

**Tugas Besar 1 IF3070 Dasar Inteligensi Artifisial Pencarian
Solusi Diagonal Magic Cube dengan Local Search**



Oleh

Kelompok 30

Tamara Mayranda Lubis (18222026)

Yovanka Sandrina Maharaja (18222094)

Bryan P. Hutagalung (18222130)

Yusril Fazri Mahendra (18222141)

Program Studi Sistem dan Teknologi Informasi

Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

2024

Daftar Isi

Daftar Isi	2
Daftar Tabel	3
Daftar Gambar	5
BAB I Deskripsi Persoalan	8
BAB II Hasil dan Pembahasan	9
A. Objective Function	9
B. Utilities	15
C. Local Search	16
1. Steepest Ascent Hill-climbing	16
2. Hill-climbing with Sideways Move	18
3. Stochastic Hill-climbing	22
4. Random Restart Hill-climbing	25
5. Simulated Annealing	28
6. Genetic Algorithm	32
D. Integrasi	37
1. main.py	37
2. App.jsx	41
E. Hasil Eksperimen dan Analisis	46
1. Eksperimen I	47
a. Inisialisasi Cube	47
b. Steepest Ascent Hill-climbing	48
c. Hill-climbing with Sideways Move	49
d. Stochastic Hill-climbing	50
e. Random Restart Hill-climbing	51
f. Simulated Annealing	52
g. Genetic Algorithm	53
i. Population = 300 Iteration = 500	53
ii. Population = 300 Iteration = 1000	56
iii. Population = 300 Iteration = 1500	59
iv. Iteration = 1500 Population = 50	62
v. Iteration = 1500 Population = 100	65
vi. Iteration = 1500 Population = 150	68
2. Eksperimen II	71
a. Inisialisasi Cube	71
b. Steepest Ascent Hill-climbing	72
c. Hill-climbing with Sideways Move	73

d. Stochastic Hill-climbing	74
e. Random Restart Hill-climbing	75
f. Simulated Annealing	76
g. Genetic Algorithm	79
i. Population = 300 Iteration = 500	79
ii. Population = 300 Iteration = 1000	81
iii. Population = 300 Iteration = 1500	84
iv. Iteration = 1500 Population = 50	87
v. Iteration = 1500 Population = 100	90
vi. Iteration = 1500 Population = 150	93
3. Eksperimen III	96
a. Inisialisasi Cube	96
b. Steepest Ascent Hill-climbing	97
c. Hill-climbing with Sideways Move	98
d. Stochastic Hill-climbing	99
e. Random Restart Hill-climbing	100
f. Simulated Annealing	101
g. Genetic Algorithm	103
i. Population = 300 Iteration = 500	103
ii. Population = 300 Iteration = 1000	105
iii. Population = 300 Iteration = 1500	108
iv. Iteration = 1500 Population = 50	111
v. Iteration = 1500 Population = 100	114
vi. Iteration = 1500 Population = 150	117
F. Bonus	123
1. Seluruh Algoritma	123
2. Video Player	123
i. Source Code	123
ii. Eksperimen	128
3. Save and Load	130
i. Source Code	130
ii. Eksperimen	135
BAB III Kesimpulan dan Saran	137
A. Kesimpulan	137
B. Saran	137
Pembagian Tugas	138
Referensi	139

Daftar Tabel

Tabel 2.1 Perbandingan Algoritma Eksperimen I	120
Tabel 2.2 Perbandingan Algoritma Eksperimen II	121
Tabel 2.3 Perbandingan Algoritma Eksperimen III	122

Daftar Gambar

Gambar 2.1 Laman Awal Visualisasi	46
Gambar 2.2 Inisialisasi Cube I	48
Gambar 2.3 Steepest Ascent Hill-climbing Eksperimen I	49
Gambar 2.4 Hill-climbing with Sideways Move Eksperimen I	50
Gambar 2.5 Stochastic Hill-climbing Eksperimen I	51
Gambar 2.6 Random Restart Hill-climbing Eksperimen I	52
Gambar 2.7 Simulated Annealing Eksperimen I	53
Gambar 2.8 Genetic Algorithm i (1) Eksperimen I	54
Gambar 2.9 Genetic Algorithm i (2) Eksperimen I	55
Gambar 2.10 Genetic Algorithm i (3) Eksperimen I	56
Gambar 2.11 Genetic Algorithm ii (1) Eksperimen I	57
Gambar 2.12 Genetic Algorithm ii (2) Eksperimen I	58
Gambar 2.13 Genetic Algorithm ii (3) Eksperimen I	59
Gambar 2.14 Genetic Algorithm iii (1) Eksperimen I	60
Gambar 2.15 Genetic Algorithm iii (2) Eksperimen I	61
Gambar 2.16 Genetic Algorithm iii (3) Eksperimen I	62
Gambar 2.17 Genetic Algorithm iv (1) Eksperimen I	63
Gambar 2.18 Genetic Algorithm iv (2) Eksperimen I	64
Gambar 2.19 Genetic Algorithm iv (3) Eksperimen I	65
Gambar 2.20 Genetic Algorithm v (1) Eksperimen I	66
Gambar 2.21 Genetic Algorithm v (2) Eksperimen I	67
Gambar 2.22 Genetic Algorithm v (3) Eksperimen I	68
Gambar 2.23 Genetic Algorithm vi (1) Eksperimen I	69
Gambar 2.24 Genetic Algorithm vi (2) Eksperimen I	70
Gambar 2.25 Genetic Algorithm vi (3) Eksperimen I	71
Gambar 2.26 Inisialisasi Cube II	72
Gambar 2.27 Steepest Ascent Hill-climbing Eksperimen II	73
Gambar 2.28 Hill-climbing with Sideways Move Eksperimen II	74
Gambar 2.29 Stochastic Hill-climbing Eksperimen II	75
Gambar 2.30 Random Restart Hill-climbing Eksperimen II	76
Gambar 2.31 Simulated Annealing Eksperimen II	78
Gambar 2.32 Genetic Algorithm i (1) Eksperimen II	79
Gambar 2.33 Genetic Algorithm i (2) Eksperimen II	80
Gambar 2.34 Genetic Algorithm i (3) Eksperimen II	81
Gambar 2.35 Genetic Algorithm ii (1) Eksperimen II	82

Gambar 2.36 Genetic Algorithm ii (2) Eksperimen II	83
Gambar 2.37 Genetic Algorithm ii (3) Eksperimen II	84
Gambar 2.38 Genetic Algorithm iii (1) Eksperimen III	85
Gambar 2.39 Genetic Algorithm iii (2) Eksperimen III	86
Gambar 2.40 Genetic Algorithm iii (3) Eksperimen III	87
Gambar 2.41 Genetic Algorithm iv (1) Eksperimen I	88
Gambar 2.42 Genetic Algorithm iv (2) Eksperimen I	89
Gambar 2.43 Genetic Algorithm iv (3) Eksperimen I	90
Gambar 2.44 Genetic Algorithm v (1) Eksperimen I	91
Gambar 2.45 Genetic Algorithm v (2) Eksperimen I	92
Gambar 2.46 Genetic Algorithm v (3) Eksperimen I	93
Gambar 2.47 Genetic Algorithm vi (1) Eksperimen II	94
Gambar 2.48 Genetic Algorithm vi (2) Eksperimen II	95
Gambar 2.49 Genetic Algorithm vi (3) Eksperimen II	96
Gambar 2.50 Inisialisasi Cube III	97
Gambar 2.51 Steepest Ascent Hill-climbing Eksperimen III	98
Gambar 2.52 Hill-climbing with Sideways Move Eksperimen III	99
Gambar 2.53 Stochastic Hill-climbing Eksperimen III	100
Gambar 2.54 Random Restart Hill-climbing Eksperimen II	101
Gambar 2.55 Simulated Annealing Eksperimen III	102
Gambar 2.56 Genetic Algorithm i (1) Eksperimen III	103
Gambar 2.57 Genetic Algorithm i (2) Eksperimen III	104
Gambar 2.58 Genetic Algorithm i (3) Eksperimen III	105
Gambar 2.59 Genetic Algorithm ii (1) Eksperimen III	106
Gambar 2.60 Genetic Algorithm ii (2) Eksperimen III	107
Gambar 2.61 Genetic Algorithm ii (3) Eksperimen III	108
Gambar 2.62 Genetic Algorithm iii (1) Eksperimen III	109
Gambar 2.63 Genetic Algorithm iii (2) Eksperimen III	110
Gambar 2.64 Genetic Algorithm iii (3) Eksperimen III	111
Gambar 2.65 Genetic Algorithm iv (1) Eksperimen III	112
Gambar 2.66 Genetic Algorithm iv (2) Eksperimen III	113
Gambar 2.67 Genetic Algorithm iv (3) Eksperimen III	114
Gambar 2.68 Genetic Algorithm v (1) Eksperimen III	115
Gambar 2.69 Genetic Algorithm v (2) Eksperimen III	116
Gambar 2.70 Genetic Algorithm v (3) Eksperimen III	117
Gambar 2.71 Genetic Algorithm vi (1) Eksperimen III	118
Gambar 2.72 Genetic Algorithm vi (2) Eksperimen III	119
Gambar 2.73 Genetic Algorithm vi (3) Eksperimen III	120

Gambar 2.74 Eksperimen Video Player (1)	129
Gambar 2.75 Eksperimen Video Player (2)	130
Gambar 2.76 Eksperimen Save and Load (1)	135
Gambar 2.77 Eksperimen Save and Load (2)	136
Gambar 2.78 Eksperimen Save and Load (3)	136

BAB I

Deskripsi Persoalan

Tugas Besar 1 IF3070 Dasar Inteligensi Artifisial menerapkan algoritma local search untuk mencari solusi terbaik dari permasalahan *Diagonal Magic Cube* berukuran $5 \times 5 \times 5$. *Diagonal Magic Cube* adalah kubus yang terdiri dari angka-angka mulai dari 1 hingga n^3 tanpa pengulangan dengan n adalah panjang sisi dari kubus tersebut. Angka-angka ini disusun sehingga memenuhi beberapa aturan yaitu terdapat adanya satu angka yang disebut *magic number* yang tidak harus berada dalam rentang angka 1 hingga n^3 dan tidak termasuk dalam angka yang dimasukkan ke dalam kubus. Selain itu, jumlah angka pada setiap baris, kolom, dan tiang dalam kubus, serta semua diagonal di dalam ruang kubus harus sama dengan *magic number*. Dalam mencari solusi dari permasalahan *Diagonal Magic Cube*, dibutuhkan penggunaan algoritma pencarian yaitu algoritma *local search* yang berfungsi untuk menemukan solusi optimal melalui proses iteratif. *Diagonal Magic Cube* yang berukuran $5 \times 5 \times 5$ akan dimulai dari kondisi awal (*initial state*) berupa susunan acak angka 1 hingga 5^3 .

Untuk menyelesaikan permasalahan ini, akan digunakan algoritma *local search*: *hill-climbing*, *simulated annealing*, dan *genetic algorithm*, yang masing-masing akan dimulai dari kondisi awal berupa susunan angka acak dalam kubus. Setiap algoritma dirancang untuk mengoptimalkan susunan angka melalui serangkaian langkah iteratif yang bertujuan mendekati solusi yang memenuhi kriteria *Diagonal Magic Cube*. Dalam tugas ini, selain mengimplementasikan ketiga algoritma tersebut, dilakukan juga pemilihan dan perancangan fungsi tujuan (*objective function*) yang tepat, yaitu fungsi yang dapat mengukur seberapa baik suatu susunan angka mendekati konfigurasi *Diagonal Magic Cube*. Selain itu, dilakukan analisis perbandingan antar algoritma berdasarkan beberapa aspek, termasuk jumlah iterasi yang diperlukan, waktu komputasi yang dibutuhkan, dan nilai *cost* atau *fitness* yang dicapai di akhir proses. Evaluasi ini penting untuk memahami efektivitas dan efisiensi masing-masing algoritma dalam konteks masalah *Diagonal Magic Cube*, serta untuk menentukan pendekatan terbaik dalam mencapai solusi optimal dengan sumber daya yang tersedia.

BAB II

Hasil dan Pembahasan

A. Objective Function

Objective function adalah fungsi penting yang digunakan untuk mengevaluasi seberapa baik suatu solusi berhasil memenuhi tujuan yang telah ditentukan. Dalam konteks *magic cube*, *objective function* bertujuan untuk mengukur ketepatan susunan angka di dalam kubus sehingga mencapai status ideal dengan seluruh parameter yang dijumlahkan dapat menghasilkan *magic number*. Terdapat lima parameter yang perlu mencapai nilai *magic number* tersebut yaitu setiap baris, kolom, tiang, diagonal ruang, dan diagonal sisi dari sembilan potong bidang.

Berdasarkan sumber yang diambil dari laman MathWorld, *magic number* dari sebuah *magic cube* dapat diperoleh dengan:

$$M_3(n) = \frac{n(n^3+1)}{2}$$

dengan n sebagai panjang sisi dari *magic cube*.

Sesuai persoalan pada tugas besar ini, *magic cube* memiliki ukuran 5x5x5. Dengan rumus *magic number* di atas, maka diperoleh *magic number* sebesar:

$$M_3(5) = \frac{5(5^3+1)}{2} = 315$$

Oleh karena itu, jumlah elemen pada setiap baris (*row*), kolom (*column*), tiang (*pillar*), diagonal ruang, dan diagonal sisi harus berjumlah 315. Selanjutnya, berdasarkan penjelasan mengenai *objective function* yang telah diuraikan sebelumnya, rumus matematis dapat diturunkan sebagai acuan dalam melakukan perbandingan algoritma. Rumus matematis yang diperoleh adalah sebagai berikut:

- Untuk setiap baris r akan dihitung jumlah elemen di baris tersebut dan membandingkannya dengan *magic number* M :

$$Rr = \sum_{j=1}^n \sum_{k=1}^n C[r][j][k]$$

Keterangan:

Rr : Total jumlah elemen pada baris ke- r di sepanjang baris.

r : Indeks baris dalam kubus.

j : Indeks kolom dalam kubus.

k : Indeks tiang dalam kubus.

$C[r][j][k]$: Elemen dalam kubus pada posisi baris r , kolom j , dan tiang k .

- Untuk setiap kolom c akan dihitung jumlah elemen di kolom tersebut dan membandingkannya dengan *magic number* M:

$$Cc = \sum_{i=1}^n \sum_{k=1}^n C[i][c][k]$$

Keterangan:

Cc : Total jumlah elemen pada kolom ke- c di sepanjang kolom.

c : Indeks kolom dalam kubus.

i : Indeks baris dalam kubus.

k : Indeks tiang dalam kubus.

$C[i][c][k]$: Elemen dalam kubus pada posisi baris i , kolom c , dan tiang k .

- Untuk setiap tiang p akan dihitung jumlah elemen di tiang tersebut dan membandingkannya dengan *magic number* M:

$$Pp = \sum_{i=1}^n \sum_{j=1}^n C[i][j][p]$$

Keterangan:

Pp : Total jumlah elemen pada tiang ke- p di sepanjang tiang.

p : Indeks tiang dalam kubus.

i : Indeks baris dalam kubus.

j : Indeks kolom dalam kubus.

$C[i][j][p]$: Elemen dalam kubus pada posisi baris i , kolom j , dan tiang p .

- Untuk setiap diagonal c akan dihitung jumlah elemen di diagonal ruang tersebut dan membandingkannya dengan *magic number* M:

$$\circ D_1 = \sum_{i=1}^n C[i][i][i]$$

Keterangan:

- Indeks i berjalan dari 1 hingga n (ukuran kubus).

- Pada setiap layer ke- i , baris ke- i , dan kolom ke- i diambil elemen.

$$\circ D_2 = \sum_{i=1}^n C[i][n-i-1][i]$$

Keterangan:

- Indeks i berjalan dari 1 hingga n (ukuran kubus).

- Pada setiap layer ke- i , baris ke- i , dan kolom dari kanan ke kiri ($n - i - 1$) diambil elemen.

$$\circ D_3 = \sum_{i=1}^n C[i][i][n-i-1]$$

Keterangan:

- Indeks i berjalan dari 1 hingga n (ukuran kubus).

- Pada setiap layer ke- i , baris ke- i , dan kolom dari kanan ke kiri ($n - i - 1$) diambil elemen.
- $D_4 = \sum_{i=1}^n C[i][n-i-1][n-i-1]$
- Keterangan:
- Indeks i berjalan dari 1 hingga n (ukuran kubus).
 - Pada setiap layer ke- i , baris ke- i , dan kolom dari kanan ke kiri ($n - i - 1$) serta layer dari belakang ke depan ($n - i - 1$) diambil elemen.
- Untuk setiap diagonal c akan dihitung jumlah elemen di diagonal sisi tersebut dan membandingkannya dengan *magic number* M:
- $D_5 = \sum_{j=1}^n C[j][i][j]$
- Keterangan:
- Diagonal ini mengambil elemen $C[j][i][j]$ dengan i adalah layer (sumbu z) yang tetap.
 - Indeks j berjalan di sepanjang **baris** dan **kolom** yang sama pada bidang **xy**.
- $D_6 = \sum_{j=1}^n C[j][i][j]$
- Keterangan:
- Diagonal ini mengambil elemen $C[i][j][n-j-1]$ dengan layer i tetap.
 - Indeks j berjalan di sepanjang **baris** j dan **kolom** $n-j-i$ yang bergerak berlawanan arah di bidang **xy** (arah sebaliknya).
- $D_7 = \sum_{j=1}^n C[j][i][j]$
- Keterangan:
- Diagonal ini mengambil elemen $C[j][i][j]$ dengan i adalah posisi tetap pada **kolom**.
 - Indeks j berjalan di sepanjang **baris** dan **layer** yang sama.
- $D_8 = \sum_{j=1}^n C[n-j-1][i][j]$
- Keterangan:
- Diagonal ini mengambil elemen $C[n-j-i][i][j]$ dengan i adalah posisi tetap, tetapi **baris** berjalan mundur dari ke 0, sementara **layer** tetap berjalan maju. Hal ini menciptakan diagonal sebaliknya dalam bidang **yz**.
- $D_9 = \sum_{j=1}^n C[j][i][i]$
- Keterangan:
- Diagonal ini mengambil elemen $C[j][j][i]$ dengan i adalah posisi tetap pada **kolom**.

