculate_deviation_numba(value, magic_number):
8     return abs(value - magic_number)
9
10 @jit(nopython=True)
11 def evaluate_cube(cube, n, magic_number):
12     total_deviation = 0
13
14
15     for i in range(n):
16         for j in range(n):
17
18             total_deviation += calculate_deviation_numba(np.sum(cube[i, j, :]), magic_number)
19
20             total_deviation += calculate_deviation_numba(np.sum(cube[i, :, j]), magic_number)
21
22             total_deviation += calculate_deviation_numba(np.sum(cube[:, i, j]), magic_number)
23
24     # Face diagonals (along each slice)
25     for k in range(n): # for each slice
26         # XY plane diagonals (for each Z slice)
27         diagonal_sum = 0
28         anti_diagonal_sum = 0
29         for i in range(n):
30             diagonal_sum += cube[i, i, k]
31             anti_diagonal_sum += cube[i, n-1-i, k]
32         total_deviation += calculate_deviation_numba(diagonal_sum, magic_number)
33         total_deviation += calculate_deviation_numba(anti_diagonal_sum, magic_number)
34
35     # XZ plane diagonals (for each Y slice)
36     diagonal_sum = 0
37     anti_diagonal_sum = 0
38     for i in range(n):
39         diagonal_sum += cube[i, k, i]
40         anti_diagonal_sum += cube[i, k, n-1-i]
41     total_deviation += calculate_deviation_numba(diagonal_sum, magic_number)
42     total_deviation += calculate_deviation_numba(anti_diagonal_sum, magic_number)
43
```

```

44     # YZ plane diagonals (for each X slice)
45     diagonal_sum = 0
46     anti_diagonal_sum = 0
47     for i in range(n):
48         diagonal_sum += cube[k, i, i]
49         anti_diagonal_sum += cube[k, i, n-1-i]
50     total_deviation += calculate_deviation_numba(diagonal_sum, magic_number)
51     total_deviation += calculate_deviation_numba(anti_diagonal_sum, magic_number)
52
53     # Diagonal Ruang
54     main_diagonal_sum = 0
55     anti_diagonal_sum_1 = 0
56     anti_diagonal_sum_2 = 0
57     anti_diagonal_sum_3 = 0
58     for i in range(n):
59         main_diagonal_sum += cube[i, i, i]
60         anti_diagonal_sum_1 += cube[i, n - 1 - i, i]
61         anti_diagonal_sum_2 += cube[n - 1 - i, i, i]
62         anti_diagonal_sum_3 += cube[i, i, n - 1 - i]
63
64     total_deviation += calculate_deviation_numba(main_diagonal_sum, magic_number)
65     total_deviation += calculate_deviation_numba(anti_diagonal_sum_1, magic_number)
66     total_deviation += calculate_deviation_numba(anti_diagonal_sum_3, magic_number)
67     total_deviation += calculate_deviation_numba(anti_diagonal_sum_2, magic_number)
68     return total_deviation
69
70 @jit(nopython=True)
71 def generate_neighbors_numba(flat_cube, n):
72     size = len(flat_cube)
73     num_neighbors = (size * (size - 1)) // 2
74     neighbor_cubes = np.zeros((num_neighbors, size), dtype=flat_cube.dtype)
75
76     idx = 0
77     for i in range(size):
78         for j in range(i + 1, size):
79             neighbor_cubes[idx] = flat_cube.copy()
80             neighbor_cubes[idx, i], neighbor_cubes[idx, j] = flat_cube[j], flat_cube[i]
81             idx += 1
82
83     return neighbor_cubes.reshape(-1, n, n, n)
84

```

```

85 class DiagonalMagicCube:
86     def __init__(self, n=5):
87         self.n = n
88         self.size = n**3
89         self.magic_number = self.calculate_magic_number()
90         self.cube = self.initialize_cube()
91
92     @classmethod
93     def constructor(cls, cube):
94         instance = cls()
95         instance.cube = cube
96         instance.n = cube.shape[0]
97         instance.size = cube.size
98         instance.magic_number = instance.calculate_magic_number()
99         return instance
100
101    def calculate_magic_number(self):
102        return ((self.size + 1) * self.n) // 2
103
104    def initialize_cube(self):
105        numbers = np.arange(1, self.size + 1, dtype=np.int32)
106        np.random.shuffle(numbers)
107        return numbers.reshape((self.n, self.n, self.n))
108
109    def evaluate(self):
110        return evaluate_cube(self.cube, self.n, self.magic_number)
111
112    def get_neighbors(self):
113        flat_cube = self.cube.flatten()
114        neighbor_cubes = generate_neighbors_numba(flat_cube, self.n)
115        return [DiagonalMagicCube.constructor(cube) for cube in neighbor_cubes]

```

Fungsi calculate magic number berguna untuk menghitung nilai magic number constant yang sudah ada pada rumus untuk digunakan pada objective function yang kita miliki. Lalu Initialize cube berguna untuk membuat array 3D yang berisi angka 1-125 yang tersusun secara acak. Kemudian evaluate adalah fungsi yang berguna untuk menghitung nilai score state cube berdasarkan objective function yang kita rencanakan, yaitu total nilai tiap komponen pembangun magic cube dikurangi dengan nilai magic constant nya. Semakin mendekati 0 nilainya berarti semakin baik nilai dari cube tersebut. Kemudian fungsi swap dan get random position berguna untuk mengambil koordinat secara random pada cube lalu swap bertugas untuk melakukan pertukaran antara 2 koordinat hasil dari get random position. Terakhir, get neighbor berguna untuk membangkitkan semua kemungkinan neighbor dari cube saat ini dan menghasilkan list neighbor cube dari cube saat ini untuk dibandingkan pada algoritma-algoritma local search yang membutuhkan.

Adapun beberapa algoritma yang akan dibahas untuk penyelesaian permasalahan magic number 5x5 adalah sebagai berikut:

a. Steepest Ascent Hill-climbing

function HILL-CLIMBING(*problem*) returns a state that is a local maximum

current \leftarrow MAKE-NODE(*problem.INITIAL-STATE*)

loop do

neighbor \leftarrow a highest-valued successor of *current*

if *neighbor.VALUE* \leq *current.VALUE* **then return** *current.STATE*

current \leftarrow *neighbor*

Starting from a randomly generated initial state

Loop that continually moves in the direction of increasing value (objective) or decreasing value (cost)

Terminates when it reaches a "peak" including "flat" where no neighbor has a higher value

1. Initial State
 - Menetapkan initial state dari magic cube secara random (inisialisasi kubus).
 - Menetapkan current value dari objective function pada kubus saat ini
2. Looping
 - Membangkitkan semua successor dari current state dengan melakukan swap antara setiap pasangan posisi secara sistematis.
 - Dari semua kemungkinan successor, dihitung nilai objective functionnya dan dipilih yang memiliki value terkecil (mendekati 0 atau goal state).
 - Penetapan pembaharuan current state dengan,
 - Jika value dari successor lebih kecil dari current value, maka program akan mengganti nilai current state menjadi successor tersebut, lalu looping dilanjutkan.
 - Jika nilainya lebih besar atau sama dari current value, maka current value langsung direturn dan looping diberhentikan.
 - Jika sudah mencapai global atau local optimum maka iterasi bisa diberhentikan (sesuai yang dijelaskan pada tahap pembaharuan current state di atas).

Eksperimen:



Tubes_AI - steepest_ascent-HC.py