- Indeks j berjalan di sepanjang **baris** dan **layer** yang sama.
 - o
$$D_{10} = \sum_{j=1}^n C[j][n-j-1][i]$$
- Keterangan:
- Diagonal ini mengambil elemen $C[j][n-j-1][i]$ dengan kolom tetap tetapi **baris** berjalan maju sementara **layer** berjalan mundur dari n ke 0. Hal ini menciptakan diagonal sebaliknya dalam bidang **xz**.

Berdasarkan seluruh rumus di atas, didapatkan satu rumus *objective function* yaitu:

$$f = \sum_r^n |Rr - M| + \sum_c^n |Cc - M| + \sum_p^n |Pp - M| + \sum_{d=1}^{10} |Dd - M|$$

Keterangan:

- Rr : Jumlah elemen dalam **baris** tertentu.
- Cc : Jumlah elemen dalam **kolom** tertentu.
- Pp : Jumlah elemen dalam **tiang** tertentu.
- Dd : Jumlah elemen dalam **diagonal** tertentu.
- M : **Magic number** yang diharapkan dari penjumlahan elemen-elemen tersebut.

Dari persamaan di atas, terlihat bahwa *objective function* akan menghitung selisih antara jumlah elemen dalam berbagai parameter (kolom, baris, tiang, diagonal ruang, dan diagonal sisi) dan membandingkan hasilnya dengan nilai *magic number* yaitu 315 dalam konteks tugas besar ini. Tujuan utamanya adalah meminimalkan total penyimpangan dari *goal state*, idealnya hingga mencapai nol, yang menunjukkan bahwa *goal state* tersebut telah memenuhi karakteristik dari *magic cube* seperti yang telah dijelaskan sebelumnya.

Dengan menggunakan *objective function* dari persamaan-persamaan yang ada, kami membuat algoritma untuk menghitung *objective function* seperti di bawah ini.

1. **calculate_magic_number**: Fungsi ini menghitung *magic number* untuk kubus berukuran N. *Magic number* adalah jumlah yang harus dicapai oleh setiap baris, kolom, tiang, diagonal ruang, dan diagonal sisi dalam kubus sehingga dapat memenuhi karakteristik *magic cube*.
2. **line_sum**: Fungsi ini menerima daftar indeks yang mewakili sebuah garis dalam kubus dan menghitung jumlah total nilai pada indeks tersebut. Hal ini berguna untuk menentukan seberapa dekat garis tersebut untuk mencapai *magic number*.
3. **diagonal_indices**: Fungsi ini bertujuan untuk menghasilkan indeks yang mewakili berbagai diagonal dalam kubus bergantung pada parameter *plane*. Jika bidang yang dipilih adalah ‘main’, maka fungsi akan memberikan indeks untuk diagonal di bidang atas.
4. **objective_function**: Fungsi ini menghitung cost berdasarkan deviasi dari *magic number* untuk setiap baris, kolom, tiang, diagonal ruang, dan diagonal sisi dalam kubus. Pada

tahap pertama, fungsi ini akan menghitung *magic number* untuk kubus dengan ukuran N . Selanjutnya, fungsi ini menghitung jumlah nilai pada setiap baris, kolom, dan tiang dalam kubus, lalu mengukur deviasi dari *magic number* dan kemudian ditambahkan ke *cost*. Setelah itu, fungsi *objective_function* ini juga menghitung deviasi untuk semua diagonal ruang serta diagonal sisi, lalu ditambahkan ke *cost*. *Cost* menunjukkan seberapa jauh konfigurasi kubus pada saat ini dari kondisi ideal *magic cube* yang semakin rendah *cost*-nya (mendekati nol), maka semakin solusi saat ini dengan kondisi ideal *magic cube*.

```

def calculate_magic_number(N: int) -> int:
    """Calculates magic number for cube of size N"""
    return (N * (N**3 + 1)) // 2

def line_sum(cube: List[List[List[int]]], indices: List[Tuple[int, int, int]]) -> int:
    """Calculates sum of values along given indices"""
    return sum(cube[i][j][k] for i, j, k in indices)

def diagonal_indices(N: int, plane: str, i: int = 0) -> List[Tuple[int, int, int]]:
    """Returns indices for different types of diagonals"""
    if plane == "main":
        diagonals = [
            [(j, j, i) for j in range(N)], # Top face
            [(j, N-1-j, i) for j in range(N)]
        ]
    elif plane == "face":
        diagonals = [
            [(i, j, j) for j in range(N)], # Front face
            [(i, j, N-1-j) for j in range(N)]
        ]
    elif plane == "side":
        diagonals = [
            [(j, i, j) for j in range(N)], # Side face
            [(j, i, N-1-j) for j in range(N)]
        ]
    else: # space
        diagonals = [
            [(i, i, i) for i in range(N)], # Main diagonal

```

```

        [(i, i, N-1-i) for i in range(N)], # Anti-diagonal
        [(i, N-1-i, i) for i in range(N)],
        [(N-1-i, i, i) for i in range(N)]
    ]
return diagonals

def objective_function(cube: List[List[List[int]]]) -> int:
    """Calculates total cost based on deviation from magic
    number"""
    N = len(cube)
    magic_number = calculate_magic_number(N)
    cost = 0

    # Line sums (rows, columns, pillars)
    for i in range(N):
        for j in range(N):
            row_indices = [(i, j, k) for k in range(N)]
            col_indices = [(i, k, j) for k in range(N)]
            pillar_indices = [(k, i, j) for k in range(N)]

            cost += abs(magic_number - line_sum(cube,
row_indices))
            cost += abs(magic_number - line_sum(cube,
col_indices))
            cost += abs(magic_number - line_sum(cube,
pillar_indices))

    # Main space diagonals
    for diagonal in diagonal_indices(N, "main"):
        cost += abs(magic_number - line_sum(cube, diagonal))

    # Plane diagonals
    for i in range(N):
        for plane in ["face", "side", "top"]:
            for diagonal in diagonal_indices(N, plane, i):
                cost += abs(magic_number - line_sum(cube,
diagonal))

    return cost

```

B. Utilities

Utilities atau *utils* merupakan sekumpulan fungsi pendukung dalam proses optimasi algoritma *local search* untuk menyelesaikan masalah *Diagonal Magic Cube*. Fungsi-fungsi ini digunakan secara luas oleh berbagai algoritma seperti *hill-climbing*, *simulated annealing*, dan *genetic algorithm* untuk mempercepat dan menyederhanakan operasi tertentu. Selain fungsi yang digunakan untuk menghitung *objective function* dan telah dijelaskan sebelumnya, terdapat satu fungsi yang terdapat pada *utilities* dan digunakan untuk menginisialisasi kubus.

1. **initialize_random_cube**: Fungsi ini menginisialisasi kubus tiga dimensi berukuran $N \times N \times N$ dengan angka unik secara acak, yang nantinya akan menjadi konfigurasi awal untuk algoritma *local search*. Proses ini memastikan bahwa setiap angka dalam kubus berbeda, yang menjadi dasar penting untuk eksperimen.

```
def initialize_random_cube(N: int) -> List[List[List[int]]]:  
    """Initializes a 3D cube with unique random integers"""  
    total_cells = N * N * N  
    used: Set[int] = set()  
    cube = [[[0 for _ in range(N)] for _ in range(N)] for _ in range(N)]  
  
    for i, j, k in [(i,j,k) for i in range(N) for j in range(N)  
    for k in range(N)]:  
        while (num := random.randint(1, total_cells)) in used:  
            continue  
        cube[i][j][k] = num  
        used.add(num)  
  
    return cube
```

C. Local Search

1. Steepest Ascent Hill-climbing

Dalam algoritma *Steepest Ascent Hill-climbing* akan dilakukan evaluasi terhadap semua tetangga (semua kemungkinan langkah dari solusi saat ini) dan memilih tetangga yang memberikan perbaikan terbaik. Penerapan *Steepest Ascent Hill-climbing* pada persoalan *Diagonal Magic Cube* 5x5x5 dimulai dengan menghasilkan susunan angka 1 hingga 5^3 secara acak sebagai solusi awal. Pada setiap iterasi, akan dilakukan evaluasi terhadap semua tetangga dengan *objective function* agar menentukan sejauh mana susunan angka tersebut mendekati syarat *Diagonal Magic Cube* yaitu jumlah angka pada setiap baris, kolom, dan tiang dalam kubus, serta semua diagonal di dalam ruang kubus harus sama dengan *magic number* yang pada persoalan ini adalah 315.

Langkah berikutnya adalah menukar dua angka di dalam *Diagonal Magic Cube* secara acak untuk menghasilkan semua tetangga dari solusi saat ini. Selanjutnya, dilakukan evaluasi kembali terhadap *objective function* untuk menentukan apakah terjadi peningkatan atau penurunan dalam mendekati solusi optimal. Tetangga yang memberikan perbaikan terbaik akan dipilih menjadi *current state*. Tahap ini akan terus diulang sampai algoritma tidak menemukan lagi tetangga yang dapat memperbaiki nilai lebih lanjut. Algoritma akan berhenti dan menghasilkan susunan angka dari 1 hingga 5^3 pada *Diagonal Magic Cube*.

Dengan kemampuan *Steepest Ascent Hill-climbing* yang hanya dapat bergerak ke tetangga yang membuat nilai *objective function* lebih baik, maka algoritma ini kemungkinan besar akan mudah terjebak pada *local optima*, terutama jika hasil *random generate* pada tahap awal memiliki susunan angka yang buruk. Tak hanya itu, walaupun *Steepest Ascent Hill-climbing* dapat menemukan solusi *local optima* dengan cepat, namun ruang untuk eksplorasi solusi dari *Diagonal Magic Cube* akan terbatas dikarenakan mudah terjebak pada *local optima*.

```
import copy
import random
import time
from typing import List, Tuple
from . import utils

#fungsi untuk swap
def swap(cube: List[List[List[int]]], posisi1: Tuple[int, int, int], posisi2: Tuple[int, int, int]) -> None:
    i1, j1, k1 = posisi1
    i2, j2, k2 = posisi2
```

```

        cube[i1][j1][k1], cube[i2][j2][k2] = cube[i2][j2][k2],
cube[i1][j1][k1]

# Algoritma Steepest Ascent Hill Climbing
def steepest_ascent_algorithm(cube):
    N = len(cube)
    start_time = time.time()
    current_cost = utils.objective_function(cube)
    best_cost = current_cost
    costs = []
    states = []

    #membuat daftar semua pasangan
    all_positions = [(i//(N*N), (i//N)%N, i%N) for i in
range(N*N*N)]
    all_pairs = []
    for i in range(len(all_positions)):
        for j in range(i+1, len(all_positions)):
            all_pairs.append((all_positions[i], all_positions[j]))

    while True:
        found_improvement = False
        random.shuffle(all_pairs)

        for pos1, pos2 in all_pairs:
            # coba/try swap
            swap(cube, pos1, pos2)
            new_cost = utils.objective_function(cube)

            if new_cost < best_cost:
                best_cost = new_cost
                costs.append(best_cost)
                states.append(copy.deepcopy(cube))
                found_improvement = True
                break
            else:
                swap(cube, pos1, pos2) # mengembalikan/undo swap

    #kalau tidak ada improvement dari cost nya

```

```

        if not found_improvement:
            break

        duration = time.time() - start_time
        #return hasil
        return {
            "final_cube": cube,
            "final_cost": best_cost,
            "average_cost": round(best_cost/109, 4),
            "duration": round(duration, 2),
            "iteration": len(costs),
            "costs": costs,
            "states": states
        }
    }
}

```

2. Hill-climbing with Sideways Move

Hill-climbing with Sideways Move adalah variasi dari algoritma *hill climbing*. Namun, algoritma ini berbeda dengan algoritma *hill climbing* konvensional yang hanya bergerak ke tetangga yang lebih baik (nilai *objective function* yang dihasilkan lebih rendah). Algoritma *Hill-climbing with Sideways Move* memungkinkan untuk tetap bergerak ke tetangga yang menghasilkan nilai *objective function* yang sama. Tujuan *sideways move* adalah untuk membantu algoritma melewati *local optima* dan menemukan solusi *global optima*.

Penerapan algoritma *Hill-climbing with Sideways Move* dalam mencari solusi dari *Diagonal Magic Cube* dimulai dengan *initial state* yang membentuk konfigurasi awal kubus. Pada tahap inisialisasi, algoritma ini melakukan pengecekan apakah susunan awal telah memenuhi syarat *Diagonal Magic Cube* yaitu jumlah seluruh angka pada setiap baris, kolom, tiang, diagonal ruang, dan diagonal sisi memenuhi nilai *magic number*. Jika konfigurasi awal kubus sudah memenuhi syarat, maka algoritma akan berhenti. Jika belum, konfigurasi kubus ini akan diambil sebagai *current state*.

Pada setiap iterasi, algoritma ini akan memilih tetangga secara acak dari *current state* dengan cara menukar dua angka di dalam kubus. Setelah memilih tetangga secara acak, algoritma akan melakukan evaluasi apakah tetangga yang dipilih memberikan perbaikan atau nilai yang sama pada *objective function* dibandingkan dengan *current state*. Jika tetangga yang dipilih menghasilkan perbaikan atau nilai yang sama, maka algoritma akan berpindah ke tetangga tersebut dan diambil sebagai *current state* yang baru. Jika tidak ada perbaikan, algoritma akan memilih tetangga acak lain dan mengulangi proses hingga terdapat perbaikan atau sampai batas maksimal *sideways move* yaitu 10.

Algoritma akan berhenti jika nilai *objective function* sudah tercapai atau jika jumlah *sideways move* telah mencapai batas maksimal yang ditentukan. Setelah berhenti, algoritma akan mengembalikan konfigurasi kubus (*cube*), *best_cost*, dan waktu eksekusi (*duration*).

```
import copy
import time
import random
from typing import Tuple, List
from . import utils

def swap_elements(cube: List[List[List[int]]], pos1: Tuple[int, int, int], pos2: Tuple[int, int, int]) -> None:
    """Swaps elements in the cube at the given positions."""
    i, j, k = pos1
    l, m, n = pos2
    cube[i][j][k], cube[l][m][n] = cube[l][m][n], cube[i][j][k]

def find_best_neighbor(cube: List[List[List[int]]], N: int, all_pairs: List[Tuple[Tuple[int, int, int], Tuple[int, int, int]]],
                      best_cost: int, max_sideways: int, sideways: int) -> Tuple[int, Tuple[int, int, int, int, int, int]]:
    """Finds best neighbor using random pair selection."""
    best_neighbor_cost = best_cost
    best_swap = (-1, -1, -1, -1, -1, -1) #Default untuk swap yang invalid

    # Shuffle pasangan untuk random search
    random.shuffle(all_pairs)

    for pos1, pos2 in all_pairs:
        # Swap tiap elemen dan cek cost
        swap_elements(cube, pos1, pos2)
        new_cost = utils.objective_function(cube)

        # Update jika new cost lebih baik atau bisa sideways move
        if new_cost < best_neighbor_cost or (new_cost ==
```

```

best_neighbor_cost and sideways < max_sideways):
    best_neighbor_cost = new_cost
    i, j, k = pos1
    l, m, n = pos2
    best_swap = (i, j, k, l, m, n)

    # Membalikkan swap ke yang semula
    swap_elements(cube, pos1, pos2)

    # Break jika ditemukan cost yang lebih baik
    if new_cost < best_neighbor_cost:
        break

return best_neighbor_cost, best_swap

def sideways_move_algorithm(cube: List[List[List[int]]],
max_sideways: int = 10) -> dict:
    """Executes a hill climbing approach that allows sideways
moves and uses random selection for neighbors."""
    N = len(cube)
    current_cost = utils.objective_function(cube)
    best_cost = current_cost
    costs = []
    states = []
    iteration = 0
    sideways = 0

    # Mulai timer
    start_time = time.time()

    # Generate seluruh pasangan yang mungkin
    all_positions = [(i//(N*N), (i//N)%N, i%N) for i in
range(N*N*N)]
    all_pairs = []
    for i in range(len(all_positions)):
        for j in range(i+1, len(all_positions)):
            all_pairs.append((all_positions[i], all_positions[j]))

    while True:

```

```

# Cari tetangga yang lebih baik
best_neighbor_cost, best_swap = find_best_neighbor(cube,
N, all_pairs, best_cost, max_sideways, sideways)

# Stop jika tidak menemukan tetangga yang lebih baik
if best_swap == (-1, -1, -1, -1, -1, -1):
    break

# Perform swap yang terbaik
i, j, k, l, m, n = best_swap
swap_elements(cube, (i, j, k), (l, m, n))

# Update cost dan check sideways move
if best_neighbor_cost < best_cost:
    best_cost = best_neighbor_cost
    sideways = 0
elif best_neighbor_cost == best_cost:
    sideways += 1
    if sideways >= max_sideways:
        break

costs.append(best_cost)
states.append(copy.deepcopy(cube))
iteration += 1

# Menghitung total durasi
duration = time.time() - start_time

return {
    "final_cube": cube,
    "final_cost": best_cost,
    "average_cost": round(best_cost/109, 4),
    "duration": round(duration, 2),
    "iteration": len(costs),
    "costs": costs,
    "states": states
}

```

3. Stochastic Hill-climbing

Stochastic Hill-climbing adalah algoritma yang tidak mengevaluasi semua tetangga untuk memutuskan tetangga mana yang akan dipilih selanjutnya. Algoritma ini hanya memilih satu tetangga secara acak dan nantinya akan memutuskan apakah akan bergerak ke tetangga tersebut berdasarkan perbaikan nilai *objective function* atau melanjutkan ke tetangga lain.

Penerapan algoritma ini pada masalah *Diagonal Magic Cube* dimulai dengan *initial state* yaitu susunan angka 1 hingga 5^3 secara acak. Jika susunan angka di awal sudah memenuhi syarat *Diagonal Magic Cube* dengan jumlah angka pada setiap baris, kolom, tiang, dan diagonal ruang serta diagonal bidang sesuai dengan nilai *magic number*, maka algoritma akan berhenti. Namun, jika tidak maka susunan angka di awal tersebut akan dibuat sebagai *current state*.

Pada setiap iterasi, algoritma *Stochastic Hill-climbing* akan memilih salah satu tetangga dari *current state* secara acak yang diperoleh dengan menukar dua angka di dalam kubus. Lalu, akan dilakukan evaluasi apakah tetangga yang dipilih dapat memberikan perbaikan nilai *objective function* dibandingkan *current state*. Jika ada perbaikan, maka tetangga tersebut akan diambil sebagai *current state*, sedangkan jika tidak maka algoritma akan memilih tetangga lain secara acak dan mengulangi prosesnya sampai tidak ada kondisi perbaikan lebih lanjut.

Pada algoritma *Stochastic Hill Climbing*, jika telah mencapai nilai *objective function* terbaik atau jumlah iterasi telah mencapai batas maksimal yang ditentukan, maka algoritma akan berhenti. Fungsi ini kemudian akan mengembalikan beberapa hasil utama, yaitu *cube*, *best_cost*, *costs*, dan *duration*.

Penggunaan algoritma ini dalam penyelesaian masalah *Diagonal Magic Cube* dapat menyebabkan solusi yang tidak optimal. Hal ini dikarenakan algoritma ini hanya mencari tetangga secara acak dan memilihnya jika membuat nilai *objective function* mendekati solusi. Walaupun algoritma ini akan cepat, efektif dan sederhana saat digunakan untuk menyelesaikan permasalahan *Diagonal Magic Cube*, namun dengan pengecekan tetangga yang acak, algoritma ini rentan terjebak dalam *local optima*.

```
import copy
import time
import random
from typing import List, Tuple
from . import utils

# Fungsi untuk memilih posisi random di cube
def select_random_position(N: int) -> int:
    return random.randint(0, N - 1)
```

```

# Fungsi untuk menghitung cost dari hasil suatu swap di cube
def calculate_cost_change(cube: List[List[List[int]]],
                           pos1: Tuple[int, int, int],
                           pos2: Tuple[int, int, int]) -> int:
    original_cost = utils.objective_function(cube)
    # Melakukan swap
    i1, j1, k1 = pos1
    i2, j2, k2 = pos2
    cube[i1][j1][k1], cube[i2][j2][k2] = cube[i2][j2][k2],
    cube[i1][j1][k1]
    # Menghitung cost hasil swap cube
    new_cost = utils.objective_function(cube)
    # Melakukan undo swap
    cube[i1][j1][k1], cube[i2][j2][k2] = cube[i2][j2][k2],
    cube[i1][j1][k1]
    return new_cost - original_cost

# Algoritma Stochastic Algorithm
def stochastic_algorithm(cube: List[List[List[int]]]) -> dict:
    N = len(cube)
    current_cost = utils.objective_function(cube)
    best_cost = current_cost
    costs = []
    states = []
    max_iteration = 10000
    iteration = 0

    start_time = time.time()

    while iteration < max_iteration:
        # Menghasilkan indeks random dari suatu posisi cube
        i1, j1, k1 = random.randint(0, N-1), random.randint(0,
N-1), random.randint(0, N-1)
        i2, j2, k2 = random.randint(0, N-1), random.randint(0,
N-1), random.randint(0, N-1)

        # Memastikan posisi 1 dan 2 bukan merupakan posisi yang
        sama

```

```

        while (i1, j1, k1) == (i2, j2, k2):
            i2, j2, k2 = random.randint(0, N-1), random.randint(0,
N-1), random.randint(0, N-1)

            # Menghitung cost dari cube awal
            original_cost = utils.objective_function(cube)

            # Melakukan swap
            cube[i1][j1][k1], cube[i2][j2][k2] = cube[i2][j2][k2],
cube[i1][j1][k1]
            new_cost = utils.objective_function(cube)
            cost_change = new_cost - original_cost

            if cost_change < 0:
                current_cost = new_cost
                best_cost = min(best_cost, current_cost)
            else:
                # Jika tidak terdapat perubahan cost ke arah yang
                lebih baik, maka dilakukan undo swap
                cube[i1][j1][k1], cube[i2][j2][k2] = cube[i2][j2][k2],
cube[i1][j1][k1]

                costs.append(current_cost)
                states.append(copy.deepcopy(cube))
                iteration += 1

duration = time.time() - start_time

return {
    "final_cube": cube,
    "final_cost": best_cost,
    "average_cost": round(best_cost/109, 4),
    "duration": round(duration, 2),
    "iteration": len(costs),
    "costs": costs,
    "states": states
}

```

4. Random Restart Hill-climbing

Random Restart Hill-climbing adalah salah satu varian dari algoritma *hill-climbing* yang menghindari terjebaknya algoritma dalam local optima. Berbeda dengan *Stochastic Hill-climbing* yang hanya memilih tetangga secara acak, *Random Restart Hill-climbing* menggabungkan pendekatan *hill-climbing* biasa dengan cara *me-restart* untuk memberikan peluang pencarian solusi yang lebih baik.

Penerapannya dimulai dengan pemilihan *initial state* secara acak, kemudian algoritma *hill-climbing* diterapkan untuk mencapai *local optima*. Jika *local optima* tercapai dan algoritma tidak dapat menemukan perbaikan dalam nilai *objective function*, maka algoritma akan melakukan *restart* dengan memilih *initial state* baru secara acak dan mengulangi proses *hill-climbing* dari awal. Proses ini akan terus diulangi hingga solusi terbaik ditemukan atau hingga batas jumlah *restart* yang ditentukan tercapai.

Contoh penerapan *Random Restart Hill-climbing* pada masalah *Diagonal Magic Cube* dimulai dengan *initial state* berupa susunan angka 1 hingga 5^3 secara acak. Jika *initial state* sudah memenuhi syarat *Diagonal Magic Cube*, di mana jumlah angka pada setiap baris, kolom, tiang, dan diagonal bidang serta ruang sesuai dengan nilai *magic number*, maka algoritma berhenti. Jika tidak, algoritma akan mengatur susunan tersebut sebagai *current state* dan menjalankan proses pencarian solusi.

Pada setiap iterasi dalam proses *hill-climbing*, algoritma memilih tetangga yang diperoleh dengan menukar dua angka di dalam kubus. Tetangga dievaluasi untuk melihat apakah memberikan perbaikan pada nilai *objective function* dibandingkan dengan *current state*. Jika terjadi perbaikan, tetangga tersebut akan dianggap sebagai *current state*. Ketika tidak ada tetangga yang memperbaiki nilai *objective function* dan *local optima* tercapai, algoritma akan melakukan *restart*.

```
import copy
import time
import random
from typing import List, Tuple
from . import utils

def swap_elements(cube: List[List[List[int]]], pos1: Tuple[int, int, int], pos2: Tuple[int, int, int]) -> None:
    # Menukar elemen
    i, j, k = pos1
    l, m, n = pos2
    cube[i][j][k], cube[l][m][n] = cube[l][m][n], cube[i][j][k]
```

```

def generate_random_neighbor(cube: List[List[List[int]]]) ->
List[List[List[int]]]:
    # Menghasilkan anak secara acak dengan menukar 2 elemen
    N = len(cube)
    i1, j1, k1 = random.randint(0, N-1), random.randint(0, N-1),
random.randint(0, N-1)
    i2, j2, k2 = random.randint(0, N-1), random.randint(0, N-1),
random.randint(0, N-1)

    # Untuk memastikan pertukarannya gasama
    while (i1, j1, k1) == (i2, j2, k2):
        i2, j2, k2 = random.randint(0, N-1), random.randint(0,
N-1), random.randint(0, N-1)

    # Salin kubus untuk tetangga baru
    new_cube = [[[cube[i][j][k] for k in range(N)] for j in
range(N)] for i in range(N)]

    # Tukar elemen
    swap_elements(new_cube, (i1, j1, k1), (i2, j2, k2))

    return new_cube

def random_restart_algorithm(cube: List[List[List[int]]],
max_restart: int = 10, max_iterations: int = 100) -> dict:
    """
    Melakukan algoritma random restart untuk meminimalkan cost
    kubus dengan mempertahankan variabel asli.

    Setiap restart memiliki batas maksimal iterasi, dan best_cube
    dibandingkan antar-restart.
    """
    N = len(cube)
    current_cube = utils.initialize_random_cube(N)
    current_cost = utils.objective_function(current_cube)
    cube = current_cube
    best_cost = current_cost
    restart = 0
    total_iteration = 0
    costs = []

```

```

states = []
iteration_restart = []

start_time = time.time()

while restart < max_restart:
    iteration = 0
    while iteration < max_iterations:
        # Melakukan generate random neighbor
        neighbor_cube = generate_random_neighbor(current_cube)
        neighbor_cost =
utils.objective_function(neighbor_cube)

        if neighbor_cost >= current_cost:
            iteration += 1
            costs.append(current_cost)
            states.append(copy.deepcopy(current_cube))
            # Jika tetangga lebih buruk atau sama,
keluar dari pencarian
            break
        else:
            # Jika tetangga lebih baik, update current_cube
            current_cube = neighbor_cube
            current_cost = neighbor_cost
            iteration += 1
            costs.append(current_cost)
            states.append(copy.deepcopy(current_cube))

    # Bandingkan best_cube di setiap restart setelah restart
pertama
    if restart == 0:
        cube = current_cube
        best_cost = current_cost
    else:
        # Jika solusi restart saat ini lebih baik, update
best_cube
        if current_cost < best_cost:
            cube = current_cube
            best_cost = current_cost

```

```

iteration_restart.append(iteration)
restart += 1

# Restart pencarian dengan solusi acak yang baru
current_cube = utils.initialize_random_cube(N)
current_cost = utils.objective_function(current_cube)

duration = time.time() - start_time

return {
    "final_cube": cube,
    "final_cost": best_cost,
    "average_cost": round(best_cost/109, 4),
    "duration": round(duration, 2),
    "iteration": len(costs),
    "iteration_restart": iteration_restart,
    "restart": restart,
    "costs": costs,
    "states": states
}

```

5. Simulated Annealing

Sesuai dengan namanya, algoritma *Simulated Annealing* diambil dari proses *annealing* dalam metalurgi. *Annealing* adalah proses dari sebuah material yang dipanaskan dan kemudian didinginkan perlahan untuk menghilangkan ketidak sempurnaan. Dengan fleksibilitas yang dimiliki oleh algoritma ini, *Simulated Annealing* dapat digunakan baik untuk permasalahan optimasi kontinu dan optimasi diskrit.

Dalam penerapan algoritma ini untuk menyelesaikan permasalahan *Diagonal Magic Cube*, akan dimulai dengan *initial state* berupa penyusunan angka 1 hingga 5^3 secara acak. Selanjutnya, pada setiap tahap akan dilakukan pengecekan tetangga dengan mengevaluasi nilai *objective function*. Jika tetangga tersebut memberikan solusi yang lebih baik, maka akan dipilih sebagai solusi terbaru. Namun jika tidak, tetangga tersebut masih dapat dipilih dengan probabilitas yang bergantung pada temperatur dan perbedaan nilai *objective function* antara solusi saat ini dengan solusi yang terbaru. Rumus probabilitas untuk penerimaan solusi tersebut adalah:

$$e^{\Delta E/T}$$

Dengan:

- ΔE sebagai perbedaan nilai *objective function* dengan solusi baru dan solusi lama.
- T sebagai suhu pada saat itu.

Pada *Diagonal Magic Cube*, suhu awal akan ditetapkan tinggi untuk memungkinkan eksplorasi ruang solusi yang lebih luas. Seiring dengan berjalananya tiap iterasi, suhu akan diturunkan secara perlahan sesuai dengan *cooling schedule* yang sudah ditentukan. Saat suhu telah mencapai level rendah, maka algoritma akan berhenti dikarenakan tidak ada lagi perubahan signifikan yang dapat dilakukan pada solusi permasalahan.

Dengan kemampuan dari *Simulated Annealing* yang dapat bergerak ke tetangga yang menghasilkan solusi lebih buruk, algoritma ini dapat terhindar dari *local optima*. Tak hanya itu, algoritma ini juga memiliki ruang eksplorasi yang luas dengan parameter yang dapat diatur yaitu suhu sehingga cocok untuk permasalahan Diagonal Magic Cube yang kompleks. Namun, algoritma ini membutuhkan beberapa parameter yang harus diatur dengan baik di awal proses yaitu temperatur dan waktu atau jadwal pendinginan (*cooling schedule*).

```
import copy
import time
import math
import random
from typing import List, Tuple
from . import utils

# Fungsi untuk melakukan swap antara 2 elemen di cube dengan
# posisi tertentu
def swap_elements(cube: List[List[List[int]]], pos1: Tuple[int,
int, int], pos2: Tuple[int, int, int]) -> None:
    i1, j1, k1 = pos1
    i2, j2, k2 = pos2
    cube[i1][j1][k1], cube[i2][j2][k2] = cube[i2][j2][k2],
cube[i1][j1][k1]

# Fungsi untuk menghasilkan dua posisi random di cube
def get_random_neighbor(N: int) -> Tuple[Tuple[int, int, int],
Tuple[int, int, int]]:
    pos1 = (random.randint(0, N-1), random.randint(0, N-1),
random.randint(0, N-1))
    pos2 = (random.randint(0, N-1), random.randint(0, N-1),
```

```

random.randint(0, N-1))

    # Memastikan bahwa posisi yang dihasilkan bukan merupakan
    posisi yang sama
    while pos1 == pos2:
        pos2 = (random.randint(0, N-1), random.randint(0, N-1),
random.randint(0, N-1))
    return pos1, pos2

# Fungsi untuk menentukan apakah solusi dapat diterima berdasarkan
probabilitas
def acceptance_function(delta_cost: float, temperature: float) ->
bool:
    if delta_cost > 0:
        return True # Menerima solusi baru yang lebih baik
    else:
        r = random.random()
        # Menerima solusi baru yang lebih buruk berdasarkan
probabilitas
        return r < math.exp(delta_cost / temperature)

# Algoritma Simulated Annealing
def simulated_annealing_algorithm(cube: List[List[List[int]]],
T_max: float = 100.0, T_min: float = 0.1,
                                         E_threshold: float = 0.01,
cooling_rate: float = 0.9993,
                                         max_no_improvement: int = 1000,
max_iteration: int = 10000):
    """Performs the simulated annealing algorithm to optimize the
cube configuration."""
    current_cost = utils.objective_function(cube) # Initial cost
    best_cost = current_cost # Menetapkan current_cost menjadi
best_cost saat ini
    temperature = T_max # Mengatur suhu menjadi suhu maksimum
    iteration = 0
    no_improvement = 0
    local_optima = 0

    costs = []
    exps = []

```

```

states = []

# Memulai timer
start_time = time.time()

while temperature > T_min and current_cost > E_threshold and iteration < max_iteration:
    # Menghasilkan posisi random
    pos1, pos2 = get_random_neighbor(len(cube))
    # Melakukan swap
    swap_elements(cube, pos1, pos2)
    # Menghitung cost dari hasil swap
    new_cost = utils.objective_function(cube)
    delta_cost = current_cost - new_cost

    # Memeriksa apakah solusi sekarang dapat diterima
    if acceptance_function(delta_cost, temperature):
        if delta_cost < 0:
            local_optima += 1 # Menambah jumlah local_optima
apabila solusi yang lebih buruk dapat diterima
            current_cost = new_cost
            if new_cost < best_cost:
                best_cost = new_cost
                no_improvement = 0
            else:
                no_improvement += 1
        else:
            swap_elements(cube, pos1, pos2) # Melakukan undo swap
jika solusi tidak dapat diterima
            no_improvement += 1

    # Keluar dari perulangan while apabila sudah mencapai
max_no_improvement
    if no_improvement >= max_no_improvement:
        break

temperature *= cooling_rate # Menurunkan suhu
costs.append(current_cost)
states.append(copy.deepcopy(cube))

```

```

        # Mencatat probabilitas penerimaan solusi setiap 200
iterasi
        if iteration % 200 == 0:
            exp_delta_E_T = 1 if delta_cost > 0 else
math.exp(delta_cost / temperature)
            exps.append(exp_delta_E_T)

        iteration += 1

# Menghitung durasi yang diperlukan algoritma
duration = time.time() - start_time

return {
    "final_cube": cube,
    "final_cost": best_cost,
    "average_cost": round(best_cost/109, 4),
    "duration": round(duration, 2),
    "iteration": len(costs),
    "local_optima": local_optima,
    "costs": costs,
    "exps": exps,
    "states": states
}

```

6. Genetic Algorithm

Genetic Algorithm adalah algoritma yang menggunakan prinsip evolusi yang diambil dari proses seleksi alam dalam biologi. Algoritma ini memanfaatkan konsep-konsep evolusi biologis, seperti reproduksi, mutasi, seleksi, dan crossover atau persilangan. *Genetic Algorithm* sering diterapkan untuk menyelesaikan masalah optimasi, terutama saat ruang eksplorasi solusi yang sangat besar.

Dalam pemecahan masalah *Diagonal Magic Number*, dimulai dengan tahap *initial state* berupa penyusunan angka 1 hingga 5^3 secara acak. Selanjutnya, akan dilakukan evaluasi pada setiap angka yang sudah disusun pada setiap kubus di *Diagonal Magic Cube* dengan *fitness score* yang mengukur seberapa dekat susunan angka dalam kubus memenuhi kondisi *magic number* pada setiap baris, kolom, tiang, dan diagonal ruang serta bidang. Lalu akan dilakukan seleksi untuk memilih individu terbaik yang akan dijadikan *parents*. Setelah dilakukan pemilihan, akan dilakukan crossover atau persilangan yaitu penukaran sebagian angka untuk menghasilkan keturunan atau

solusi baru. Setelah *crossover*, beberapa individu baru akan mengalami mutasi untuk menjaga keragaman genetik dalam populasi. Mutasi ini dilakukan dengan menukar dua angka secara acak. Hal ini penting untuk dilakukan agar algoritma tidak hanya berfokus dengan solusi yang sudah ada tetapi dapat menemukan solusi baru. Setelahnya, individu-individu baru akan membentuk generasi baru dan seluruh proses tersebut akan diulang pada generasi berikutnya. Algoritma akan terus berlanjut hingga menemukan solusi dari permasalahan *Diagonal Magic Cube* atau mencapai batas jumlah generasi yang sudah ditentukan.