```
1  class SteepestHillClimbing:
2      def __init__(self, cube):
3          self(cube = cube
4          self.list_result=[]
5
6      def run(self):
7          start_time = time.time()
8          current_score = self(cube).evaluate()
9          print(self(cube).cube)
10         i = 0
11         self.list_result.append((i, self(cube).cube, current_score))
12
13     while(True):
14         if current_score == 0:
15             break
16
17         best_neighbor = None
18         best_score = current_score
19         neighbors = self(cube).get_neighbors()
20         for neighbor in neighbors:
21             new_score = neighbor.evaluate()
22             if new_score <= best_score:
23                 best_neighbor = neighbor
24                 best_score = new_score
25         if best_neighbor is None:
26             break
27         if best_score >= current_score:
28             break
29         self(cube)= best_neighbor
30         current_score = best_score
31         i+=1
32         self.list_result.append((i, self(cube).cube, current_score))
33
34     end_time = time.time()
35     total_time = end_time - start_time
36     return self.list_result, total_time
37
```

Kami membuat class SteepestHillClimbing yang berguna untuk melakukan penyelesaian magic cube menggunakan algoritma Steepest Hill Climbing. Sesuai algoritma steepest program akan membangkitkan seluruh neighbor dan mencari state dengan score terendah dan membandingkannya dengan current_score dari state cube saat ini, jika nilainya lebih kecil maka iterasi akan dilanjutkan dan state akan berpindah ke state neighbor tersebut serta melanjutkan algoritma. Namun jika nilai terendah ternyata lebih besar atau sama dengan nilai score saat ini maka program akan berhenti dan

mengembalikan state saat ini. Setiap perubahan state akan dimasukan ke dalam list_result yang berisi iterasi keberapa, state cube pada iterasi tersebut, nilai score pada state tersebut.

Hasil Eksperimen :

Initial State

Magic Cube Visualization

Iteration: 0

Score: 6507

Layer 1

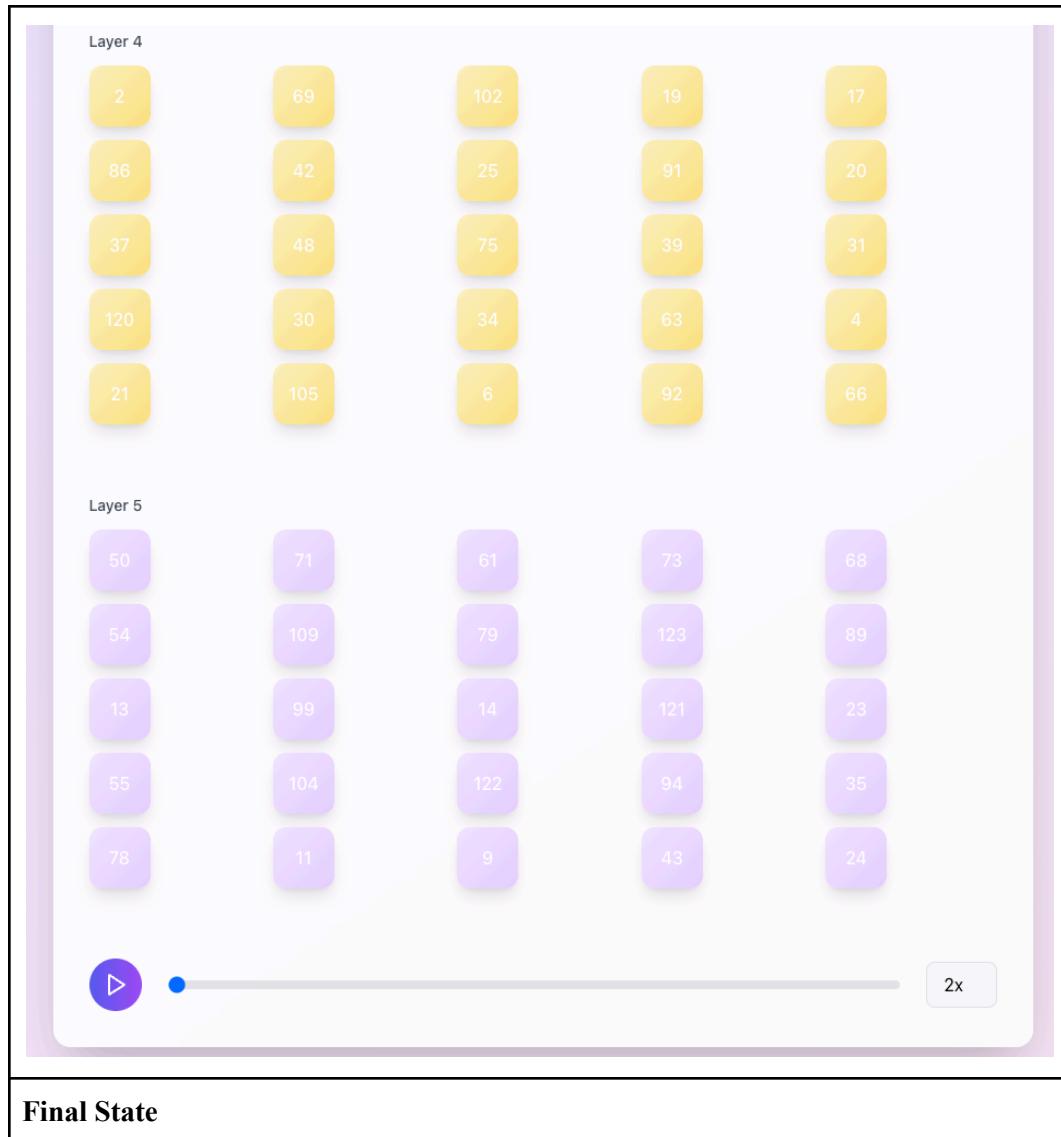
44	41	96	56	97
119	90	87	88	38
117	52	53	83	106
74	18	101	62	40
100	124	45	95	118

Layer 2

125	46	1	5	77
57	111	8	84	116
29	27	114	70	115
51	47	81	64	80
60	58	7	98	93

Layer 3

85	16	65	33	59
28	26	3	15	113
10	103	108	67	12
76	82	36	72	49
107	110	112	32	22



Magic Cube Visualization

Iteration: 105

Score: 292

Layer 1

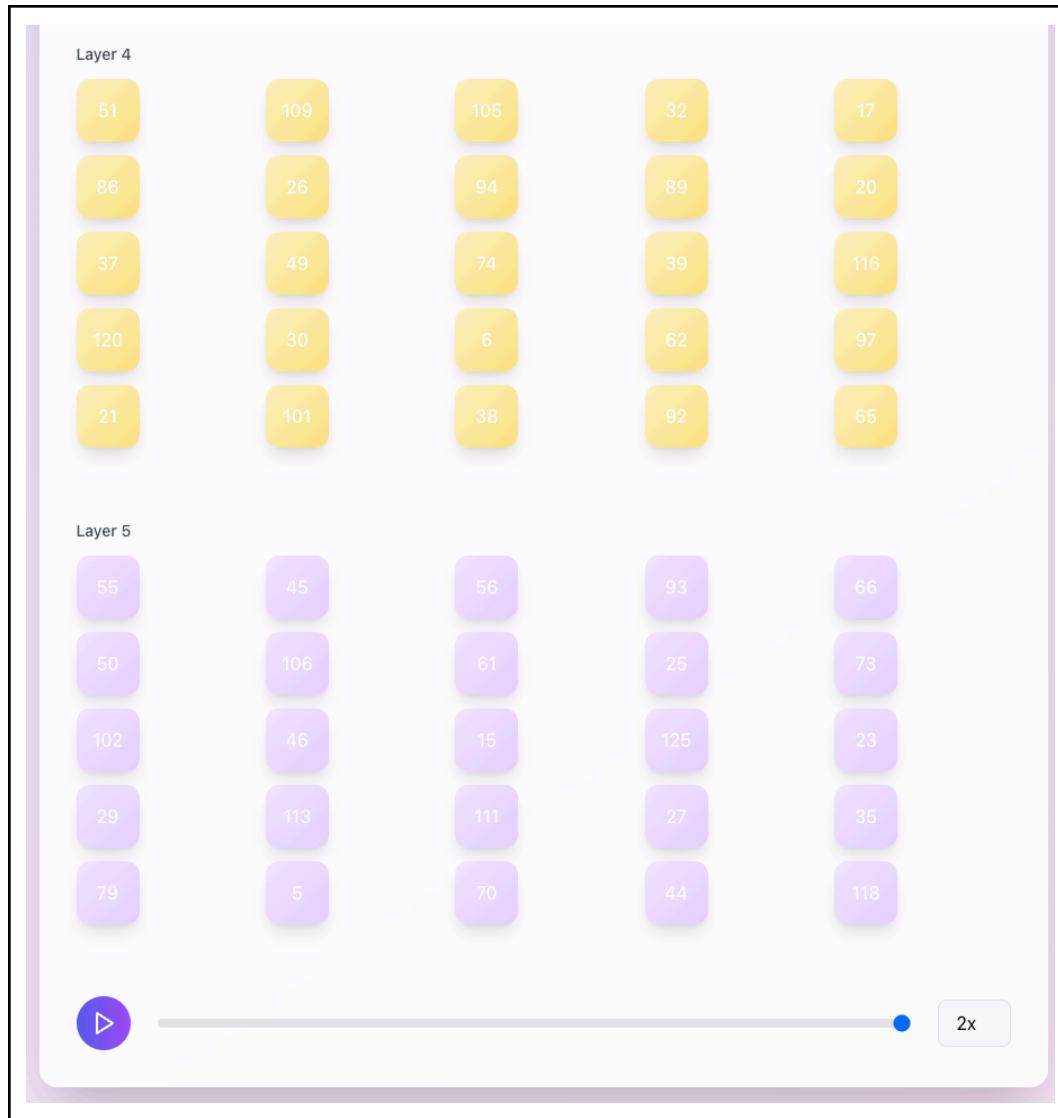
2	41	100	76	96
75	87	28	88	40
115	52	9	16	122
119	14	83	64	33
4	121	95	71	24

Layer 2

123	107	1	7	77
12	47	117	82	57
54	60	114	68	19
42	58	80	63	72
85	43	3	98	90

Layer 3

84	13	53	108	59
91	48	11	31	124
8	104	103	67	34
22	81	36	99	78
110	69	112	10	18



Eksperimen 1 :

- Iterasi : 105
- Initial Score : 6507
- Final Score : 292
- Total Time : 210 detik

Eksperimen 2 :

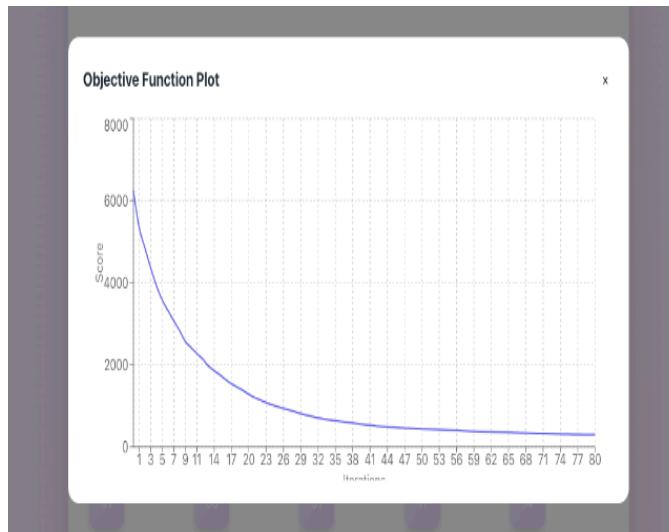
- Iterasi : 90
- Initial Score : 4854
- Final Score : 248
- Total Time : 222 detik

Eksperimen 3 :

- Iterasi : 98
- Initial Score : 6634
- Final Score : 130

- Total Time : 251 detik

Nilai objective Function dari 3 eksperimen yang terbaik adalah 130 dengan total iterasi adalah 98 kali. Namun, dari 3 eksperimen tersebut tidak ditemukan kepastian dalam hasilnya (nilai pasti menurun, tetapi tidak ada kepastian program akan mengembalikan nilai terendah dimana). Hal ini disebabkan oleh algoritma local search yang memungkinkan program untuk terjebak pada optimum local, serta algoritma program yang berhenti apabila menemukan nilai terendah tetangga yang sama besarnya atau lebih besar dari nilai score state tersebut. Ini adalah salah satu contoh plot persebaran nilai iterasi dan score nya:



b. Hill-climbing with Sideways Move

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

```

current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
loop do
  neighbor  $\leftarrow$  a highest-valued successor of current
  if neighbor.VALUE < current.VALUE then return current.STATE
  current  $\leftarrow$  neighbor

```

Terminates when it reaches a “peak”
including “flat”

Increase success for 8-queens problem.
Limit=100:
14% \rightarrow 94% success

Works **slower**:
when **success**: avg 4 \rightarrow 21 steps
when **stuck**: avg 3 \rightarrow 64 steps

Hill-climbing with Sideways Move adalah algoritma Steepest-Ascent Hill-Climbing yang memperbolehkan *sideways move*, yaitu pergerakan ke *neighbor* dengan *value* yang sama dengan *state* saat ini. Pergerakan ini dilakukan dengan harapan dapat keluar dari *local optima*.

1. Initial State
 - Menetapkan initial state dari magic cube secara random (inisialisasi kubus).
 - Menetapkan current value dari objective function pada kubus saat ini
2. Looping
 - Membangkitkan semua successor dari current state dengan melakukan swap antara setiap pasangan posisi secara sistematis.
 - Dari semua kemungkinan successor, dihitung nilai objective functionnya dan dipilih yang memiliki value terkecil (mendekati 0 atau goal state).
 - Karena pemilihan value terkecil dilakukan dengan perbandingan, untuk mencegah pemilihan state yang terus menerus ‘bolak-balik’ pada *sideways move*, list neighbor dishuffle secara random.
 - Penetapan pembaharuan current state dengan,
 - Jika value dari successor lebih kecil atau sama dengan current value, maka program akan mengganti nilai current state menjadi successor tersebut, lalu looping dilanjutkan.
 - Jika nilainya lebih besar atau sama dari current value, maka current value langsung direturn dan looping diberhentikan.
 - Jika sudah mencapai global atau local optimum maka iterasi bisa diberhentikan (sesuai yang dijelaskan pada tahap pembaharuan current state di atas).

Kelas:

```
1 class SidewaysHillClimbing:
2     def __init__(self, cube, max_sideways, max_iterations=1000):
3         self(cube = cube
4         self.max_iterations = max_iterations
5         self.max_sideways = max_sideways
6         self.list_result=[]
7
8     def run(self):
9         start_time = time.time()
10        current_score = self(cube).evaluate()
11        iteration = 0
12
13        while True:
14            if current_score == 0:
15                break
16
17            best_neighbor = None
18            best_score = current_score
19
20            neighbors = self(cube).get_neighbors()
21            random.shuffle(neighbors)
22
23            for neighbor in neighbors:
24                new_score = neighbor.evaluate()
25                if new_score <= best_score:
26                    if new_score < best_score:
27                        best_neighbor = neighbor
28                        best_score = new_score
29                else:
30                    if best_neighbor != neighbor:
31                        best_neighbor = neighbor
32                        best_score = new_score
33            if best_neighbor is None:
34                # print(1)
35                break
36            if best_score > current_score:
37                # print(2)
38                break
39            if self.max_sideways <= 0:
40                # print(self.max_sideways)
41                # print(3)
42                break
43            if iteration >= self.max_iterations:
44                # print(4)
45                break
46            if best_score == current_score:
47                self.max_sideways -= 1
48
49            self(cube = best_neighbor
50            current_score = best_score
51            iteration += 1
52            self.list_result.append((iteration, self(cube), current_score))
53        end_time = time.time()
54        total_time = end_time - start_time
55        return self.list_result, total_time
```

Hasil Eksperimen:

Initial State

Magic Cube Visualization

Iteration: 1

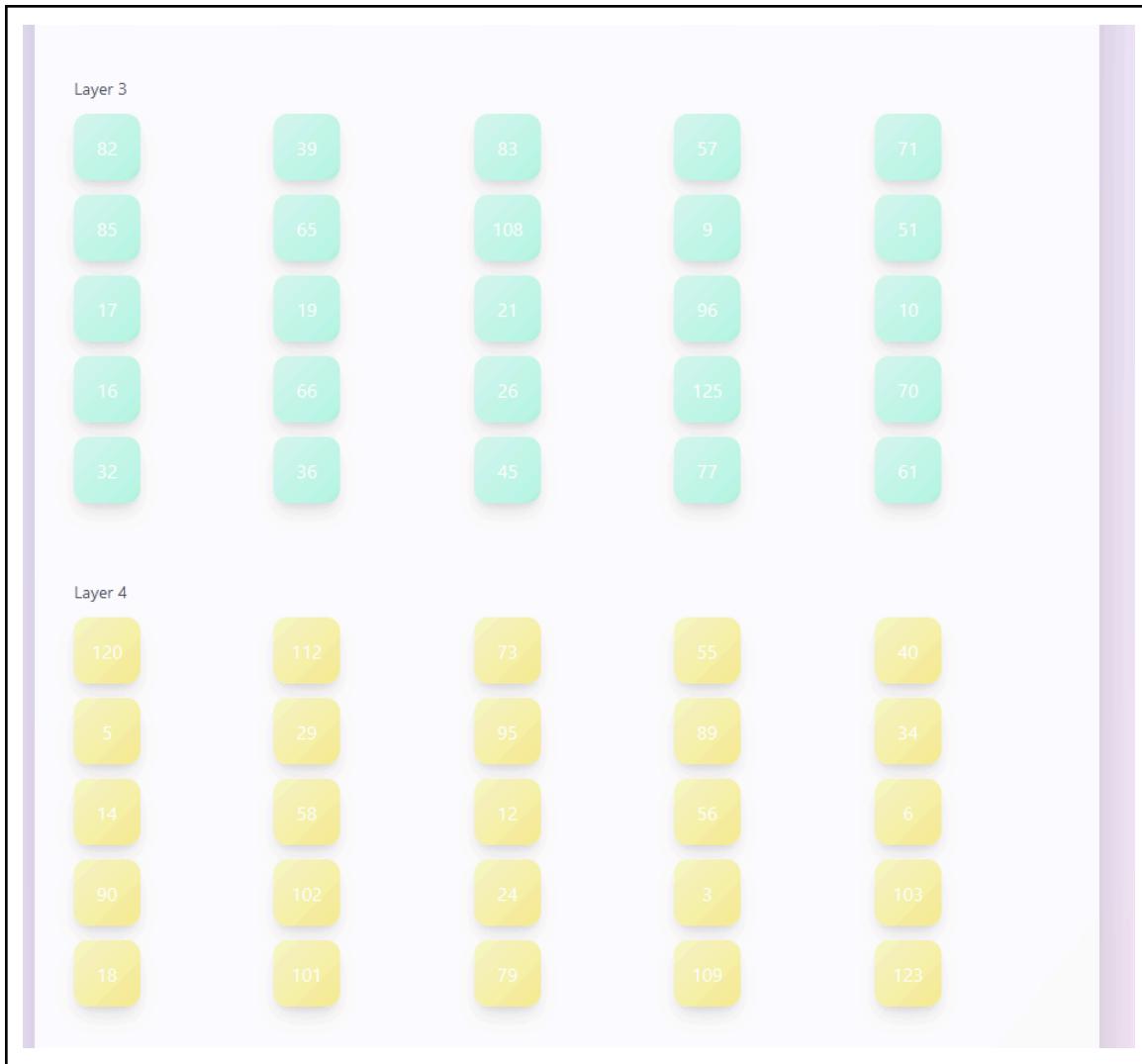
Score: 6360

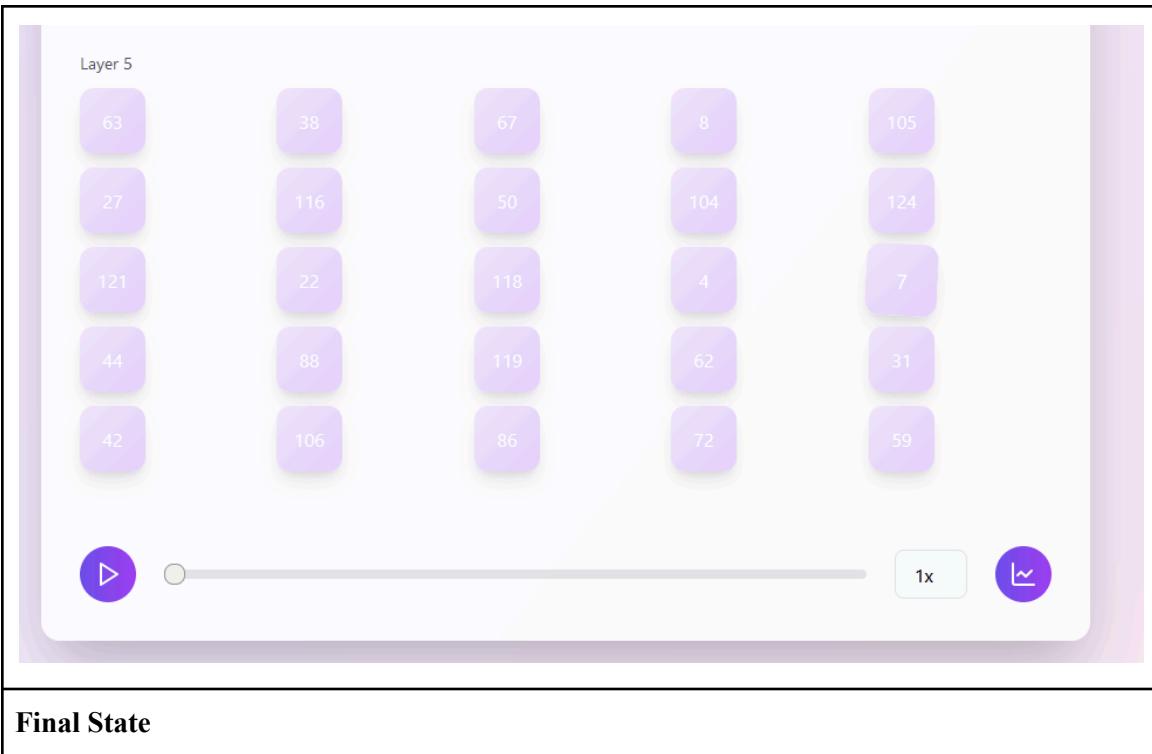
Layer 1

15	122	13	99	100
30	111	81	1	107
53	28	84	94	2
110	75	114	115	87
92	46	74	52	80

Layer 2

33	48	113	11	43
41	54	64	37	25
60	78	68	47	97
76	117	20	69	35
49	98	91	23	93





Magic Cube Visualization

Iteration: 295

Score: 760

Layer 1

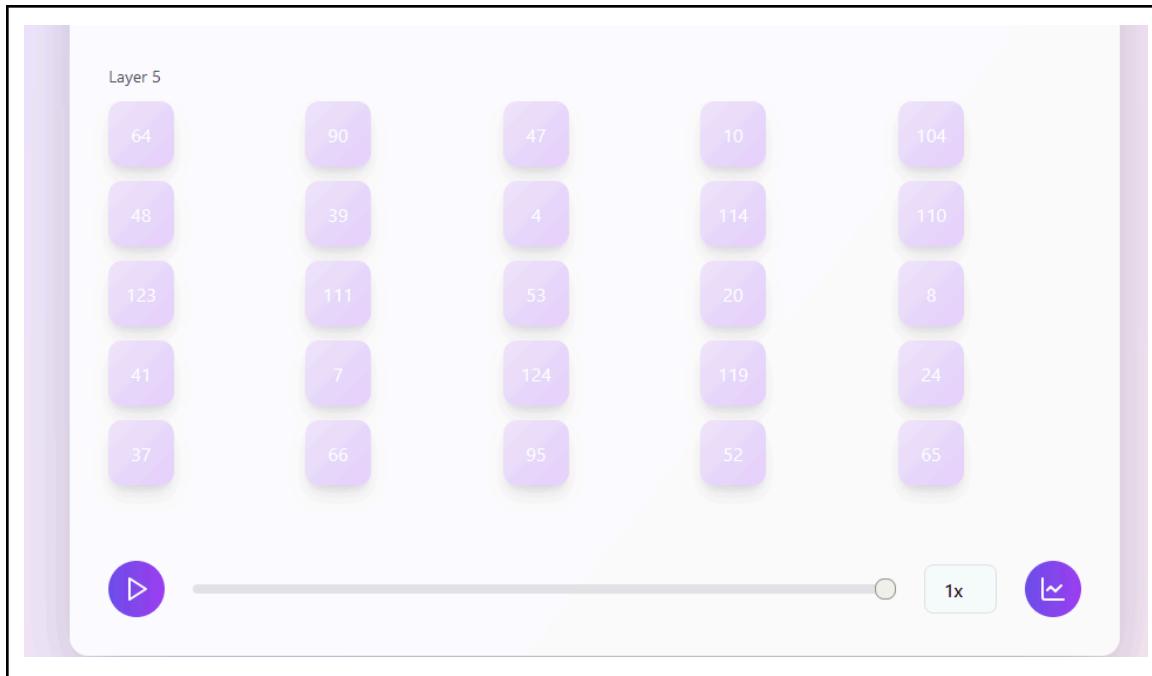
22	109	27	93	59
1	113	42	61	98
97	18	92	105	3
106	14	120	5	70
89	54	35	51	85

Layer 2

32	55	87	107	34
122	50	73	45	25
29	80	62	28	116
75	117	19	68	36
57	13	76	67	103

Layer 3				
30	74	94	72	43
108	63	78	9	60
49	15	77	102	69
11	81	40	99	84
115	82	26	33	58

Layer 4				
125	2	71	38	79
44	46	118	86	21
16	91	31	56	121
88	96	12	23	101
17	100	83	112	6



Eksperimen 1 :

- Iterasi : 295
- Initial Score : 6360
- Final Score : 760
- Total Time : 12 detik
- Max sideways move : 100

Eksperimen 2 :

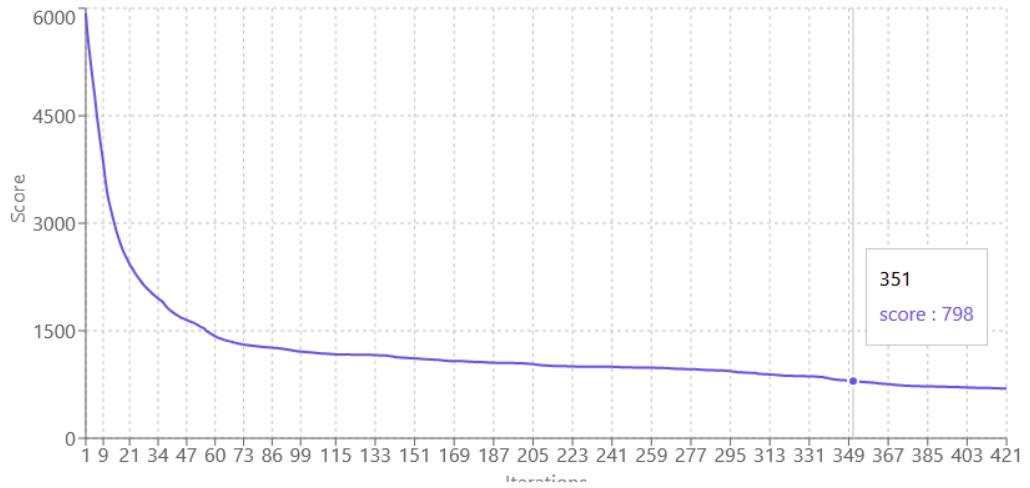
- Iterasi : 298
- Initial Score : 6086
- Final Score : 456
- Total Time : 13 detik
- Max sideways move : 100

Eksperimen 3 :

- Iterasi : 227
- Initial Score : 5932
- Final Score : 694
- Total Time : 19 detik
- Max sideways move : 100