Dengan kemampuan *Genetic Algorithm* yang dapat menghasilkan individu baru melalui proses mutasi dan variasi dalam populasi, maka *local optima* dapat hindari. Tak hanya itu, dengan proses *crossover* dan mutasi, algoritma dapat menghasilkan banyak kombinasi solusi baru. Namun, kompleksitas dari penggunaan algoritma ini termasuk tinggi dikarenakan terdapat beberapa parameter yang harus diatur seperti ukuran populasi dan laju mutasi serta dengan populasi yang besar akan memakan waktu yang lama dibandingkan dengan algoritma lainnya.

```
import copy
import time
import random
from typing import List, Tuple, Dict
from copy import deepcopy
from . import utils

# Fungsi crossover melakukan pertukaran gen antar dua parent untuk
# menghasilkan child baru
def crossover(parent1: List[List[List[int]]], parent2:
List[List[List[int]]], N: int) -> List[List[List[int]]]:
    child = deepcopy(parent1)
    for i in range(N):
        for j in range(N):
            for k in range(N):
                if random.random() < 0.5:
                    child[i][j][k] = parent2[i][j][k]
    return child

# Fungsi mutate mengubah beberapa elemen pada cube secara acak
# berdasarkan tingkat mutasi
def mutate(cube: List[List[List[int]]], N: int, mutation_rate:
float) -> List[List[List[int]]]:
```

```

        mutated = deepcopy(cube)
        for i in range(N):
            for j in range(N):
                for k in range(N):
                    if random.random() < mutation_rate:
                        mutated[i][j][k] = random.randint(1, N * N *
N)
        return mutated

def evaluate_population(population: List[List[List[int]]]) ->
List[float]:
    return [utils.objective_function(individual) for individual in
population]

# Fungsi tournament_selection memilih sejumlah parent berdasarkan
biaya (cost)
def tournament_selection(population: List[List[List[int]]], costs:
List[float], tournament_size: int) -> List[List[List[int]]]:
    selected = []
    for _ in range(len(population)):
        tournament = random.sample(list(zip(population, costs)),
k=tournament_size)
        winner = min(tournament, key=lambda x: x[1])
        selected.append(deepcopy(winner[0]))
    return selected

# Fungsi genetic_algorithm adalah algoritma genetik untuk
mengoptimalkan cube
def genetic_algorithm(cube: List[List[List[int]]], crossover_rate:
float = 0.8, initial_mutation_rate: float = 0.05, elitism_count:
int = 5, tournament_size: int = 5) -> Dict:
    N = len(cube)
    population_size = 300
    max_iteration = 500
    population = [utils.initialize_random_cube(N) for _ in
range(population_size)]
    costs = []
    states = []
    best_cost = float('inf')

```

```

best_cube = None

start_time = time.time()

for iteration in range(max_iteration):
    # Evaluate population
    population_costs = evaluate_population(population)
    fitness_scores = list(zip(population_costs, population))
    fitness_scores.sort(key=lambda x: x[0])

    current_fitness_values = [score[0] for score in
fitness_scores]
    avg_fitness = sum(current_fitness_values) /
len(current_fitness_values)

    costs.append(avg_fitness)
    states.append(deepcopy(fitness_scores[0][1]))

    if fitness_scores[0][0] < best_cost:
        best_cost = fitness_scores[0][0]
        best_cube = deepcopy(fitness_scores[0][1])

    # Elitism: preserve top elitism_count individuals
    elites = [deepcopy(ind) for _, ind in
fitness_scores[:elitism_count]]

    # Tournament selection
    selected_population = tournament_selection(population,
population_costs, tournament_size)

    # Crossover and mutation
    next_population = []
    mutation_rate = initial_mutation_rate * (1 - iteration /
max_iteration) # Adaptive mutation rate
    while len(next_population) < population_size -
elitism_count:
        parent1 = random.choice(selected_population)
        parent2 = random.choice(selected_population)
        if random.random() < crossover_rate:

```

```

        child = crossover(parent1, parent2, N)
    else:
        child = deepcopy(parent1)
        child = mutate(child, N, mutation_rate)
        next_population.append(child)

    # Add elites to the new population
    population = elites + next_population

duration = time.time() - start_time

return {
    "final_cube": best_cube,
    "final_cost": best_cost,
    "average_cost": round(best_cost / 109, 4),
    "duration": round(duration, 2),
    "iteration": len(costs),
    "population": population_size,
    "costs": costs,
    "states": states,
}

```

D. Integrasi

1. main.py

```
from fastapi import FastAPI, HTTPException, UploadFile, File
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
import os
import json
from algorithm.utils import (
    initialize_random_cube,
    objective_function
)
from algorithm import (
    steepest_ascent_algorithm,
    sideways_move_algorithm,
    stochastic_algorithm,
    random_restart_algorithm,
    simulated_annealing_algorithm,
    genetic_algorithm,
)
app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

N = 5
SAVE_DIR = "./cube"
if not os.path.exists(SAVE_DIR):
    os.makedirs(SAVE_DIR)

class CubeInitResponse(BaseModel):
    initial_cube: list
    initial_cost: int
```

```

class AlgorithmRequest(BaseModel):
    algorithm: str
    cube: list

class AlgorithmResponse(BaseModel):
    final_cube: list
    final_cost: int
    average_cost: float
    duration: float
    iteration: int
    restart: int | None = None
    iteration_restart: list | None = None
    local_optima: int | None = None
    population: int | None = None
    costs: list
    states: list
    exps: list | None = None

class CubeData(BaseModel):
    file_name: str
    cube: list

class CubeCostRequest(BaseModel):
    cube: list

class CubeCostResponse(BaseModel):
    cost: int

algorithm_map = {
    'steepest': steepest_ascent_algorithm,
    'sideways': sideways_move_algorithm,
    'stochastic': stochastic_algorithm,
    'random': random_restart_algorithm,
    'simulated': simulated_annealing_algorithm,
    'genetic': genetic_algorithm
}

@app.get("/initialize_cube", response_model=CubeInitResponse)

```

```

async def initialize_cube():
    initial_cube = initialize_random_cube(N)
    initial_cost = objective_function(initial_cube)
    return {
        "initial_cube": initial_cube,
        "initial_cost": initial_cost
    }

@app.post("/run_algorithm", response_model=AlgorithmResponse)
async def run_algorithm(request: AlgorithmRequest):
    algorithm_function = algorithm_map.get(request.algorithm)
    if not algorithm_function:
        raise HTTPException(status_code=404, detail="Algorithm not found")

    try:
        result = algorithm_function(request.cube)
        return {
            "final_cube": result.get("final_cube"),
            "final_cost": result.get("final_cost"),
            "average_cost": result.get("average_cost"),
            "duration": result.get("duration"),
            "iteration": result.get("iteration"),
            "restart": result.get("restart", None),
            "iteration_restart": result.get("iteration_restart",
None),
            "local_optima": result.get("local_optima", None),
            "population": result.get("population", None),
            "costs": result.get("costs"),
            "states": result.get("states"),
            "exps": result.get("exps", None)
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/calculate_cost", response_model=CubeCostResponse)
async def calculate_cost(request: CubeCostRequest):
    try:
        cost = objective_function(request.cube)
    
```

```

        return {"cost": cost}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/save_cube")
async def save_cube(cube_data: CubeData):
    file_path = os.path.join(SAVE_DIR, f"{cube_data.file_name}.json")
    with open(file_path, "w") as file:
        json.dump({"magic_cube": cube_data.cube}, file)
    return {"message": "Cube saved successfully"}

@app.post("/load_cube", response_model=CubeInitResponse)
async def load_cube(file: UploadFile = File(...)):
    try:
        content = await file.read()
        data = json.loads(content)
        magic_cube = data.get("magic_cube")
        if not magic_cube or not isinstance(magic_cube, list):
            raise HTTPException(status_code=400, detail="Invalid cube
format")

        # Check if the loaded cube meets the magic cube objective
        cost = objective_function(magic_cube)
        return {
            "initial_cube": magic_cube,
            "initial_cost": cost
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed to load
cube: {str(e)}")

```

Pada algoritma main.py melakukan berbagai import dari mulai fastapi, fastapi.middleware.cors, pydantic, os, json, algorithm.utils. From fastapi dengan import FastAPI, Digunakan untuk membuat aplikasi/website API berbasis FastAPI. HTTPException, Digunakan untuk mengembalikan error dengan kode status HTTP (seperti 404 atau 500) dan pesan kustom. UploadFile dan File Mendukung endpoint untuk mengunggah file (seperti file JSON untuk memuat cube). From fastapi.middleware.cors dengan import CORSMiddleware, digunakan untuk Middleware yang memungkinkan API diakses dari berbagai domain

(cross-origin). From pydantic import BaseModel, digunakan untuk membuat model data (schema) yang memvalidasi input dan output API, seperti Format request untuk algoritma, format response untuk inisialisasi cube atau hasil algoritma. Import os, membantu membuat direktori untuk menyimpan file jika direktori belum ada. Import json, digunakan untuk membaca dan menyimpan data dalam format JSON, seperti menyimpan cube ke file atau memuat file cube. From algorithm.utils Fungsi untuk membuat magic cube acak ukuran N x N dan fungsi untuk mengevaluasi cost atau nilai dari sebuah cube. From algorithm digunakan untuk mengimport fungsi dari setiap algoritma. Selanjutnya, Inisialisasi Variabel dan Direktori dan model data. Kemudian, algoritma map digunakan untuk mempermudah pemanggilan fungsi algoritma. Bagian terakhir, adalah untuk menyimpan dan memuat cube ke/dari file JSON serta mengelola interaksi dengan cube melalui API.

2. App.jsx

```
import React, { useState } from "react";
import Main from "./components/Main";
import Steepest from "./components/Steepest";
import Sideways from "./components/Sideways";
import Stochastic from "./components/Stochastic";
import Random from "./components/Random";
import Simulated from "./components/Simulated";
import Genetic from "./components/Genetic";

const API_URL = 'http://localhost:8000';

const App = () => {
  const [selectedAlgorithm, setSelectedAlgorithm] = useState(null);
  const [initialCube, setInitialCube] = useState(() => {
    const savedCube = localStorage.getItem('initialCube');
    return savedCube ? JSON.parse(savedCube) : null;
  });
  const [initialCost, setInitialCost] = useState(() => {
    const savedCost = localStorage.getItem('initialCost');
    return savedCost ? JSON.parse(savedCost) : null;
  });
  const [finalCube, setFinalCube] = useState(null);
  const [finalCost, setFinalCost] = useState(null);
  const [averageCost, setAverageCost] = useState(null);
  const [duration, setDuration] = useState(null);
```

```

const [iteration, setIteration] = useState(null);
const [restart, setRestart] = useState(null);
const [localOptima, setLocalOptima] = useState(null);
const [iterationRestart, setIterationRestart] = useState([]);
const [population, setPopulation] = useState(null);
const [costs, setCosts] = useState([]);
const [states, setStates] = useState([]);
const [exps, setExps] = useState([]);
const [isLoading, setIsLoading] = useState(false);

// Fetch new initial cube and set state
const fetchInitialCube = async () => {
  setIsLoading(true);
  try {
    const response = await fetch(`${API_URL}/initialize_cube`);
    if (!response.ok) throw new Error('Network response was not ok');
    const data = await response.json();

    // Save the initial cube to both state and local storage
    setInitialCube(data.initial_cube);
    setInitialCost(data.initial_cost);
    localStorage.setItem('initialCube',
      JSON.stringify(data.initial_cube));
    localStorage.setItem('initialCost',
      JSON.stringify(data.initial_cost));
  } catch (error) {
    console.error("Failed to fetch initial cube:", error);
  } finally {
    setIsLoading(false);
  }
};

const onLoadCube = async (loadedCube) => {
  setIsLoading(true);
  try {
    const response = await fetch(`${API_URL}/calculate_cost`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ cube: loadedCube })
  }

```

```

});  

if (!response.ok) throw new Error('Failed to calculate cost');  

const data = await response.json();  

setInitialCube(loadedCube);  

setInitialCost(data.cost);  

localStorage.setItem('initialCube', JSON.stringify(loadedCube));  

localStorage.setItem('initialCost', JSON.stringify(data.cost));  

} catch (error) {  

  console.error("Failed to calculate cost:", error);  

} finally {  

  setIsLoading(false);  

}  

};  

const handleAlgorithm = async (algorithmType) => {  

  setIsLoading(true);  

try {  

  const response = await fetch(`${API_URL}/run_algorithm`, {  

    method: 'POST',  

    headers: { 'Content-Type': 'application/json' },  

    body: JSON.stringify({  

      algorithm: algorithmType,  

      cube: initialCube
    })
  });
}  

if (!response.ok) throw new Error('Algorithm execution failed');  

const result = await response.json();  

setFinalCube(result.final_cube);  

setFinalCost(result.final_cost);  

setAverageCost(result.average_cost);  

setDuration(result.duration);  

setIteration(result.iteration);  

setCosts(result.costs);  

setStates(result.states);  

if (algorithmType === 'random') {  

  setRestart(result.restart);  

  setIterationRestart(result.iteration_restart);
}

```

```

    }

    if (algorithmType === 'simulated') {
        setLocalOptima(result.local_optima);
        setExps(result.exps);
    }

    if (algorithmType === 'genetic') {
        setPopulation(result.population);
    }

    setSelectedAlgorithm(algorithmType);
} catch (error) {
    console.error("Algorithm execution failed:", error);
} finally {
    setIsLoading(false);
}
};

const handleBack = () => {
    setSelectedAlgorithm(null);
};

const renderAlgorithm = () => {
    const commonProps = {
        initialCube,
        finalCube,
        initialCost,
        finalCost,
        averageCost,
        duration,
        iteration,
        costs,
        states,
        onBack: handleBack,
        isLoading
    };

    const specificProps = {
        random: { ...commonProps, restart, iterationRestart },
        simulated: { ...commonProps, localOptima, exps },
        genetic: { ...commonProps, population }
    };
}

```

```

};

switch (selectedAlgorithm) {
    case "steepest": return <Steepest {...commonProps} />;
    case "sideways": return <Sideways {...commonProps} />;
    case "stochastic": return <Stochastic {...commonProps} />;
    case "random": return <Random {...specificProps.random} />;
    case "simulated": return <Simulated {...specificProps.simulated}>
/>;
    case "genetic": return <Genetic {...specificProps.genetic} />;
    default: return null;
}
};

return (
<>
{!selectedAlgorithm ? (
<Main
    onAlgorithmSelect={handleAlgorithm}
    initialCube={initialCube}
    initialCost={initialCost}
    onInitialize={fetchInitialCube}
    onLoadCube={onLoadCube}
    isLoading={isLoading}
/>
) : (
    renderAlgorithm()
)
}
</>
);
};

export default App;

```

E. Hasil Eksperimen dan Analisis

Magic Cube Solver

Solve Magic Cube, Your Way.

Explore a world of solutions for your Magic Cube with our solver.
Visualize, learn, and apply different algorithms tailored to your
solving style.

Initial State Cube

Initial State Cost: 7846

Reset view

Load Cube Save Cube Initialize Random Cube

Choose Your Algorithm

- Steepest Ascent**
Optimal path finding algorithm
- Sideways Move**
Flexible movement patterns
- Stochastic**
Randomized solution approach
- Random Restart**
Multiple starting points
- Simulated Annealing**
Temperature-based solving
- Genetic**
Evolution-inspired solution

© 2024 Magic Cube Solver. All rights reserved.

Gambar 2.1 Laman Awal Visualisasi

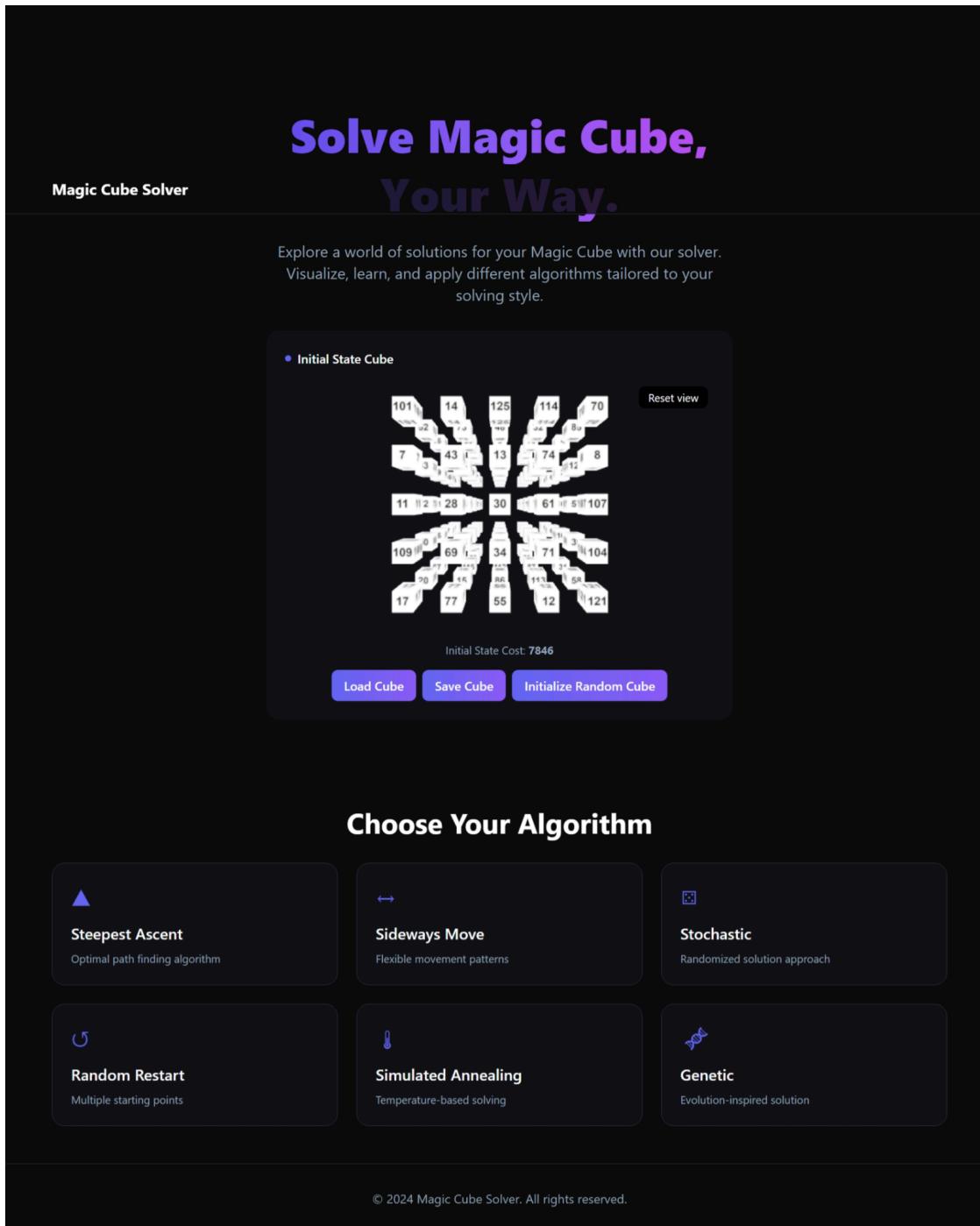
Website Magic Cube Solver menampilkan antarmuka interaktif yang memungkinkan pengguna memvisualisasikan dan menyelesaikan *Diagonal Magic Cube* dengan pendekatan yang personal. Antarmuka ini terdiri dari visualisasi kubus, informasi *objective function*, dan tombol-tombol kontrol untuk memanipulasi keadaan kubus. Pada visualisasi *Initial State Cube*, setiap sel kubus diisi dengan angka unik yang didistribusikan secara acak, memberikan gambaran tentang kompleksitas awal. Di bawahnya, terdapat nilai *Initial State Cost*, yang menunjukkan besaran biaya awal dari konfigurasi kubus. *Objective function* dari pencarian ini adalah meminimalkan *cost* tersebut, semakin rendah *cost*, semakin dekat kubus pada solusi optimal. *Website* ini menyediakan algoritma yang dapat diterapkan untuk menurunkan *cost* hingga mencapai solusi yang optimal.