Dalam eksperimen ini, kami menetapkan batas pergerakan ke samping, yaitu state dengan value yang sama, sebagai 100. Nilai objective Function dari 3 eksperimen yang terbaik adalah 456 dengan total iterasi adalah 298 kali. Namun, dari 3 eksperimen tersebut tidak ditemukan kepastian dalam hasilnya (nilai pasti menurun, tetapi tidak ada kepastian program akan mengembalikan nilai terendah dimana). Hal ini disebabkan oleh algoritma local search yang memungkinkan program untuk terjebak pada optimum local. Ini adalah salah satu contoh plot persebaran nilai iterasi dan score nya:

Objective Function Plot



Dapat dilihat bahwa ada beberapa bagian mendatar dalam grafik. Ini menandakan pergerakan *sideways*.

c. Random Restart Hill-climbing

If at first you don't succeed, try, try again. It conducts **a series of hill climbing** searches (random initial states, until a goal is found)

1. Initial State
 - Menetapkan jumlah max iteration
 - Menetapkan initial state dari magic cube secara random
 - Menetapkan current_value dari objective function current state saat ini
2. Looping Random-Restart
 - Melakukan looping hill climbing sebanyak max iteration
 - Jika sudah mencapai global optimum maka iterasi bisa diberhentikan (opsional).
3. Looping hill climbing
 - Membangkitkan semua successor dari current state mengambil neighbor dengan value terendah.
 - Menghitung value nya berdasarkan objective function
 - Jika value dari successor lebih kecil dari current_value, maka program akan mengganti nilai current state menjadi successor tersebut dan current_value menjadi nilai value tersebut lalu looping dilanjutkan.
 - Jika nilainya lebih besar atau sama dengan dari current_value, maka swap akan dibatalkan dan kondisi current akan dikembalikan lalu looping diberhentikan.

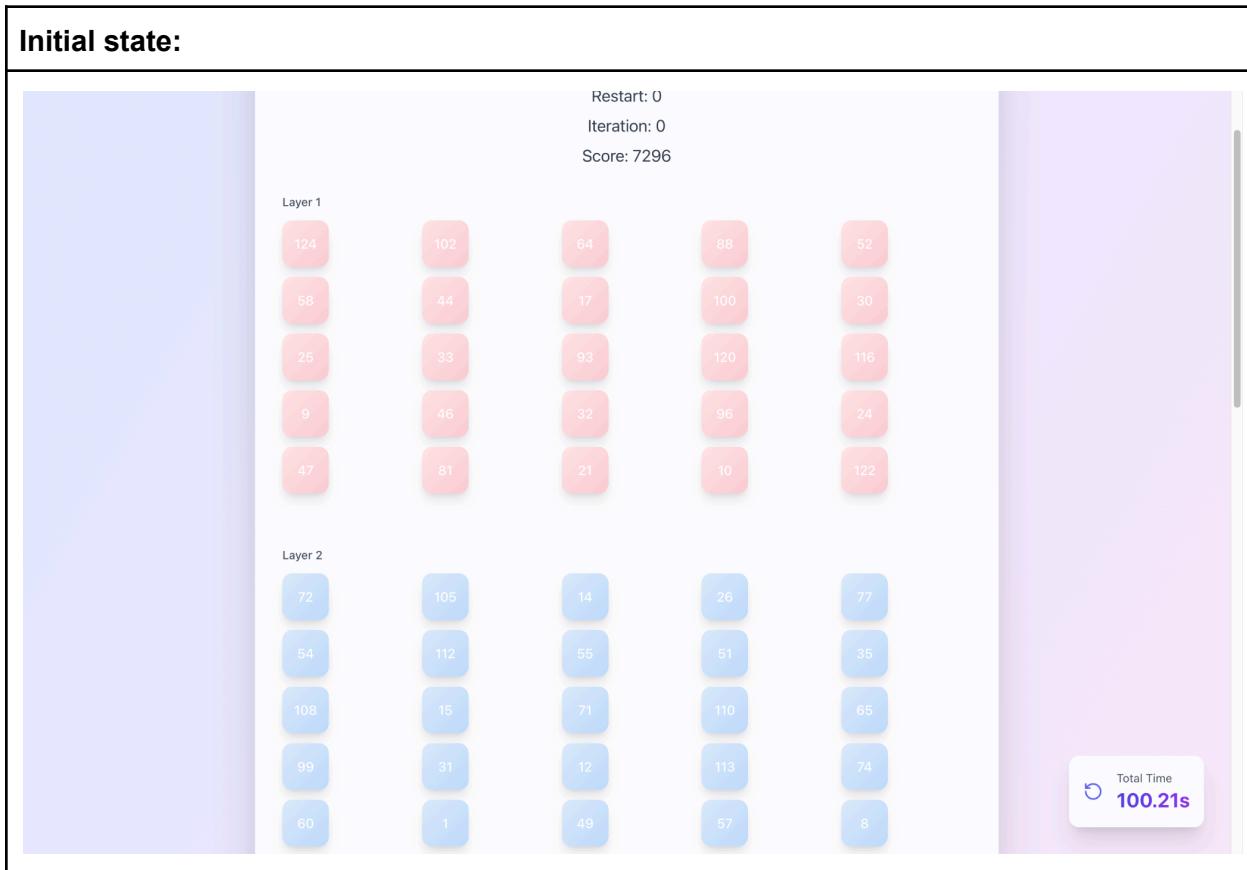
Eksperimen:

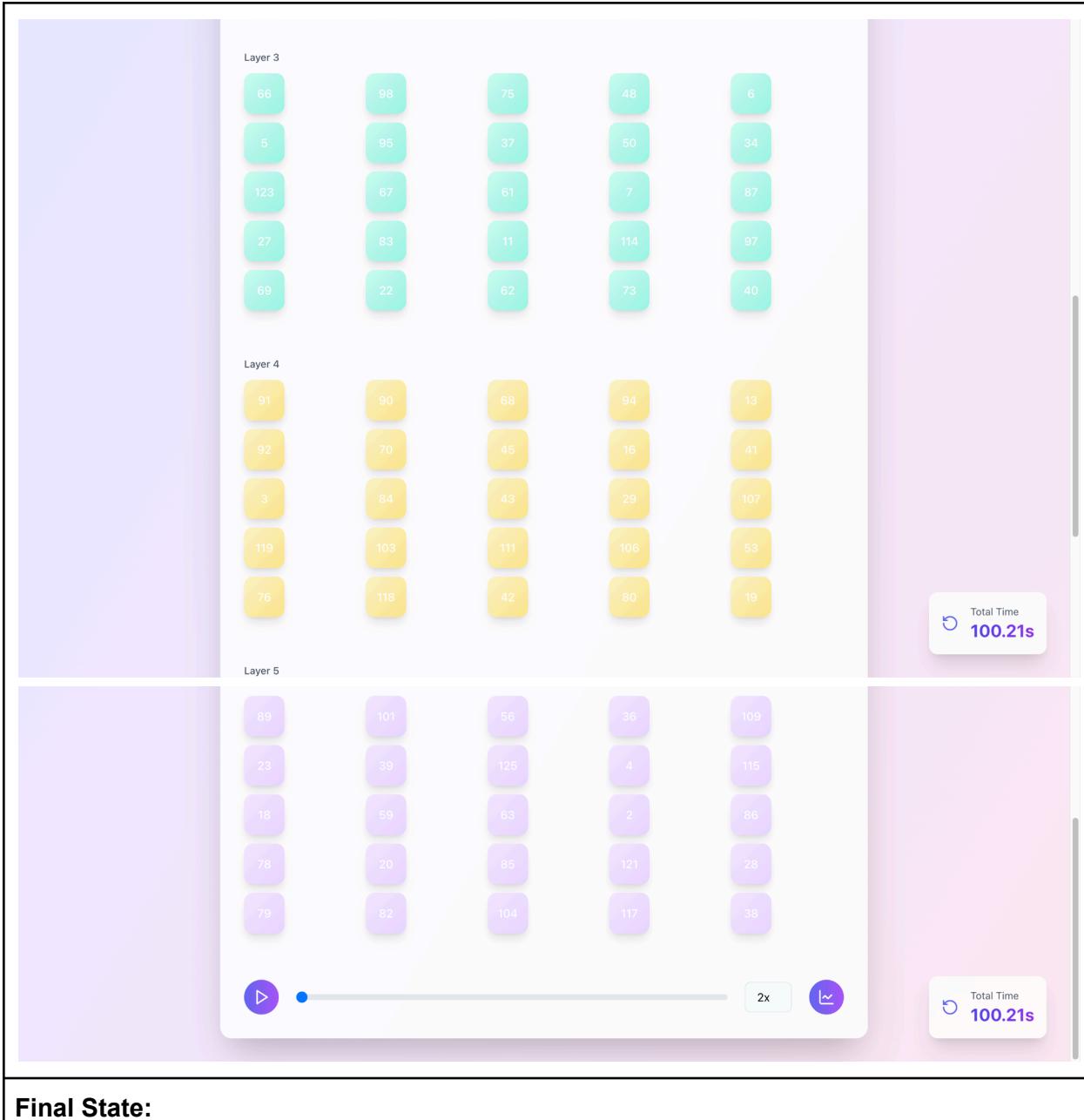


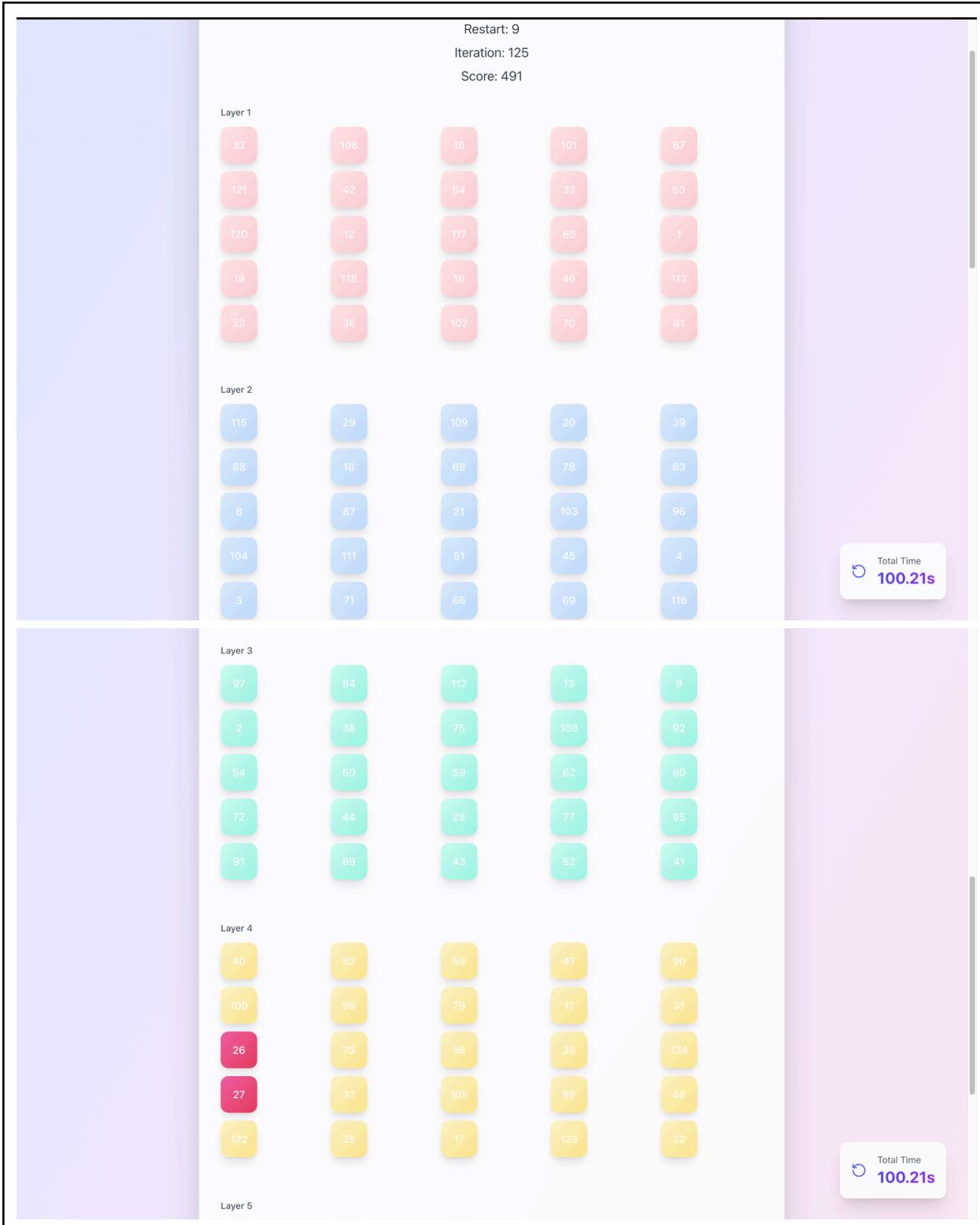
Tubes_AI - random_restart_HC.py

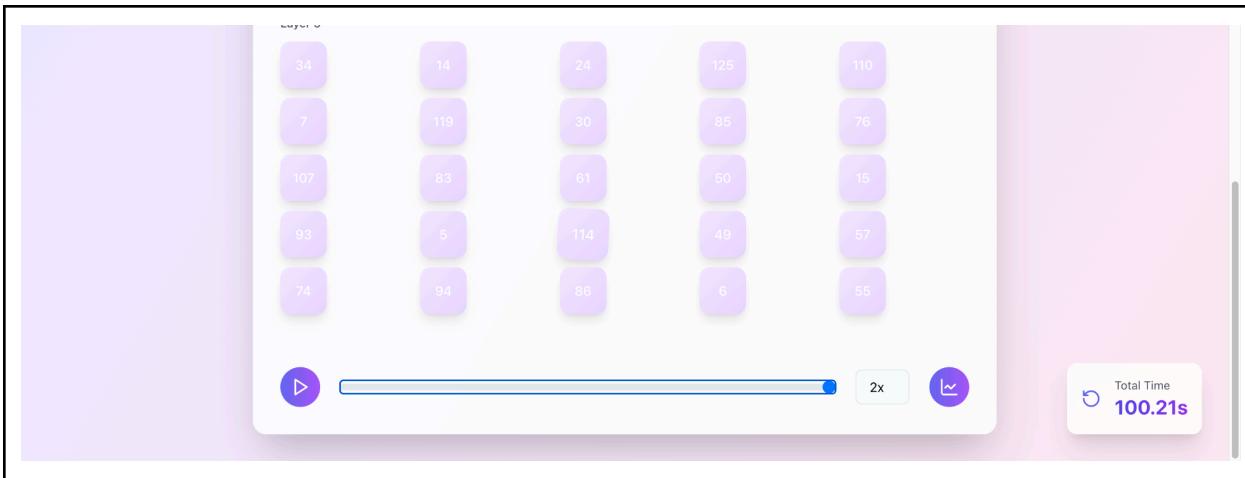
Secara keseluruhan kelas Random restart sama seperti hill climbing, bedanya random restart ini melakukan looping hill climbing pada tiap akhir state hill climbing dengan kondisi cube diacak kembali dan maksimal iterasi sesuai dengan parameter max_iteration.

Hasil Eksperimen:









Eksperimen 1 :

- Max Iterasi : 10
- Rata rata Iterasi : 87
- Initial Score : 7952
- Final Score : 290
- Total Time : 100 detik

Eksperimen 2 :

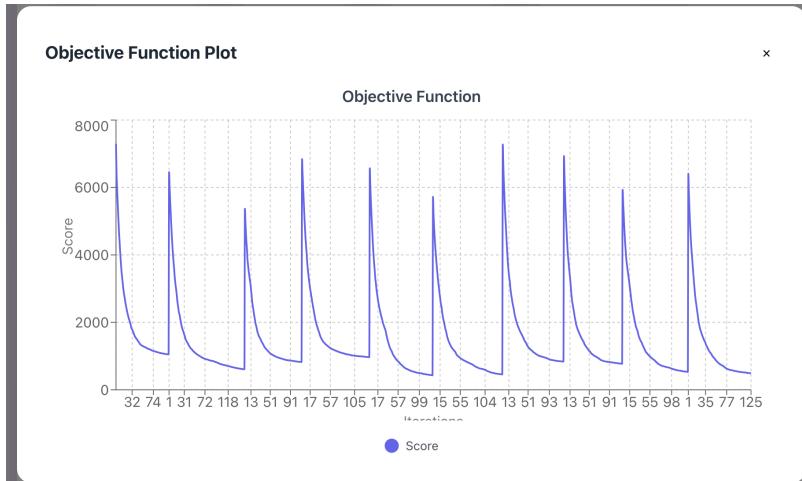
- Max Iterasi : 10
- Rata rata Iterasi : 92
- Initial Score : 7938
- Final Score : 224
- Total Time : 112 detik

Eksperimen 3 :

- Max Iterasi : 10
- Rata rata Iterasi : 84
- Initial Score : 6618
- Final Score : 245
- Total Time : 99 detik

Nilai objective Function dari 3 eksperimen yang terbaik adalah 224 dengan rata-rata iterasi adalah 92 kali. Namun, dari 3 eksperimen tersebut tidak ditemukan kepastian dalam hasilnya (nilai pasti menurun, tetapi tidak ada kepastian program akan mengembalikan nilai terendah dimana). Hal ini disebabkan oleh algoritma local search yang memungkinkan program untuk terjebak pada optimum local, serta algoritma program yang berhenti apabila menemukan nilai terendah tetangga yang sama besarnya atau lebih besar dari nilai score state tersebut, serta kondisi yang melakukan pengacakan ulang state cube jika tidak ditemukan nilai terbaik. Meskipun begitu algoritma ini seharusnya lebih baik dari steepest hill climbing karena melakukan restart pada tiap hasil

steepest hill climbing terbaik sebanyak max iteration. Ini adalah salah satu contoh plot persebaran nilai iterasi dan score nya :



d. Stochastic Hill-climbing

Variant 3: Stochastic Hill-climbing

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

```

current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
repeat nmax times
  neighbor  $\leftarrow$  a random successor of current
  if neighbor.VALUE  $>$  current.VALUE then current  $\leftarrow$  neighbor
```

Terminates
when it reaches
nmax iteration

Generating a
successor randomly
(**not all** successor)

Move to neighbor
if it is better than
current state.

Works **slower**
(more steps)

Stochastic Hill-climbing adalah algoritma hill-climbing yang membangkitkan sebuah suksesor acak daripada membangkitkan semua neighbor pada suatu *state*. Apabila suksesor yang dibangkitkan memiliki *value* dari objective function yang lebih baik dari *state* saat ini, neighbor tersebut dipilih menggantikan *state* saat ini. Apabila tidak, pembangkitan acak akan diulang.

1. Initial State

- Menetapkan initial state dari magic cube secara random (inisialisasi kubus).
- Menetapkan current value dari objective function pada kubus saat ini

2. Looping

- Membangkitkan sebuah suksesor acak dari *state* saat ini.
- Periksa value dari objective function pada suksesor tersebut.
- Apabila value lebih mendekati 0 dari value *state* saat ini, jadikan suksesor tersebut sebagai *state* saat ini. Lalu, lakukan iterasi selanjutnya.

Apabila value lebih jauh dari 0, ulangi penciptaan pembangkitan suksesor acak.

- Total pembangkitan suksesor acak adalah n kali. Dalam eksperimen ini, kami menetapkan n sebanyak 1 juta kali. Kami memilih n ini karena waktu yang dibutuhkan algoritma dengan n ini adalah 15-30 detik dan peningkatan n tidak memberi peningkatan hasil yang berarti.

Berikut adalah kelas yang kami gunakan untuk algoritma ini