Website ini memiliki tiga tombol kontrol utama yaitu *Load Cube* untuk memuat konfigurasi yang telah disimpan, *Save Cube* untuk menyimpan konfigurasi saat ini, dan *Initialize Random Cube* untuk mengacak konfigurasi kubus yang baru. Berbagai algoritma tersedia untuk membantu mencapai solusi optimal dengan memanipulasi posisi dan nilai kubus, memberikan pengalaman interaktif dan edukatif dalam menyelesaikan *Diagonal Magic Cube*.

1. Eksperimen I

a. Inisialisasi Cube

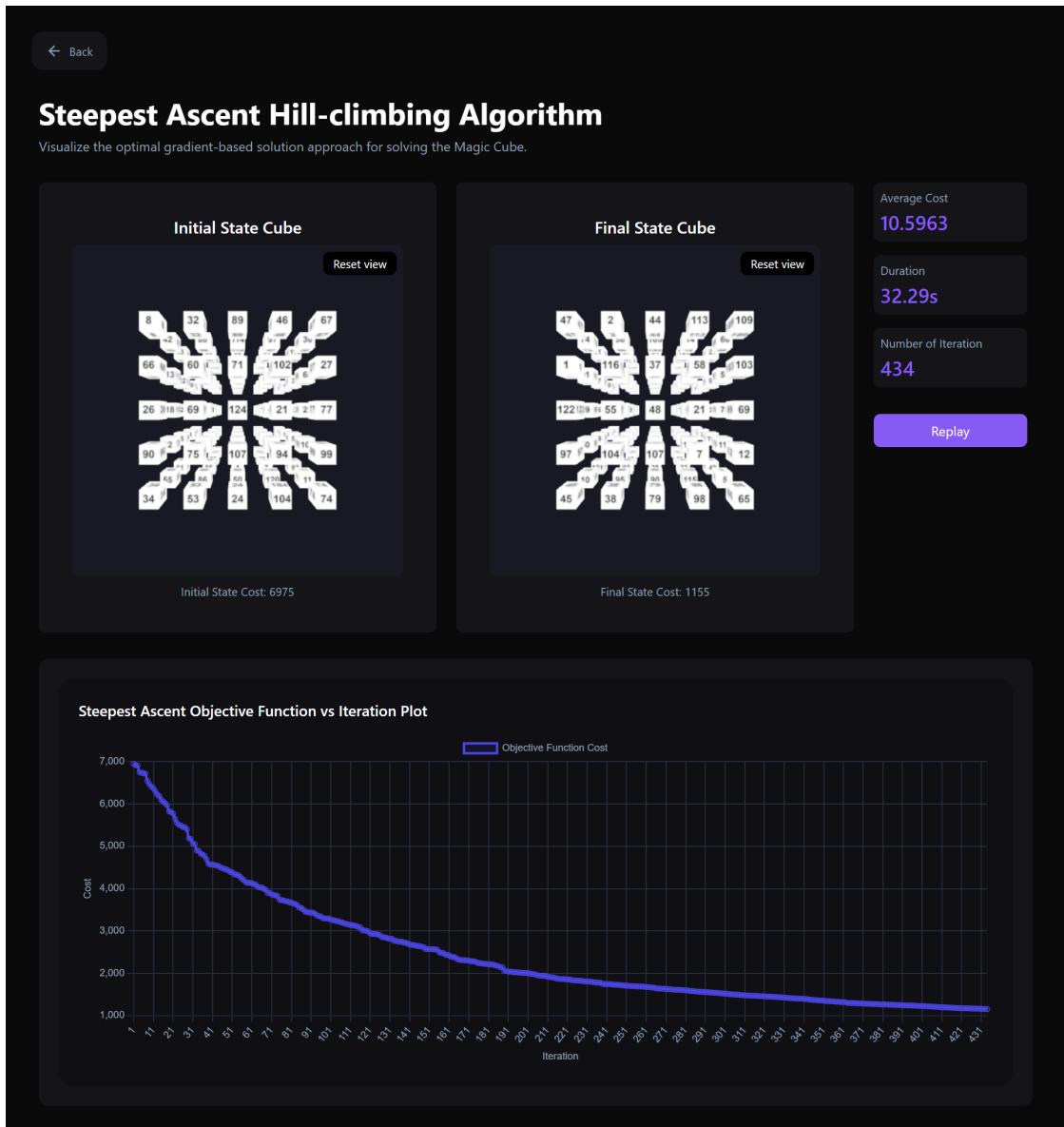
Pada eksperimen pertama, setelah dilakukan inisialisasi *random cube*, didapatkan bahwa *initial state cost* kubus adalah 7846. Kubus ini akan digunakan pada seluruh algoritma untuk menentukan seberapa dekat algoritma tersebut mendekati global optima.



Gambar 2.2 Inisialisasi Cube I

b. Steepest Ascent Hill-climbing

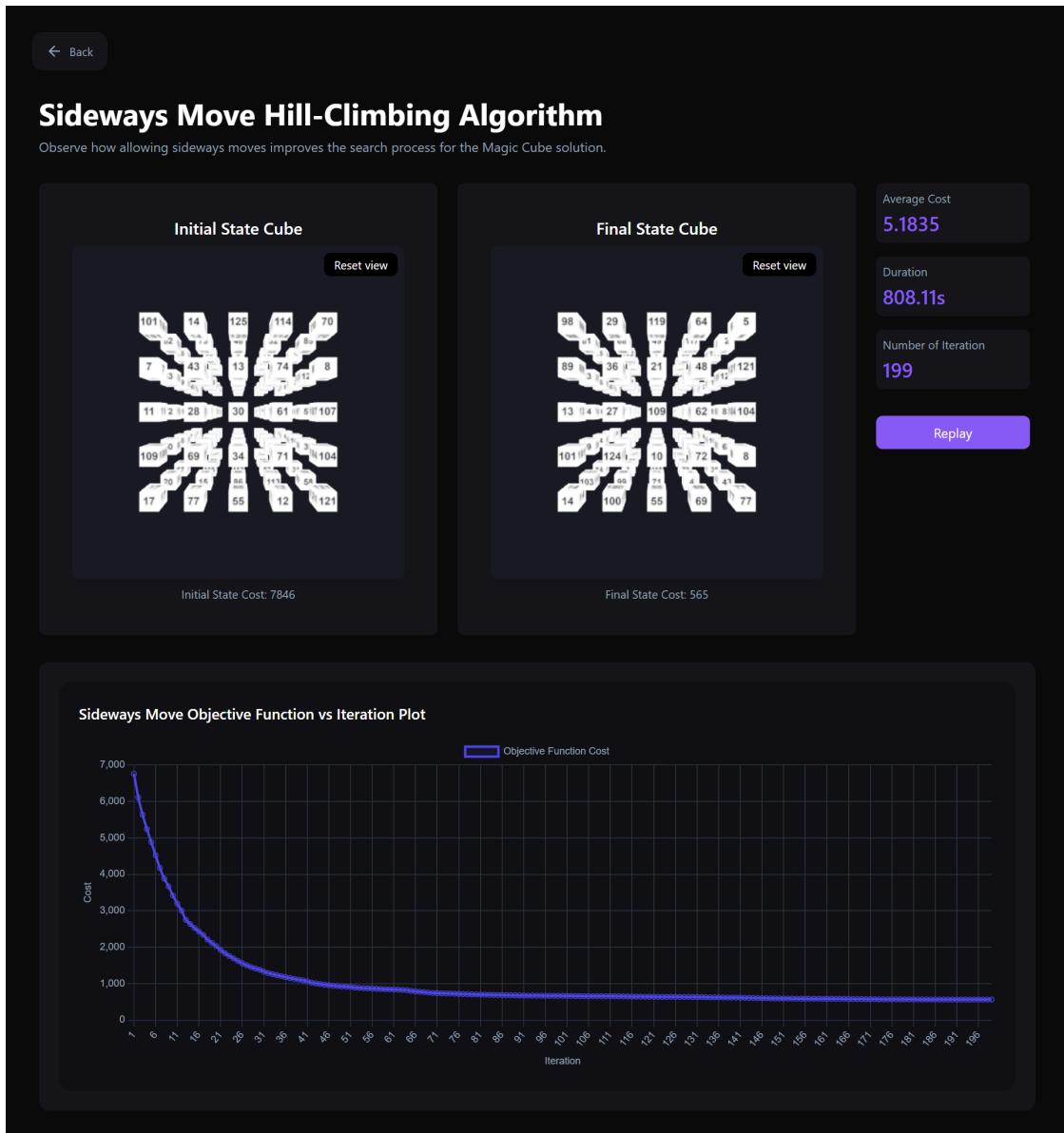
Pada algoritma *Steepest Ascent Hill-climbing*, didapatkan *final cost* yaitu 732 dengan *average cost*. Seluruh proses ini membutuhkan waktu selama 30.22 detik dan dengan total iterasi yaitu 344.



Gambar 2.3 Steepest Ascent Hill-climbing Eksperimen I

c. Hill-climbing with Sideways Move

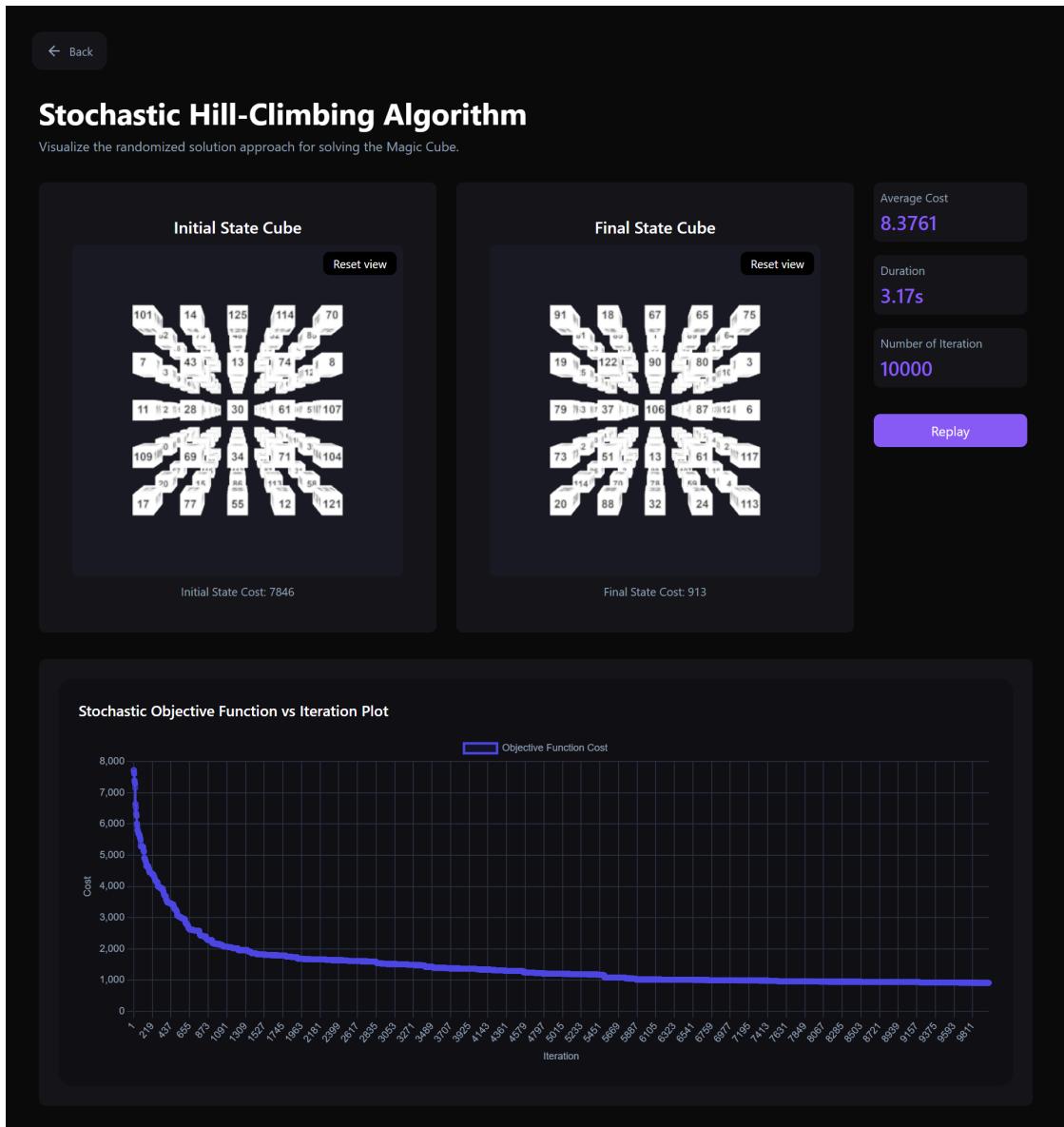
Pada algoritma *Hill-climbing with Sideways Move*, didapatkan *final cost* yaitu 565 dengan *average cost* sebesar 5.1835. Seluruh proses ini membutuhkan waktu selama 808.11 detik dan dengan total iterasi yaitu 199.



Gambar 2.4 *Hill-climbing with Sideways Move Eksperimen I*

d. Stochastic Hill-climbing

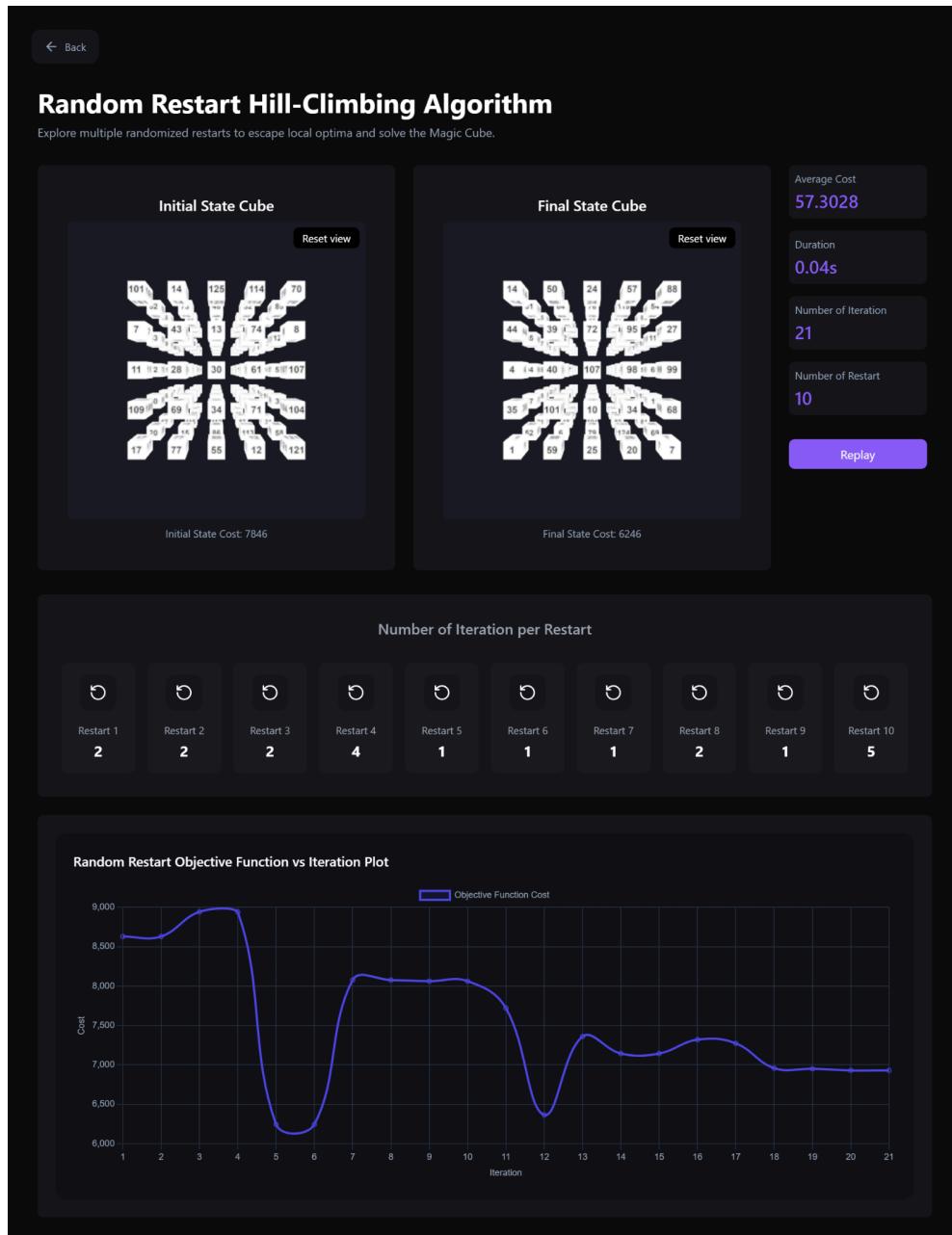
Pada algoritma *Stochastic Hill-climbing*, didapatkan *final cost* yaitu 913 dengan *average cost* sebesar 8.3761. Seluruh proses ini membutuhkan waktu selama 3.17 detik dan dengan total iterasi yaitu 10000.



Gambar 2.5 *Stochastic Hill-climbing* Eksperimen I

e. Random Restart Hill-climbing

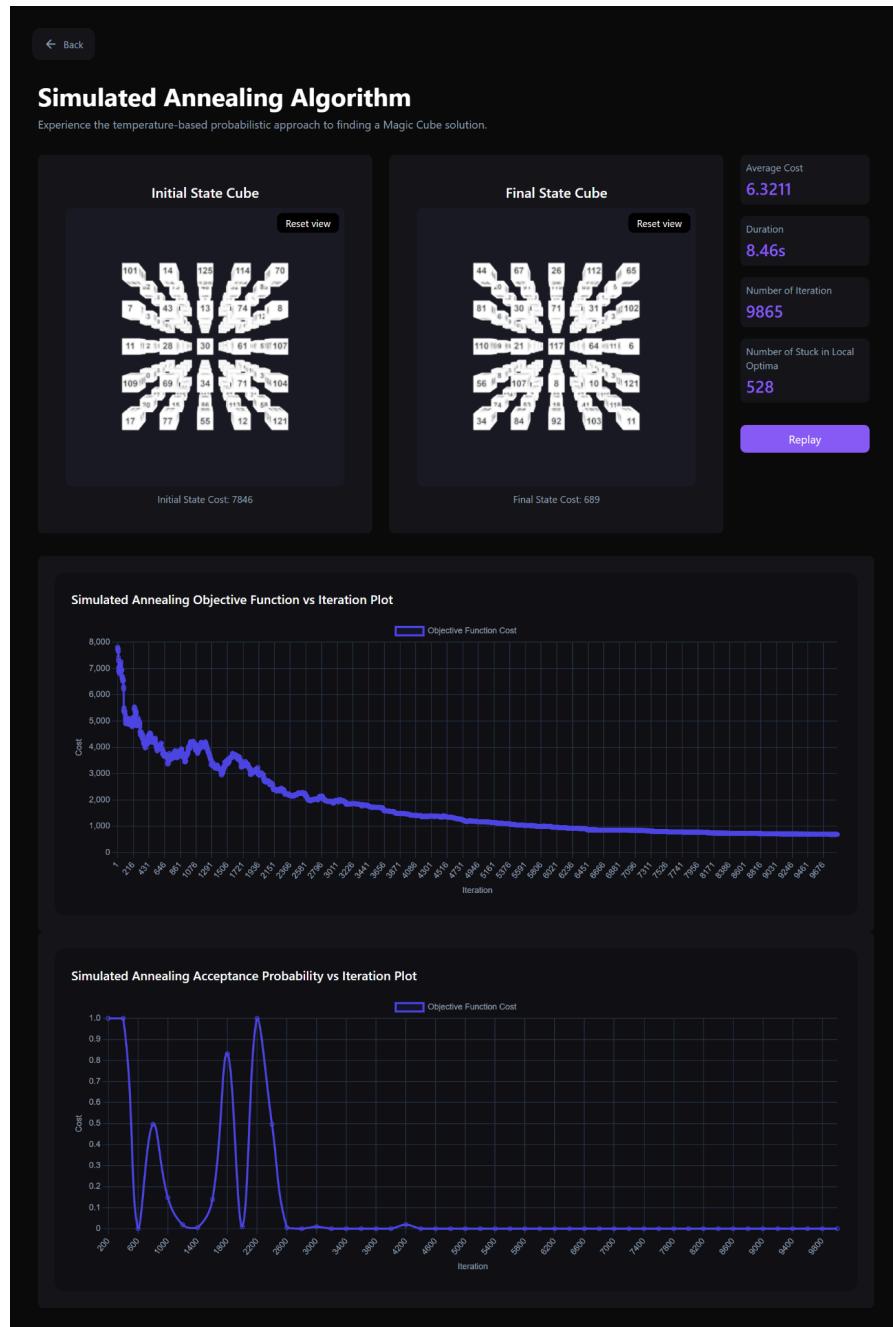
Pada algoritma *Random Restart Hill-climbing*, didapatkan *final cost* yaitu 6246 dengan *average cost* sebesar 57.3028. Seluruh proses ini membutuhkan waktu selama 0.04 detik dan dengan total iterasi 21 dan total *restart* 10.



Gambar 2.6 Random Restart Hill-climbing Eksperimen I

f. Simulated Annealing

Pada algoritma *Simulated Annealing*, didapatkan *final cost* yaitu 689 dengan *average cost* sebesar 6.3211. Seluruh proses ini membutuhkan waktu selama 8.46 detik dan dengan total iterasi yaitu 9865 dan total *stuck* di lokal optimum yaitu 528.

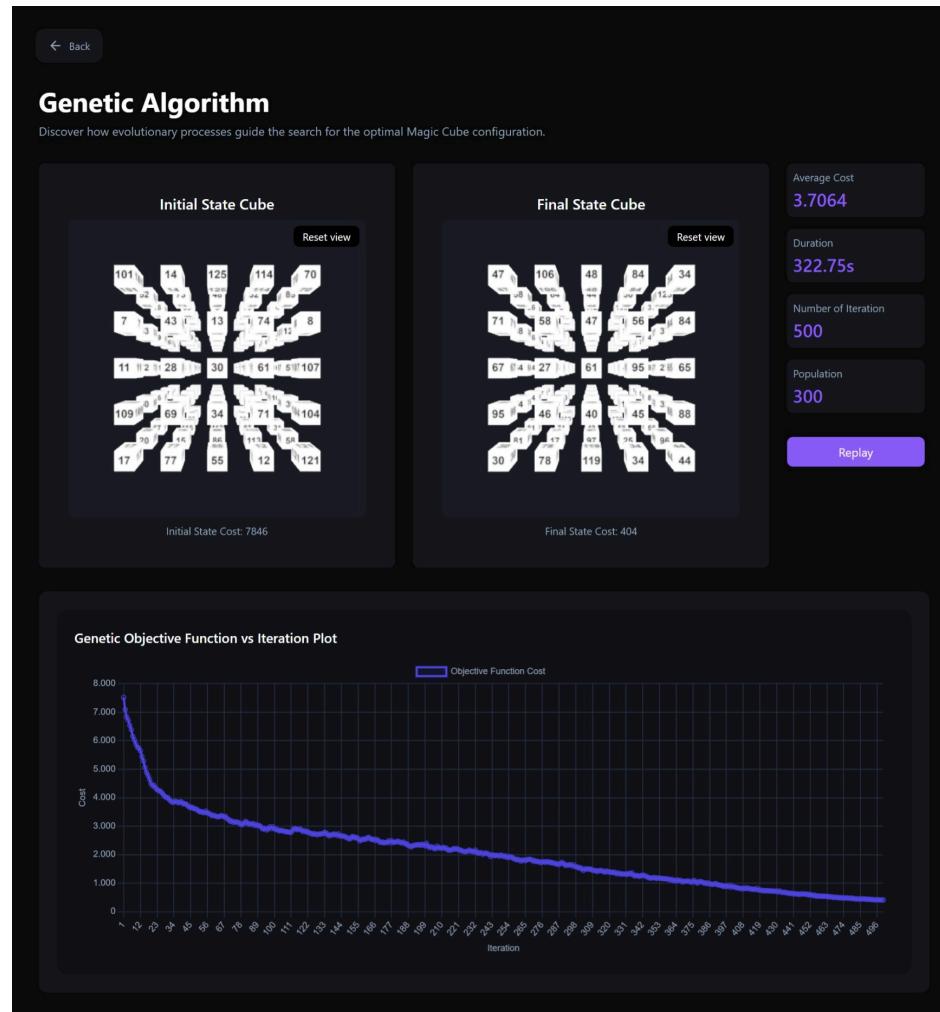


Gambar 2.7 Simulated Annealing Eksperimen I

g. Genetic Algorithm

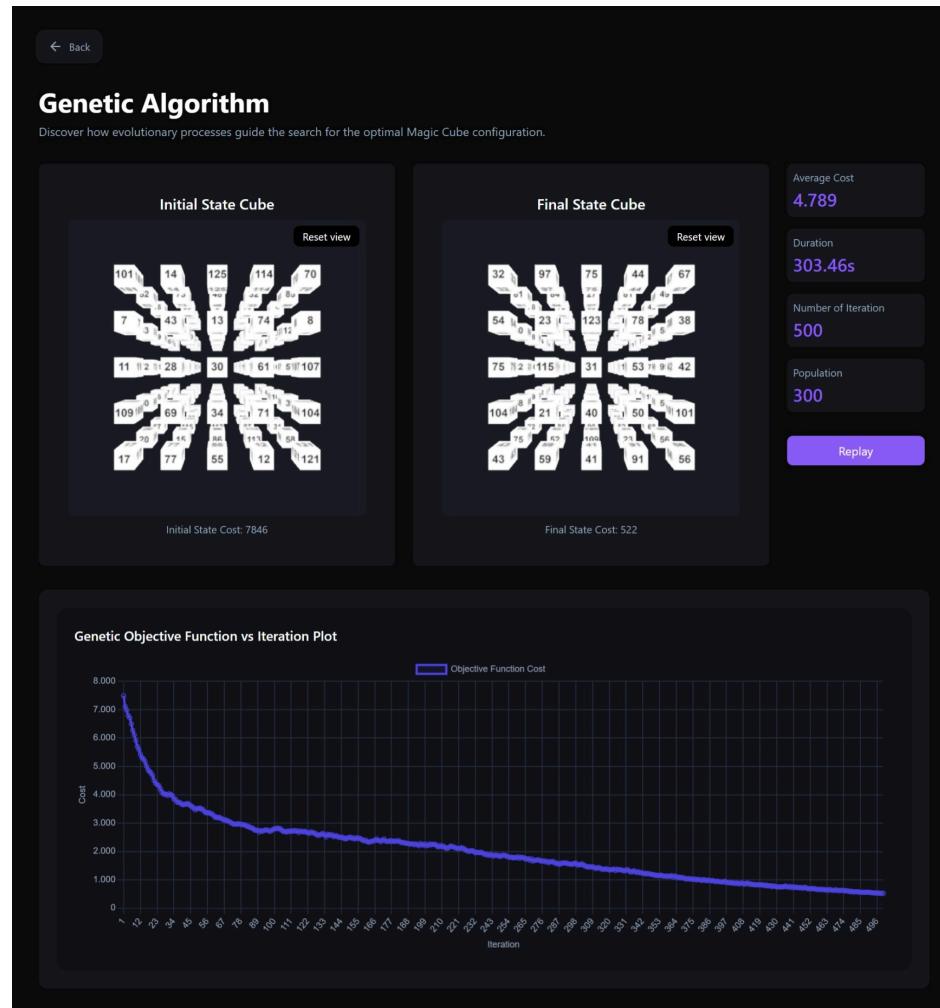
i. Population = 300 Iteration = 500

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama, yaitu 404 dengan *average cost* sebesar 3.7064. Seluruh proses ini membutuhkan waktu selama 322.75 detik dan dengan total iterasi yaitu 500 dan jumlah populasi yaitu 300.



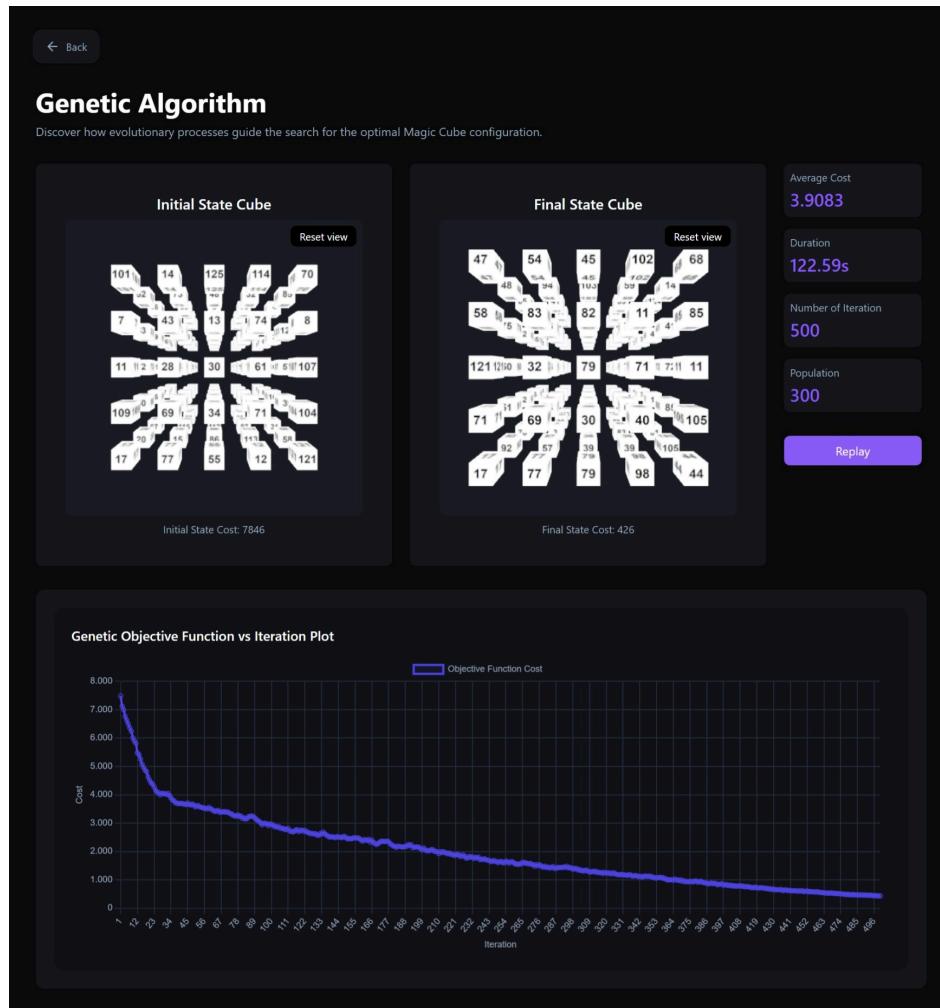
Gambar 2.8 *Genetic Algorithm* i (1) Eksperimen I

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua, yaitu 522 dengan *average cost* sebesar 4.789. Seluruh proses ini membutuhkan waktu selama 303.46 detik dan dengan total iterasi yaitu 500 dan jumlah populasi yaitu 300.



Gambar 2.9 *Genetic Algorithm i (2) Eksperimen I*

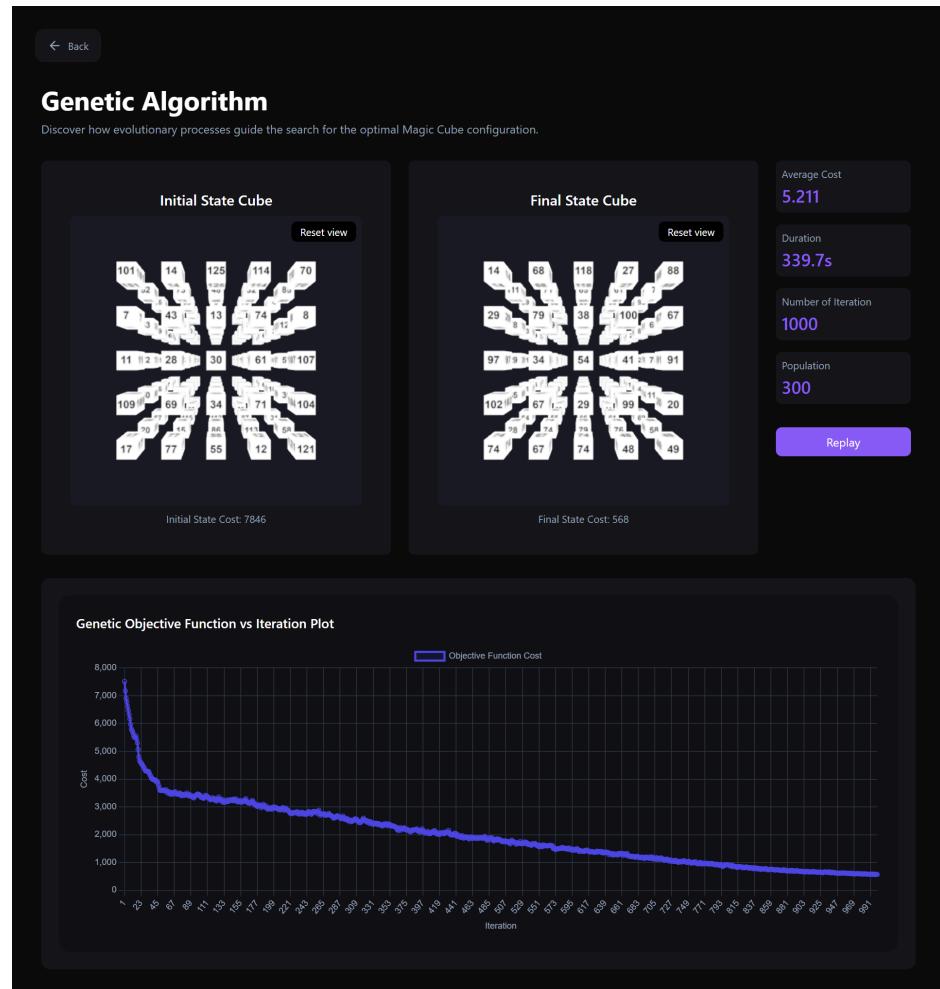
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga, yaitu 426 dengan *average cost* sebesar 3.9083. Seluruh proses ini membutuhkan waktu selama 122.59 detik dan dengan total iterasi yaitu 500 dan jumlah populasi yaitu 300.