```
● ● ●
1 class StochasticHillClimbing:
2     def __init__(self, cube):
3         self.cube = cube
4         self.list_result=[]
5
6     def run(self):
7         start_time = time.time()
8         current_score = self.cube.evaluate()
9         # print(self.cube.cube)
10        iteration = 0
11
12        for k in range(1000000):
13            # while(True):
14                if current_score == 0:
15                    break
16
17                best_score = current_score
18                pos1 = self.cube.get_random_position()
19                pos2 = self.cube.get_random_position()
20                self.cube.swap(pos1, pos2)
21                new_score = self.cube.evaluate()
22                if new_score < best_score:
23                    best_score = new_score
24                    current_score = best_score
25                    iteration+=1
26                    self.list_result.append((iteration, self.cube.cube, current_score))
27                else:
28                    self.cube.swap(pos1, pos2)
29            print(k)
30        end_time = time.time()
31        total_time = end_time - start_time
32        return self.list_result, total_time
```

Hasil Eksperimen

Initial State

Magic Cube Visualization

Iteration: 1

Score: 6447

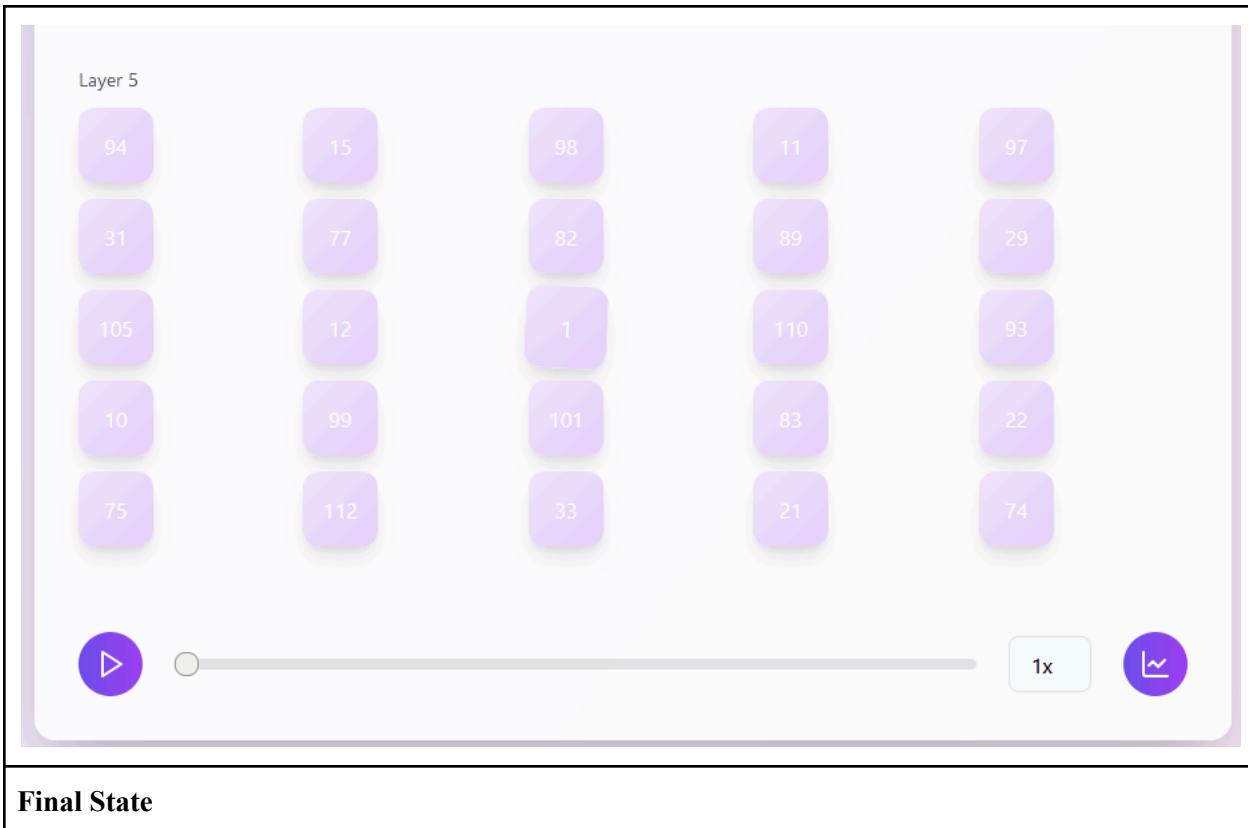
Layer 1

79	13	84	80	59
65	37	3	106	117
56	120	103	2	35
95	14	119	24	63
20	124	23	109	39

Layer 2

7	91	92	85	40
102	32	107	27	46
4	104	78	88	53
114	72	8	71	49
96	16	30	44	125

Layer 3				
73	121	6	51	64
48	111	81	41	36
18	47	62	115	70
54	17	58	67	116
122	19	108	38	28
Layer 4				
60	76	34	90	55
69	57	50	52	87
123	25	86	5	61
42	113	26	68	66
9	43	118	100	45



Magic Cube Visualization

Iteration: 362

Score: 674

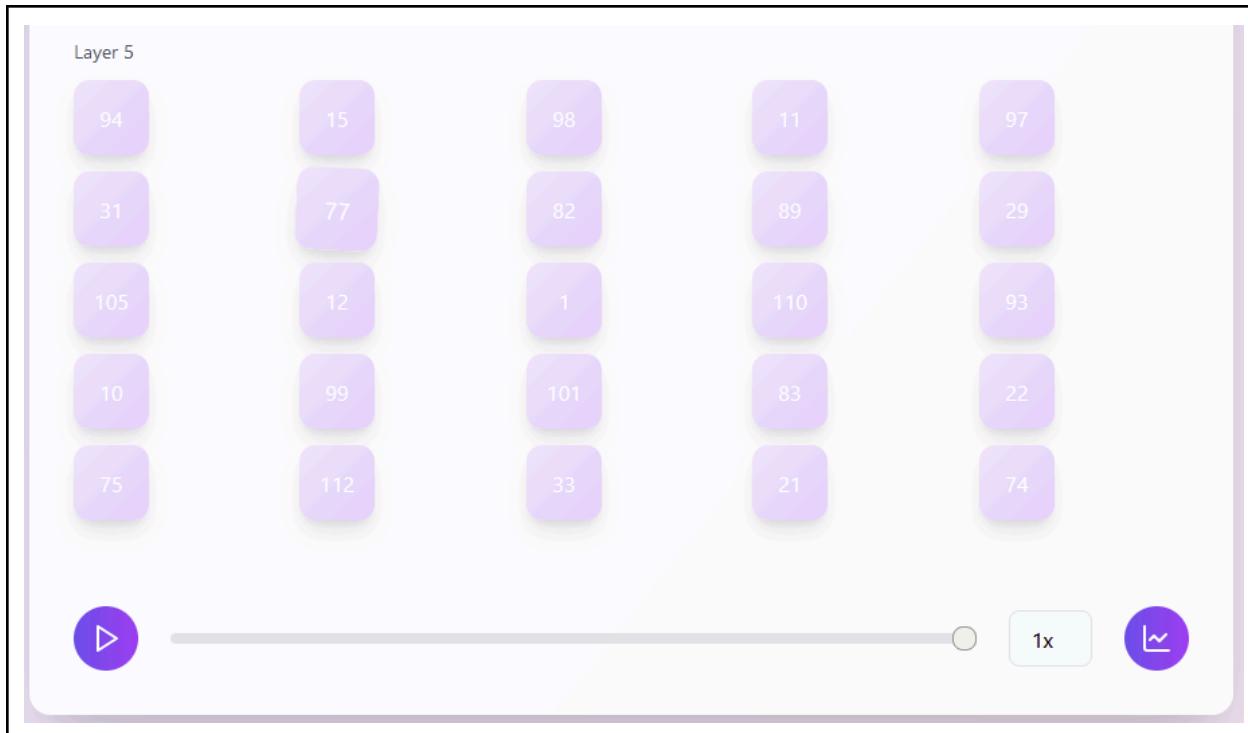
Layer 1

79	13	84	80	59
65	37	3	106	117
56	120	103	2	35
95	14	119	24	63
20	124	23	109	39

Layer 2

7	91	92	85	40
102	32	107	27	46
4	104	78	88	53
114	72	8	71	49
96	16	30	44	125

Layer 3				
73	121	6	51	64
48	111	81	41	36
18	47	62	115	70
54	17	58	67	116
122	19	108	38	28
Layer 4				
60	76	34	90	55
69	57	50	52	87
123	25	86	5	61
42	113	26	68	66
9	43	118	100	45



Eksperimen 1 :

- Nmax : 1000000
- Iterasi : 362
- Initial Score : 6447
- Final Score : 674
- Total Time : 20 detik

Eksperimen 2 :

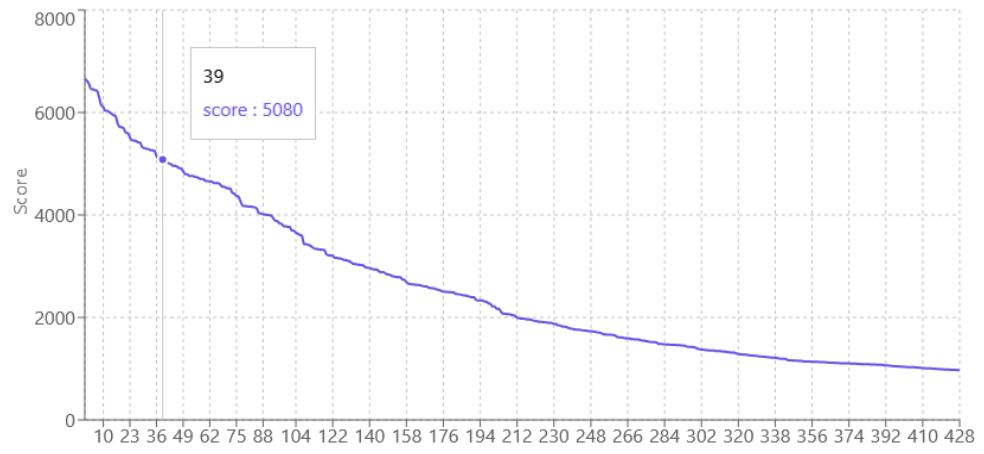
- Nmax : 1000000
- Iterasi : 428
- Initial Score : 6653
- Final Score : 972
- Total Time : 17 detik

Eksperimen 3 :

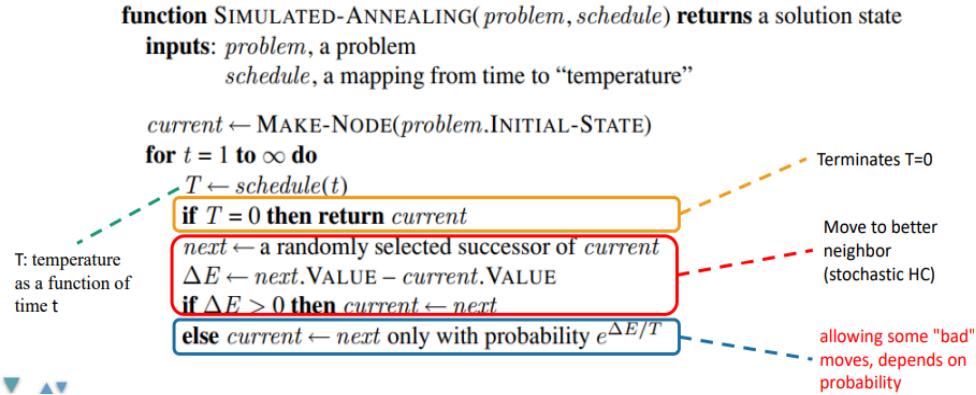
- Nmax : 1000000
- Iterasi : 418
- Initial Score : 6386
- Final Score : 606
- Total Time : 18 detik

Berikut adalah salah satu contoh plot iterasi dan nilai objective function algoritma Stochastic Hill-climbing

Objective Function Plot



e. Simulated Annealing



1. Initial State

- Menetapkan suhu awal T_0 (temperatur awal umumnya bernilai tinggi).
- Menetapkan laju pendinginan α (biasanya memiliki nilai $0 < \alpha < 1$)
- Menetapkan nilai energi awal (*current value*) dengan menghitung nilai fungsi objektif pada inisialisasi kubus awal.

2. Looping

- Memilih 2 posisi pada kubus secara acak lalu ditukar.
- Nilai fungsi objektif pada kubus yang telah ditukar 2 angkanya sebagai *next value*.
- Hitung ΔE , lalu bandingkan,
- Jika $\Delta E < 0$ (objektifnya ke nilai lebih kecil), maka *current value* diperbarui dengan *next value*.
- Sebaliknya, maka dihitung probabilitasnya ($P = \exp(-\Delta E/T)$) dan akan dibandingkan hasilnya dengan angka acak antara 0 dan 1.
- Jika angka acak tersebut $< P$, maka *current value* akan diperbarui nilainya.
- Setiap iterasi nilai *T* berkurang sesuai *cooling rate*-nya ($T' = T * \alpha^t$)
- Pengulangan berhenti dilakukan jika *T* bernilai 0 atau *goal state* sudah tercapai.

Eksperimen:

```

class SimulatedAnnealing:
    def __init__(self, cube, max_iterations=100, initial_temperature=10000, cooling_rate=0.9999):
        self(cube = cube
        self.max_iterations = max_iterations
        self.initial_temperature = initial_temperature
        self.cooling_rate = cooling_rate
        self.list_result = []

    def run(self):
        start_time = time.time()
        temperature = self.initial_temperature
        current_score = self(cube).evaluate()
        # print(self(cube))
        iteration = 0
        freq_local = 0
        self.list_result.append([iteration,(self(cube).cube).copy(),current_score, freq_local,1])

        while temperature > 1e-5:
            if current_score == 0:
                break

            pos1 = self(cube).get_random_position()
            pos2 = self(cube).get_random_position()
            while pos1 == pos2:
                pos2 = self(cube).get_random_position()

            self(cube).swap(pos1, pos2)
            new_score = self(cube).evaluate()

            delta = new_score - current_score
            if delta < 0:
                # Accept the new state
                current_score = new_score
                iteration += 1
                self.list_result.append([iteration,(self(cube).cube).copy(),current_score, freq_local,1])
            else:
                freq_local += 1
                prob = math.exp(-delta / temperature)
                if random.random() < prob:
                    # Accept the new state
                    current_score = new_score
                    iteration += 1
                    self.list_result.append([iteration,(self(cube).cube).copy(),current_score, freq_local,prob])
                else:
                    # Revert to the previous state
                    self(cube).swap(pos2, pos1)