Gambar 2.10 Genetic Algorithm i (3) Eksperimen I

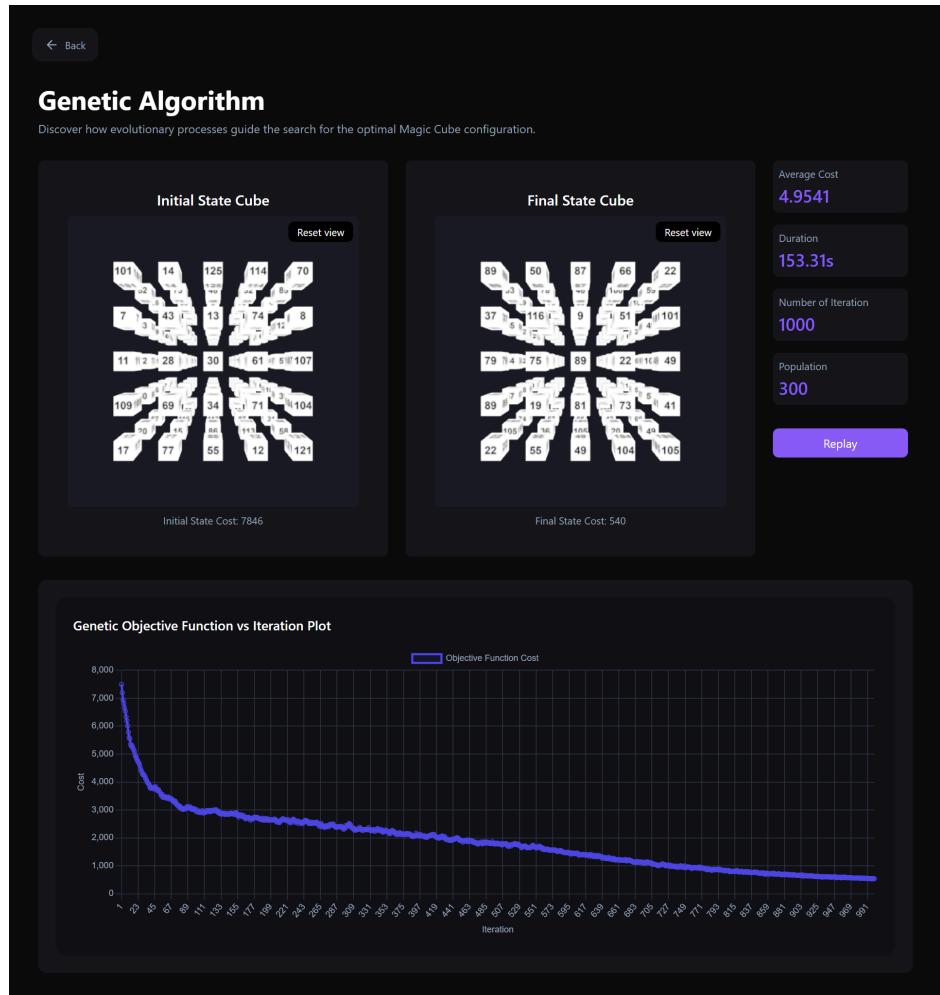
ii. Population = 300 Iteration = 1000

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama, yaitu 568 dengan *average cost* sebesar 5.211. Seluruh proses ini membutuhkan waktu selama 339.7 detik dan dengan total iterasi yaitu 1000 dan jumlah populasi yaitu 300.



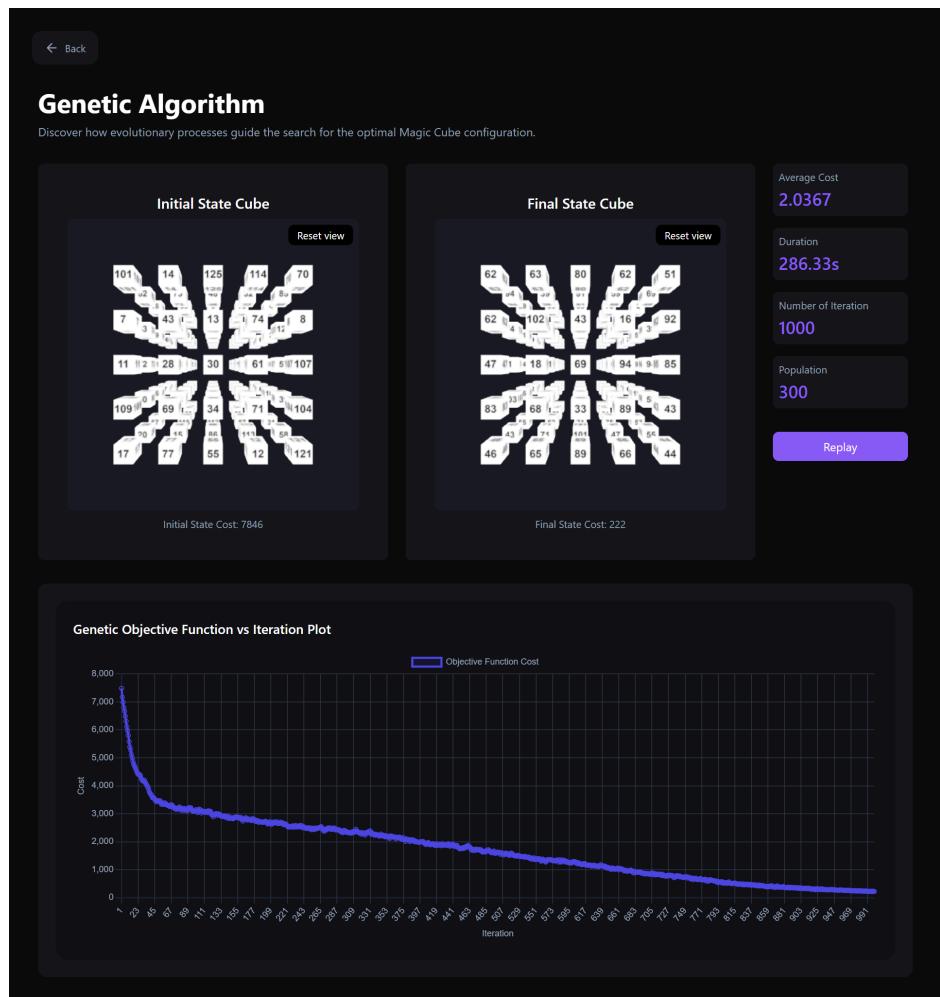
Gambar 2.11 *Genetic Algorithm* ii (1) Eksperimen I

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua, yaitu 540 dengan *average cost* sebesar 4.9541. Seluruh proses ini membutuhkan waktu selama 153.31 detik dan dengan total iterasi yaitu 1000 dan jumlah populasi yaitu 300.



Gambar 2.12 *Genetic Algorithm* ii (2) Eksperimen I

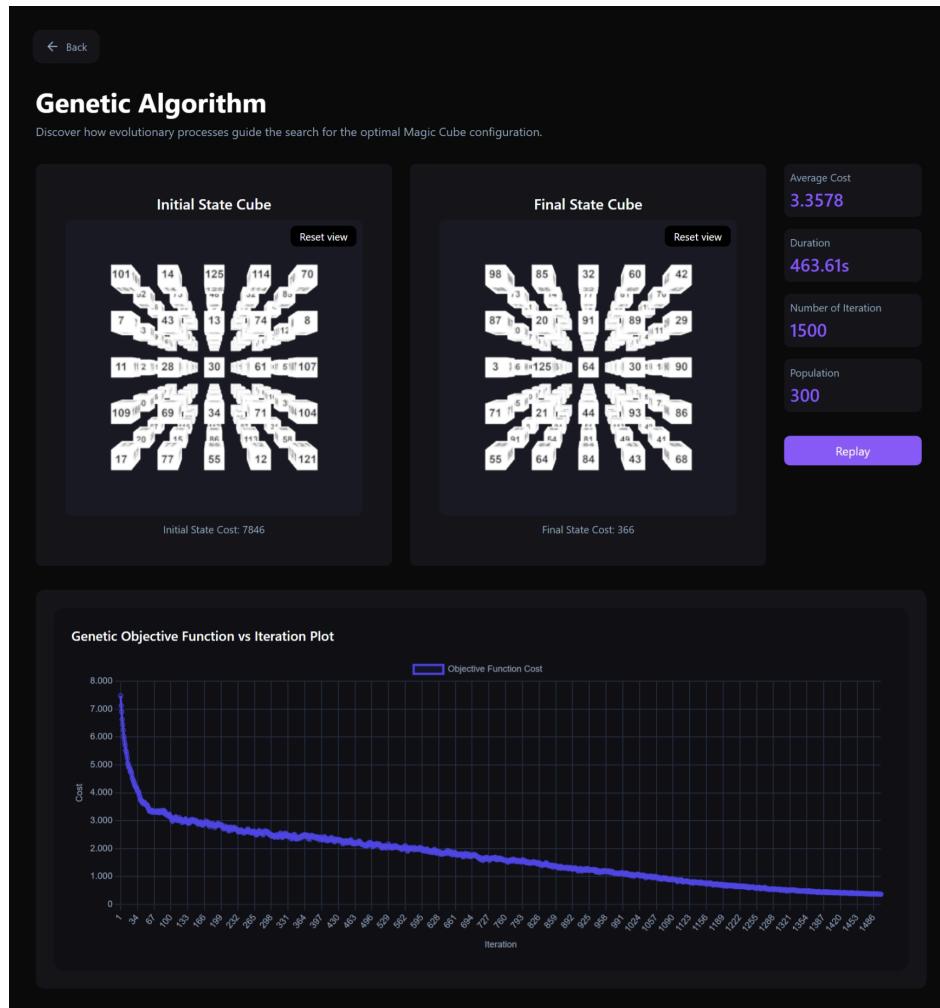
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga, yaitu 222 dengan *average cost* sebesar 2.0367. Seluruh proses ini membutuhkan waktu selama 286.33 detik dan dengan total iterasi yaitu 1000 dan jumlah populasi yaitu 300.



Gambar 2.13 *Genetic Algorithm* ii (3) Eksperimen I

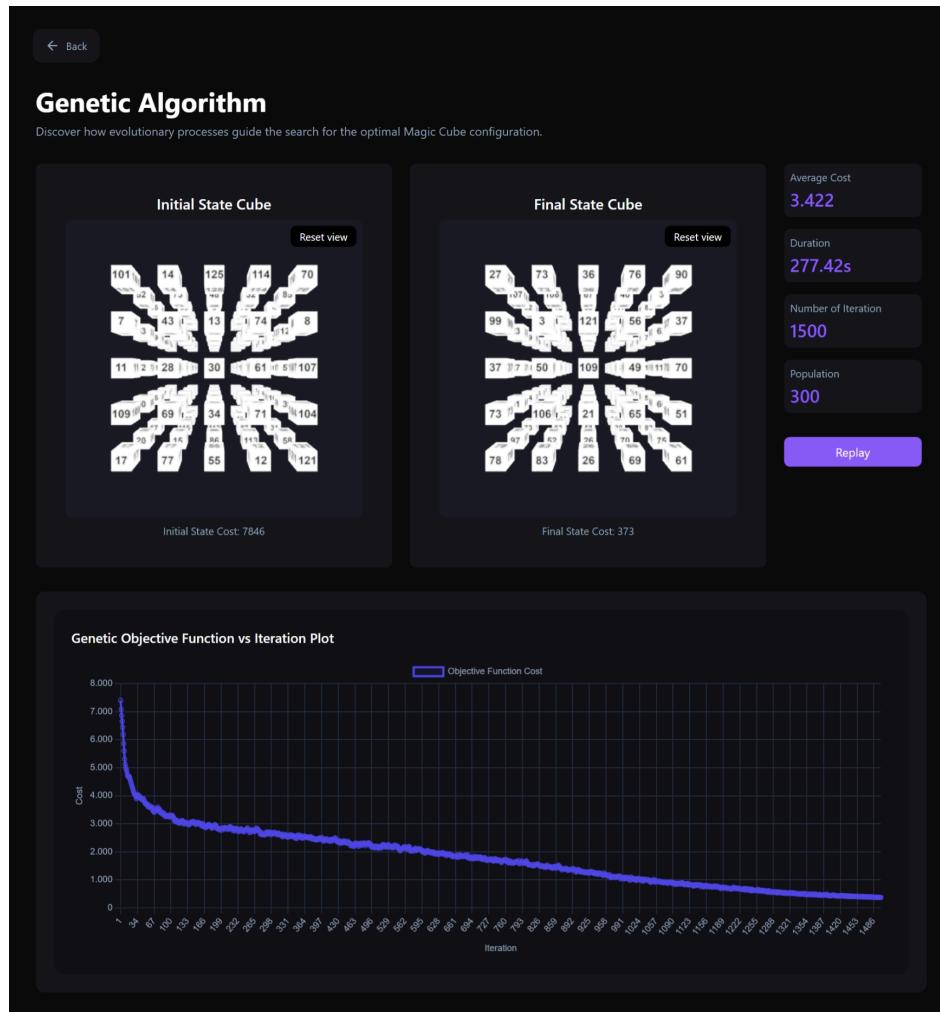
iii. Population = 300 Iteration = 1500

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama, yaitu 366 dengan *average cost* sebesar 3.3578. Seluruh proses ini membutuhkan waktu selama 463.61 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 300.



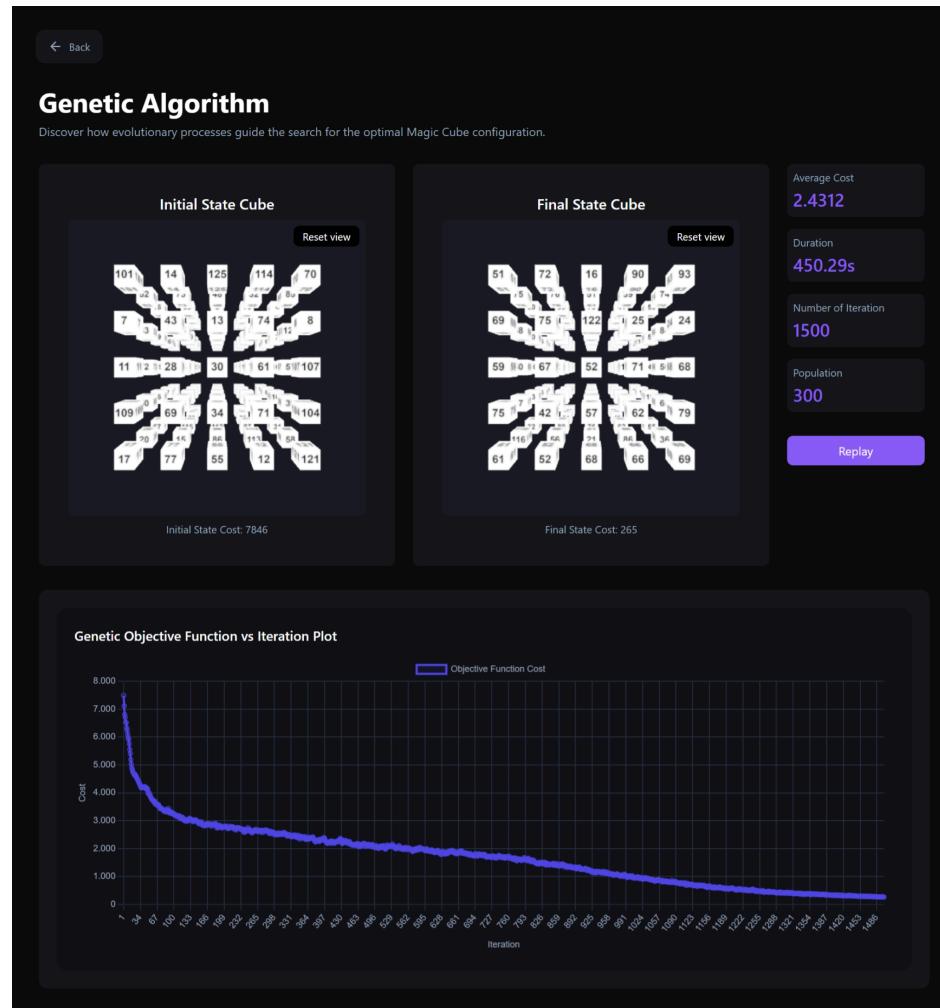
Gambar 2.14 *Genetic Algorithm* iii (1) Eksperimen I

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua, yaitu 373 dengan *average cost* sebesar 3.422. Seluruh proses ini membutuhkan waktu selama 277.42 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 300.



Gambar 2.15 *Genetic Algorithm* iii (2) Eksperimen I

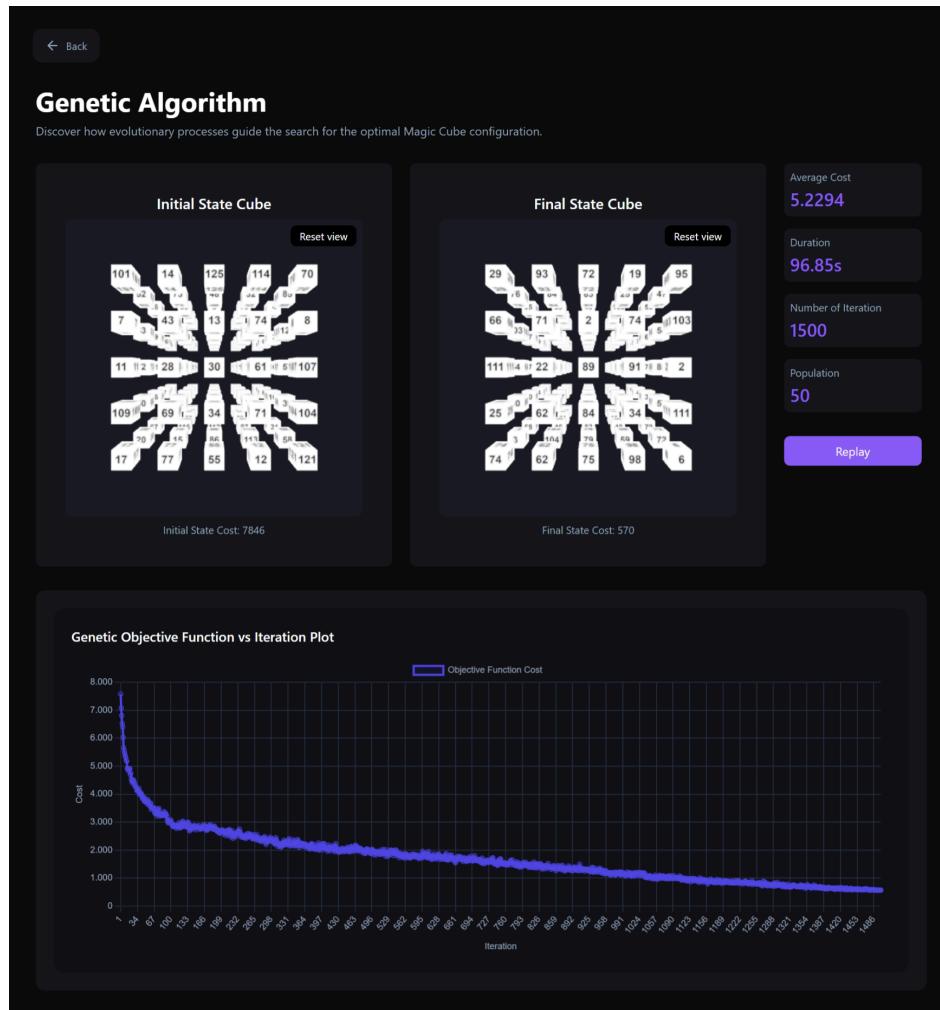
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga, yaitu 265 dengan *average cost* sebesar 2.4312. Seluruh proses ini membutuhkan waktu selama 450.29 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 300.