            # Cool down the temperature
            temperature *= self.cooling_rate

        end_time = time.time()
        total_time = end_time - start_time
        return self.list_result, total_time

```

Kelas SimulatedAnnealing digunakan untuk menyelesaikan masalah magic cube menggunakan algoritma Simulated Annealing. Algoritma ini dimulai dengan inisialisasi suhu tinggi (initial_temperature) yang secara bertahap akan didinginkan sesuai dengan cooling rate.

Pada setiap iterasi, algoritma memilih dua posisi acak pada cube untuk ditukar dan menghitung skor baru susunan tersebut. Jika skor baru lebih rendah dari skor saat ini, perubahan diterima, dan algoritma berpindah ke state baru tersebut. Namun, jika skor baru lebih tinggi (artinya solusi kurang optimal), algoritma menentukan apakah akan menerima perubahan berdasarkan probabilitas yang dihitung menggunakan suhu saat ini. Semakin tinggi perbedaan skor, semakin kecil kemungkinan perubahan diterima, tetapi suhu tinggi meningkatkan peluang menerima perubahan meski kurang optimal. Hal ini membantu algoritma menghindari jebakan local minima dengan kemungkinan pindah ke solusi sementara yang kurang optimal. Suhu terus menurun hingga mencapai nilai mendekati nol atau solusi optimal ditemukan. Algoritma mencatat setiap perubahan state ke dalam list_result, termasuk iterasi keberapa, state cube pada iterasi tersebut, skor pada state tersebut, frekuensi penolakan, dan probabilitas penerimaan.

Hasil Eksperimen:

Initial state:

Magic Cube Visualization

Iteration: 0

Score: 6759

Frequency Local: 0

Layer 1

108	114	122	125	28
91	16	52	66	4
31	47	119	1	69
40	124	111	32	68
98	105	22	9	45

Layer 2

43	34	113	48	121
27	25	78	24	5
41	23	60	90	18
101	112	104	49	107
33	67	102	19	123

Layer 3

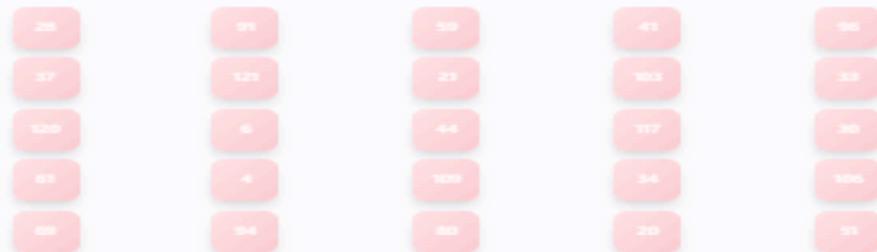
97	75	14	3	42
96	30	79	86	10
103	120	64	94	62
2	46	109	51	8
115	7	63	80	117

Layer 4				
12	15	92	82	95
17	35	21	11	61
50	70	118	99	20
88	6	29	83	54
73	13	81	71	72
Layer 5				
87	57	100	56	89
58	76	93	116	85
106	38	53	65	77
37	39	36	84	26
110	74	55	44	59
▶ ● 1x ◀				
Final State:				

Magic Cube Visualization

Iteration: 50283
Score: 191
Frequency Local: 182912

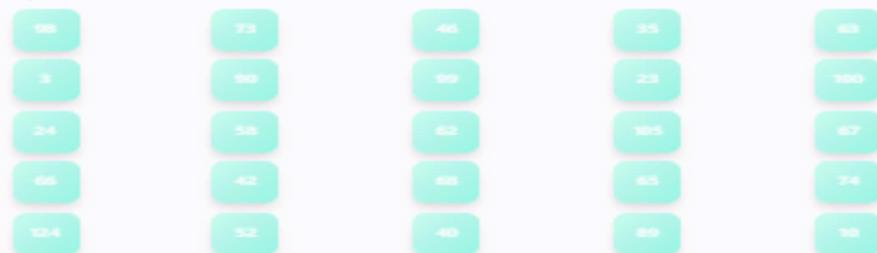
Layer 1



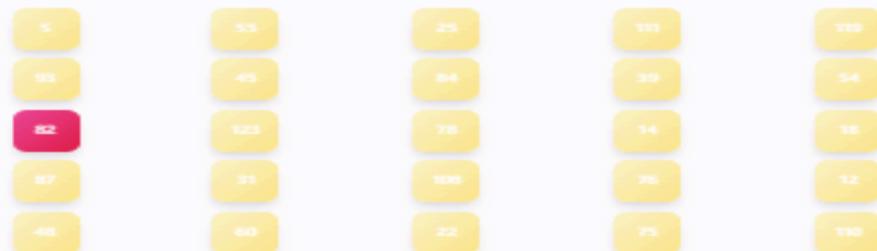
Layer 2

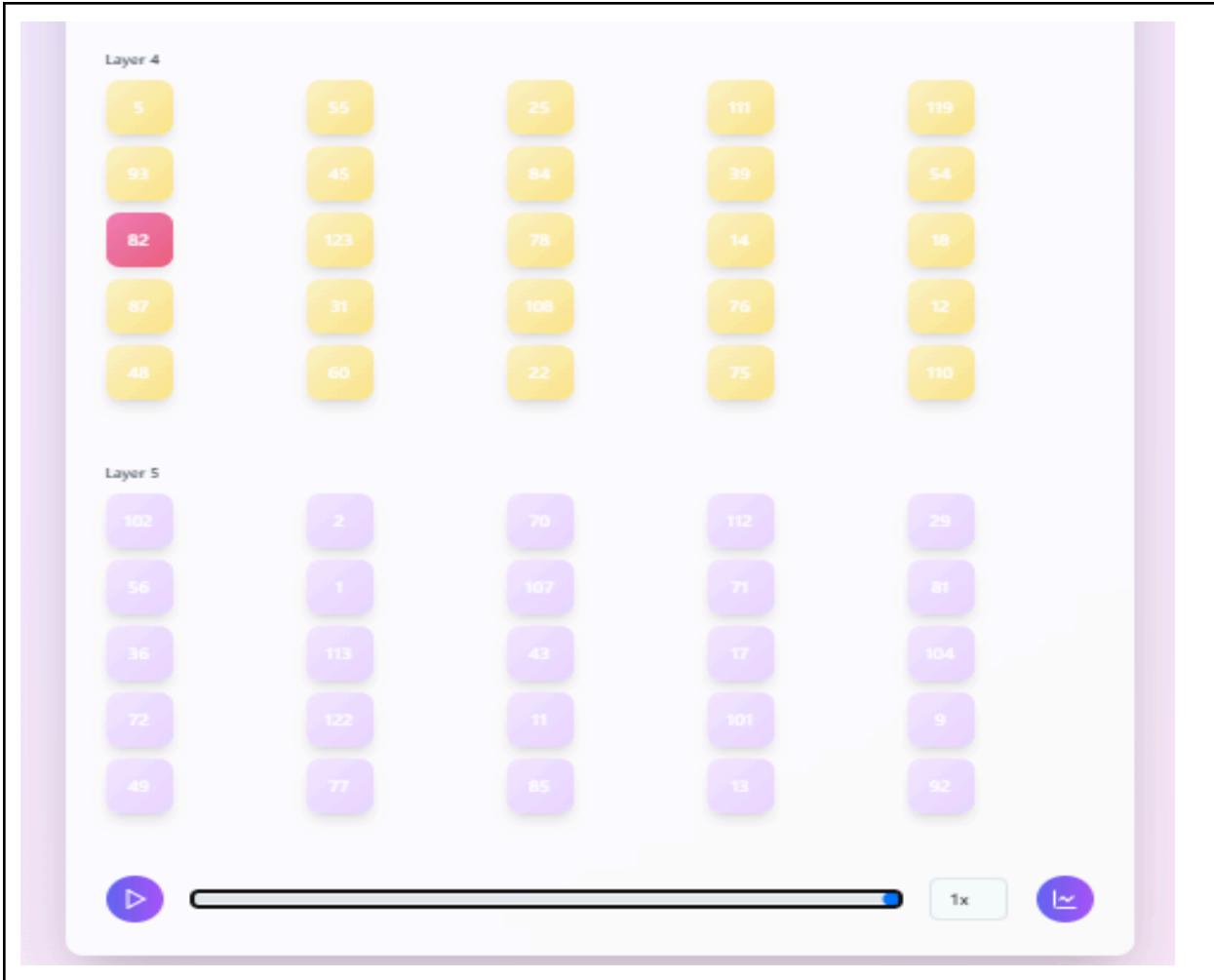


Layer 3



Layer 4





Eksperimen 1 :

- Iterasi : 50283
- Initial Score : 6759
- Final Score : 191
- Frequency Local: 183222
- Total Time : 19.98 detik

Eksperimen 2 :

- Iterasi : 50103
- Initial Score : 7421
- Final Score : 135
- Frequency Local: 183057
- Total Time : 13.24 detik

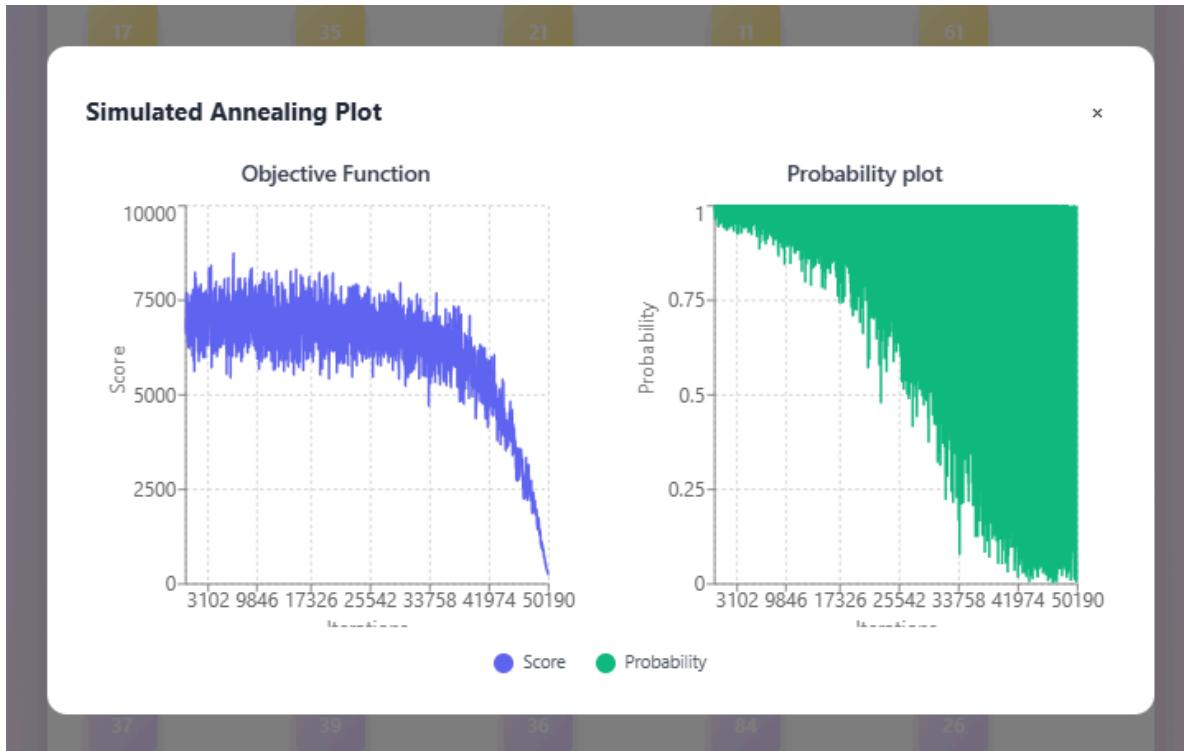
Eksperimen 3 :

- Iterasi : 50190
- Initial Score : 6442
- Final Score : 106

- Frequency Local: 182900
- Total Time : 15.78 detik

Hasil dari tiga eksperimen menunjukkan variasi skor akhir, namun dengan pola perbaikan yang konsisten. Pada eksperimen pertama, dengan iterasi sebanyak 50,283, algoritma berhasil menurunkan skor dari 6,759 menjadi 191, dengan frekuensi terjebak di kondisi lokal (local frequency) sebanyak 183,222 kali, dan waktu total 19,98 detik. Eksperimen kedua memiliki iterasi sedikit lebih rendah, yaitu 50,103, dan berhasil menurunkan skor dari 7,421 menjadi 135, dengan frekuensi lokal 183,057 dan waktu eksekusi yang lebih singkat, yakni 13,24 detik. Pada eksperimen ketiga, iterasi mencapai 50,190 kali, dengan skor awal 6,442 yang berkurang hingga 106, frekuensi lokal 182,900, dan waktu eksekusi 15,78 detik.

Dari ketiga eksperimen ini, terlihat bahwa nilai akhir terbaik dicapai pada eksperimen ketiga dengan skor 106 setelah 50,190 iterasi. Untuk konsistensi hasil, dari eksperimen tersebut didapatkan range solusi bernilai 100-200(bisa 200an) yang cukup mendekati optimum dibandingkan algoritma lainnya. Perbedaan waktu eksekusi juga menunjukkan bahwa kompleksitas perhitungan dapat berpengaruh terhadap proses pendinginan, dengan adanya eksperimen yang memiliki iterasi lebih sedikit, namun waktunya eksekusi lebih lama. Untuk durasi pencarian solusi, cukup cepat dibandingkan algoritma lainnya, kecuali Stochastic Hill Climbing karena kedua algoritma ini melakukan pemilihan neighbour secara random, jadi tidak generate all neighbour. Algoritma ini juga cenderung mengalami fluktuasi dalam nilai iterasi dan skor akhir, terutama karena probabilitas penerimaan solusi suboptimal yang tinggi di awal iterasi. Seiring suhu menurun (proses pendinginan), probabilitas menerima solusi suboptimal semakin kecil sehingga pemilihan solusi menjadi lebih selektif dan cenderung hanya menerima solusi yang lebih baik. Dengan kata lain, pendekatan ini memungkinkan algoritma untuk menjelajahi solusi yang lebih beragam di awal dan kemudian menyempitkan fokusnya pada solusi optimal saat suhu semakin rendah sesuai pada objective plot berikut.



f. Genetic Algorithm

Genetic Algorithm

```

function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population  $\leftarrow$  empty set
    for i = 1 to SIZE(population) do
      x  $\leftarrow$  RANDOM-SELECTION(population, FITNESS-FN)
      y  $\leftarrow$  RANDOM-SELECTION(population, FITNESS-FN)
      child  $\leftarrow$  REPRODUCE(x, y)
      if (small random probability) then child  $\leftarrow$  MUTATE(child)
      add child to new_population
    population  $\leftarrow$  new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
  
```

Implementasi algoritma ini didasarkan pada materi kuliah yang telah diajarkan di kelas. Proses diawali dengan melakukan generate population secara random sebanyak yang diinginkan. Kemudian, dari populasi tersebut akan dihitung fitness value dari masing masing individu. Nilai tersebut akan dibuat menjadi rentang persen untuk digunakan pada random roulette. Setelah itu akan dilakukan proses selection untuk memilih individu mana saja yang akan dilakukan persilangan. Proses selection dilakukan dengan melakukan random terhadap rentang angka [0..100%].

Pada setiap iterasi akan didapatkan 2 individu untuk dilakukan persilangan. Pada setiap iterasi akan dilakukan satu persilangan yang akan menghasilkan 2 individu baru. Dari dua individu baru tersebut akan dipilih individu dengan fitness value terbaik.

Sehingga akan terdapat populasi baru dengan jumlah yang sama dengan populasi sebelumnya.

Setelah itu populasi baru akan digabungkan dengan populasi yang lama. Saat ini populasi berjumlah dua kali dari semula. Setelah itu, akan dipilih sejumlah populasi lama dengan nilai fitness function terbaik. Hal ini adalah proses heuristik yang ditemukan secara mandiri dan terbukti meningkatkan state value.

EKSPERIMENT:

1. Populasi : 50
- Iterasi : 40

STATE AKHIR :

```
Final Objective function: 4356
Final state: [[[ 65  97  34  49  65]
   [120   7  78  27 106]
   [ 58 112  28 118  91]
   [  8 104 123  52  71]
   [ 68  60  75  63  32]]

[[ 71 118  13   1  84]
 [ 73   9  98  31 124]
 [ 81   6 114  61  19]
 [ 15  85 101  76  41]
 [ 26  53  50 107  57]]

[[ 60   5 116  25  86]
 [ 64 123  41 110   1]
 [[103  49  76  88  48]]
 [ 89  35  52  70  85]
 [ 60  61  27  53 113]]

[[ 83  36 106   5 121]
 [ 96 107  17  90  38]
 [ 59  34  77  99 125]
 [ 51  36 102  89  10]
 [ 22  99  30  56  68]]

[[ 72  55  93  67  12]
 [  4  29  82  80  42]
 [ 13 121  38  70 110]
 [ 95 119  33  20  24]
 [ 90  77  94  67  72]]]

Duration: 1.6250722408294678ms
```

2. Populasi : 50
- Iterasi : 50

STATE AKHIR :

```
Final Objective function: 4111
Final state: [[[ 42 33 38 109 84]
 [ 47 42 119 63 105]
 [ 14 125 30 25 37]
 [ 67 93 66 68 60]
 [ 52 120 58 19 61]]]

[[ 64 100 29 109 108]
 [110 59 50 73 7]
 [ 10 72 91 2 49]
 [ 84 3 36 56 95]
 [ 40 65 122 9 32]]]

[[ 89 116 45 101 4]
 [ 39 8 38 96 117]
 [ 79 41 102 44 56]
 [ 51 70 107 50 10]
 [ 88 22 26 119 90]]]

[[ 87 118 103 55 74]
 [ 35 102 6 115 57]
 [ 94 69 42 86 104]
 [113 47 62 37 57]
 [ 64 30 80 83 42]]]

[[ 34 106 97 7 52]
 [ 81 110 30 45 23]
 [ 85 15 69 77 67]
 [ 87 88 61 94 20]
 [ 18 82 68 91 73]]]

Duration: 1.9970717430114746ms
```

3. Populasi : 50
- Iterasi : 60

```
Final Objective function: 4409
Final state: [[[ 54  39  51  54  65]
   [ 74  99  49  27  40]
   [125  96  62  89  34]
   [ 55  41  53  83  52]
   [ 65  42 109  59  93]]]

[[ 57  24  68  71 125]
 [ 58  63  43  95  89]
 [ 87  25  60 107  76]
 [ 12  83  78 115   9]
 [ 91  53 111  34  50]]]

[[116  68  48 108  85]
 [ 14  49  79  61  70]
 [ 46 111  46  53 100]
 [ 80 124  43  38  28]
 [ 70  42  76  57   2]]]

[[ 47 105  16 104  41]
 [ 46  35 119  62  50]
 [ 77  10  54  81  86]
 [109  82  71  30 118]
 [ 62  86   5  66  48]]]

[[ 72  47 111  88  75]
 [ 69  54  32  53  40]
 [ 17  84 117  45  42]
 [ 60  59  63  85  77]
 [ 24  96 113  68  80]]]

Duration: 2.3294341564178467ms
```

4. Iterasi : 50
Populasi : 40

```
Final Objective function: 3770
Final state: [[[ 53  52  38  52 109]
   [ 65  77  59  20  25]
   [ 16  33 116 119  58]
   [  4  91  31  88  60]
   [ 58  60   1  62  53]]]

[[ 29  67    5 122 104]
 [ 21  40  94  84  10]
 [101 124  60   7  36]
 [113  82  86  63  64]
 [  9  44  11 108 121]]]

[[[110  55  85  26  19]
 [102  41  32  56  71]
 [ 66  25  68  27  44]
 [  3 117  54  80  23]
 [ 54  55  94  90  53]]]

[[ 82  61  72  31  46]
 [ 18 101 103  91  36]
 [ 29 104   7  95  72]
 [119  20  30  86  60]
 [ 76  38 108  37  56]]]

[[[ 14  86 107  21  28]
 [ 96  32  34  64 107]
 [100  47  85   9  69]
 [ 88  23  12  74  97]
 [ 71 119  95  41  68]]]
Duration: 1.4976449012756348ms
```

-
- 5. Iterasi : 50
 - Populasi : 60

```
Final Objective function: 3960
✓ Final state: [[[ 48  63  76  76  48]
   [ 53  86  31  98  39]
   [108 119  51  80 110]
   [ 91  40  97   9  95]
   [ 84  14  46  35 104]]]

✓ [[ 79  90  94  43  62]
   [ 58  87  17  63  83]
   [ 89  11  38  79  65]
   [ 16  33  82 113   6]
   [ 77  96  22 101  44]]]

✓ [[ 23  79 114  27  29]
   [ 97 112  59 101  18]
   [ 68  24  49  94  74]
   [ 91  71  36  53  84]
   [ 79  32  44  70 107]]]

✓ [[[124  10  24  68 109]
   [ 16  48 102  37  59]
   [ 19  74 100  80  44]
   [ 65  32  42  75 119]
   [ 25  82 115  34  40]]]

✓ [[ 83  94  51  31  85]
   [ 79    6  99 106  72]
   [ 68 121 104  11  50]
   [ 23  36    1  77  52]
   [ 54 101 106  53    7]]]
Duration: 2.3652451038360596ms
```

6. Iterasi : 50
Populasi : 70

```
Final Objective function: 3473
Final state: [[[ 74  58  54  74  58]
   [ 15 116  84   2 103]
   [ 15  68  57  81  53]
   [106  74  37  16  58]
   [107  78  47 104  34]]]

[[ 24  44 104 113  34]
 [ 91  66  99  21  64]
 [ 42  63  73  94  28]
 [ 10 102  56  69 100]
 [ 90  57  68   9  96]]]

[[110  43  75  37  79]
 [ 89  30  62  82  59]
 [115  81  55  59  62]
 [ 33  35 116 110  71]
 [ 14  75  63 108   6]]]

[[ 46  39  77  83  86]
 [ 52  85 107  25  35]
 [ 65  27  77 122  82]
 [ 78  35  18  51  89]
 [ 80 110  42  53  40]]]

[[ 67  92  63  28  46]
 [ 54  37  85  90  42]
 [ 47  69  88  34  77]
 [102  74  56  55  42]
 [ 39  28 104  92  59]]]
Duration: 2.725715398788452ms
```

Dari hasil eksperimen tersebut dapat dilihat kedua hal tersebut mengakibatkan waktu penggerjaan yang berbeda. Semakin besar nilai iterasi atau populasi nya maka akan menghasilkan waktu yang lebih lama. Dan jika dibandingkan antara variabel populasi dan variabel iterasi, dari hasil eksperimen, dapat dilihat perbedaan populasi lebih efektif dibandingkan iterasi. Semakin besar nilai populasi semakin kecil pula hasil akhir score nya. Sebenarnya keduanya menunjukan perubahan, tetapi perubahan populasi lebih menghasilkan hasil yang signifikan.

III. Kesimpulan dan Saran

Dari semua algoritma hill climbing, didapat hasil akhir eksperimen dengan algoritma Random Restart yang memiliki nilai akhir dari objective function mendekati 0 (mendekati solusi). Namun, jika dibandingkan dengan ketiga algoritma hill climbing lainnya, waktu komputasi yang dibutuhkan untuk menyelesaiakannya relatif lebih lama apalagi jika max iteration yang diberikannya cukup besar (>10). Meskipun begitu, secara keseluruhan algoritma hill climbing tidak ada yang dapat memberikan kepastian akan selalu stabil memberikan nilai terbaik. Dalam konteks ini, algoritma Simulated Annealing juga menunjukkan performa yang baik, meskipun dengan karakteristik yang berbeda. Simulated Annealing mampu menjelajahi solusi yang lebih beragam di awal proses, dan seiring dengan penurunan suhu, pemilihan solusi menjadi lebih selektif, sehingga meningkatkan kemungkinan untuk mencapai solusi optimal pada akhir iterasi. Dari eksperimen, juga didapatkan range nilai berada di kisaran 100-200 yang cukup memuaskan dengan waktu yang cukup cepat. Namun, sama seperti algoritma *hill climbing* lainnya, *Simulated Annealing* juga tidak dapat memberikan kepastian untuk selalu menghasilkan nilai terbaik, dan waktu komputasi yang dibutuhkan dapat bervariasi tergantung pada parameter yang ditetapkan serta kompleksitas masalah yang dihadapi. Sedangkan Genetic Algorithm dapat memberikan hasil yang pasti dengan syarat jumlah populasi dan iterasi yang diberikannya besar. Semakin besar kedua nilai tersebut maka dipastikan nilai akhir yang didapat sangat menuju solusi, tetapi kedua hal tersebut mengakibatkan kebutuhan penggunaan memori yang cukup besar atau biaya komputasi yang mahal. Oleh karena itu saran yang dapat kami berikan adalah menemukan cara penjalanan algoritma dengan menggunakan cpu atau pc yang lebih canggih untuk mengatasi hal tersebut.

IV. Pembagian Tugas

Mohammad Nugraha Eka Prawira	Algoritma Steepest Ascent dan Random Restart Hill Climbing Bonus, Laporan, Backend Bonus dan Frontend Bonus.
Ahmad Farid Mudrika	Algoritma Hill-Climbing with Sideways Move dan Stochastic Hill-Climbing, Laporan, dan Frontend Bonus.
Muhammad Yusuf Rafi	Algoritma Simulated Annealing, Backend API, dan FrontEnd tambahan Simulated Annealing.
Muhamad Rifki Virzadeili	Algoritma Genetika, Laporan

V. Referensi

- <https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>
[Features of the magic cube - Magisch vierkant](#)
<https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/>
<https://www.baeldung.com/cs/simulated-annealing>
<https://www.geeksforgeeks.org/genetic-algorithms/>