Gambar 2.16 *Genetic Algorithm* iii (3) Eksperimen I

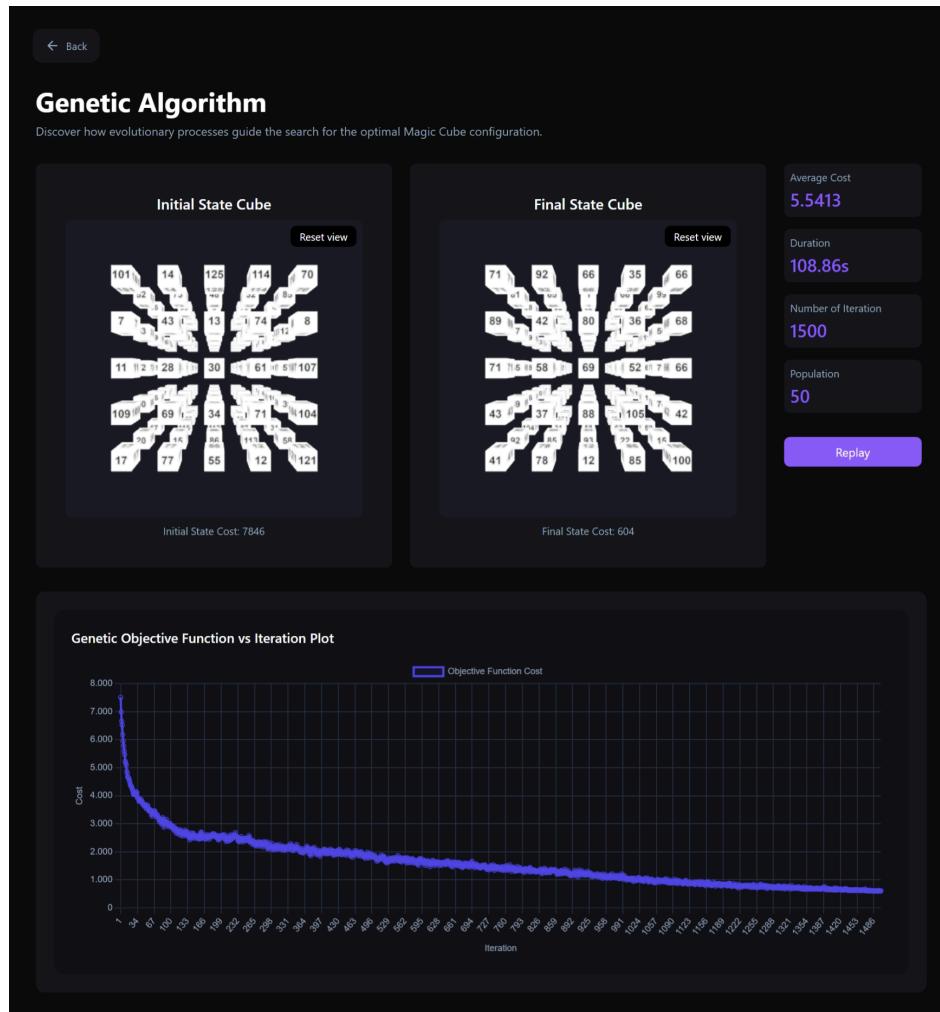
iv. **Iteration = 1500 Population = 50**

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama, yaitu 570 dengan *average cost* sebesar 5.2294. Seluruh proses ini membutuhkan waktu selama 96.85 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 50.



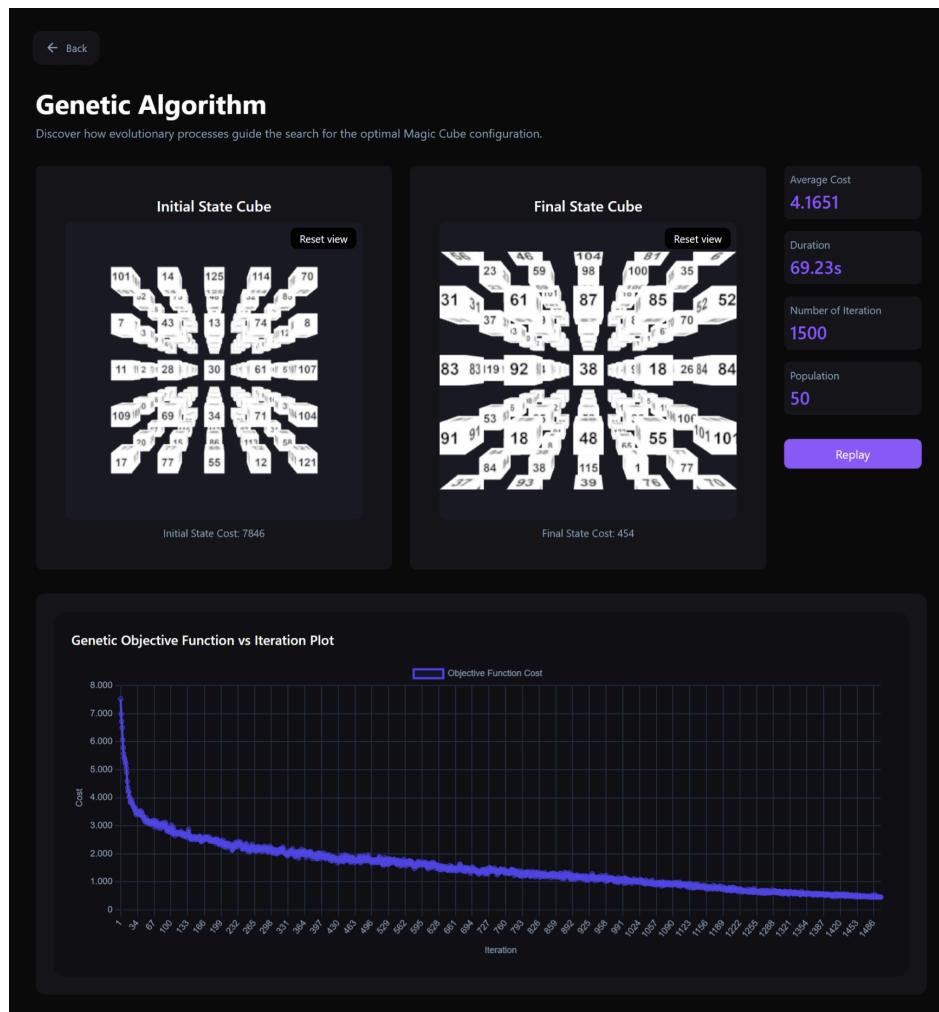
Gambar 2.17 *Genetic Algorithm* iv (1) Eksperimen I

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua, yaitu 604 dengan *average cost* sebesar 5.5413. Seluruh proses ini membutuhkan waktu selama 108.86 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 50.



Gambar 2.18 *Genetic Algorithm iv (2) Eksperimen I*

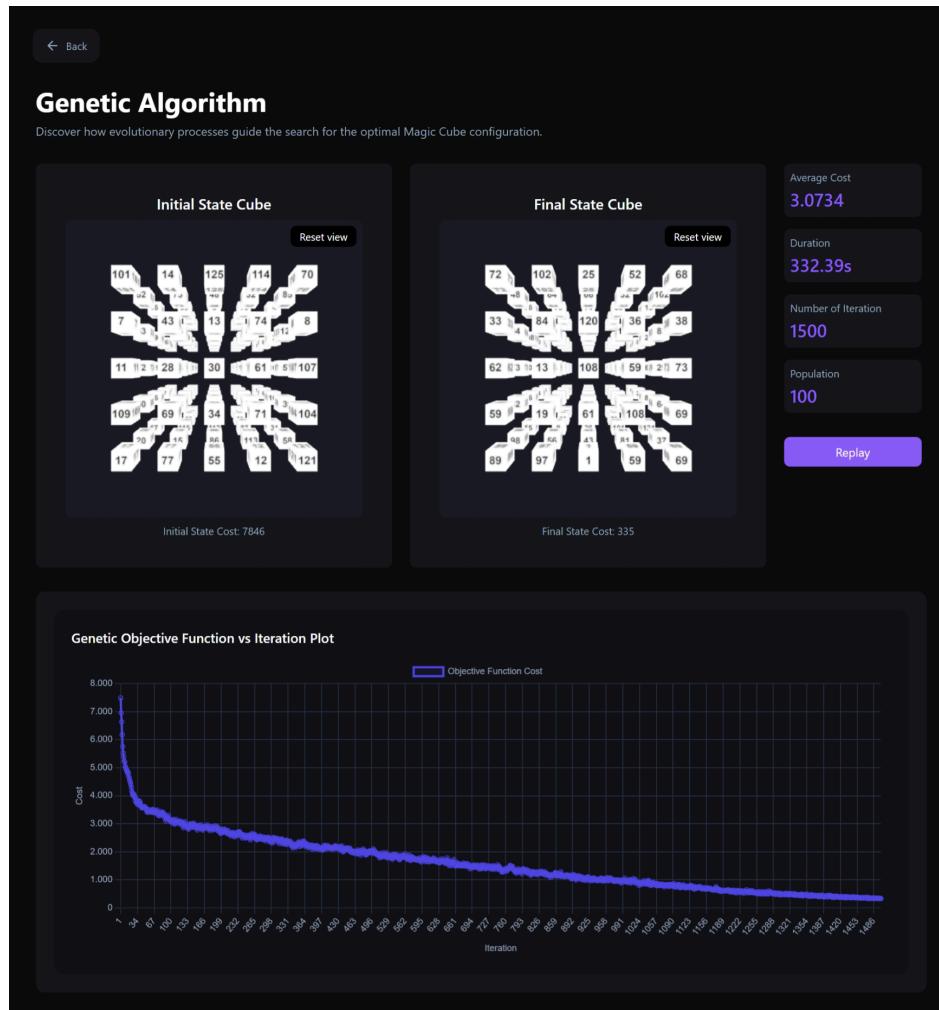
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga, yaitu 454 dengan *average cost* sebesar 4.1651. Seluruh proses ini membutuhkan waktu selama 69.23 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 50.



Gambar 2.19 *Genetic Algorithm* iv (3) Eksperimen I

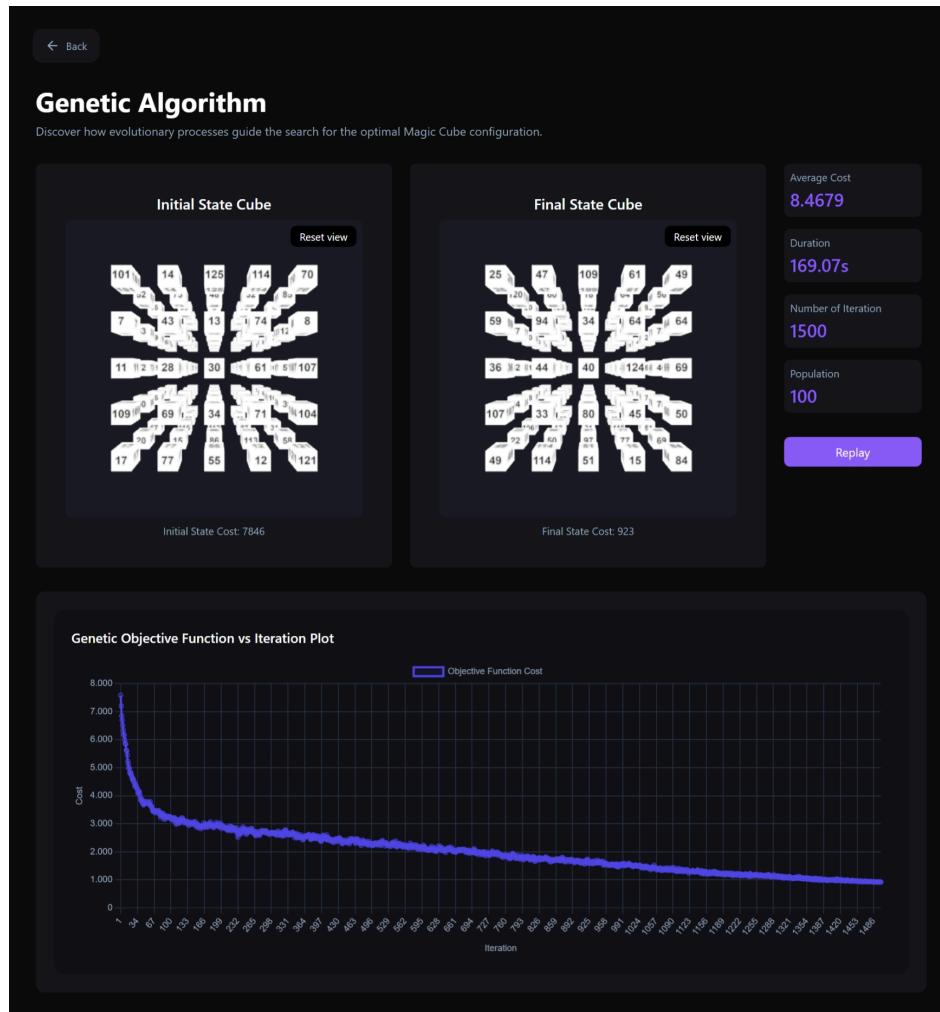
v. Iteration = 1500 Population = 100

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama, yaitu 335 dengan *average cost* sebesar 3.0734. Seluruh proses ini membutuhkan waktu selama 332.39 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 100.



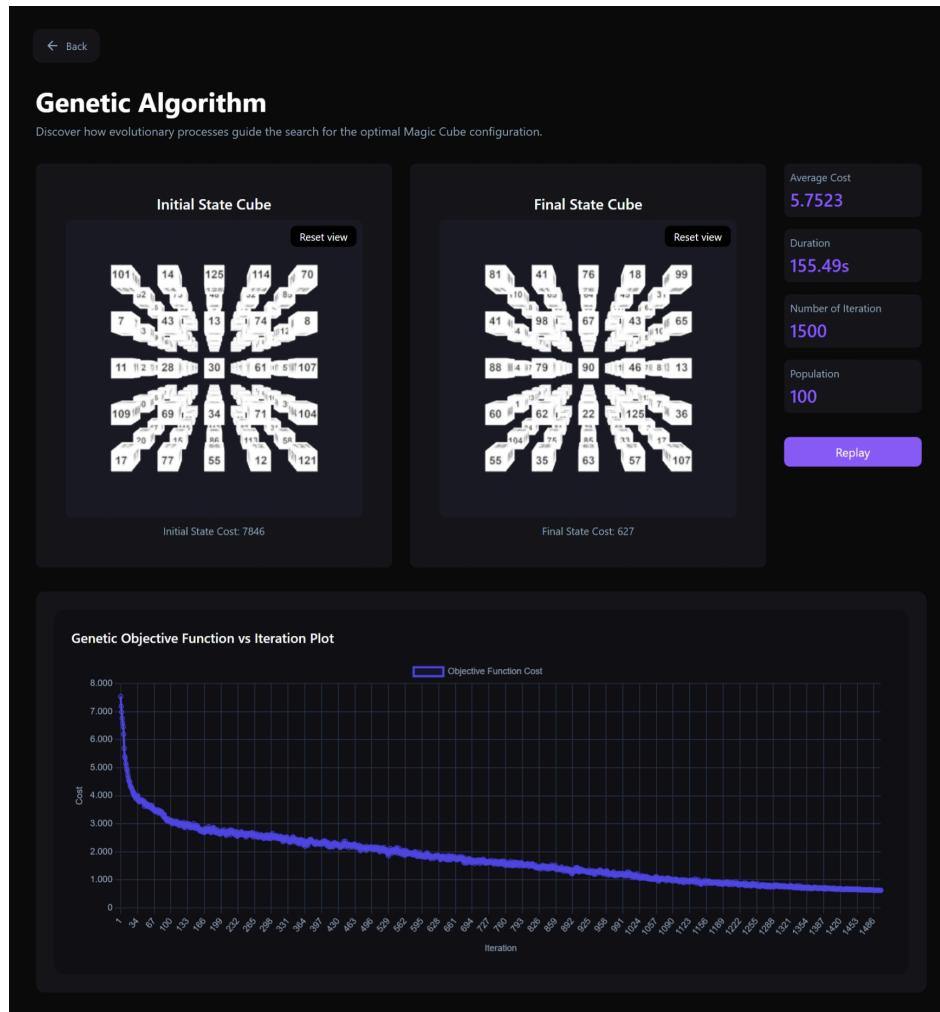
Gambar 2.20 *Genetic Algorithm* v (1) Eksperimen I

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua, yaitu 923 dengan *average cost* sebesar 8.4679. Seluruh proses ini membutuhkan waktu selama 169.07detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 100.



Gambar 2.21 *Genetic Algorithm* v (2) Eksperimen I

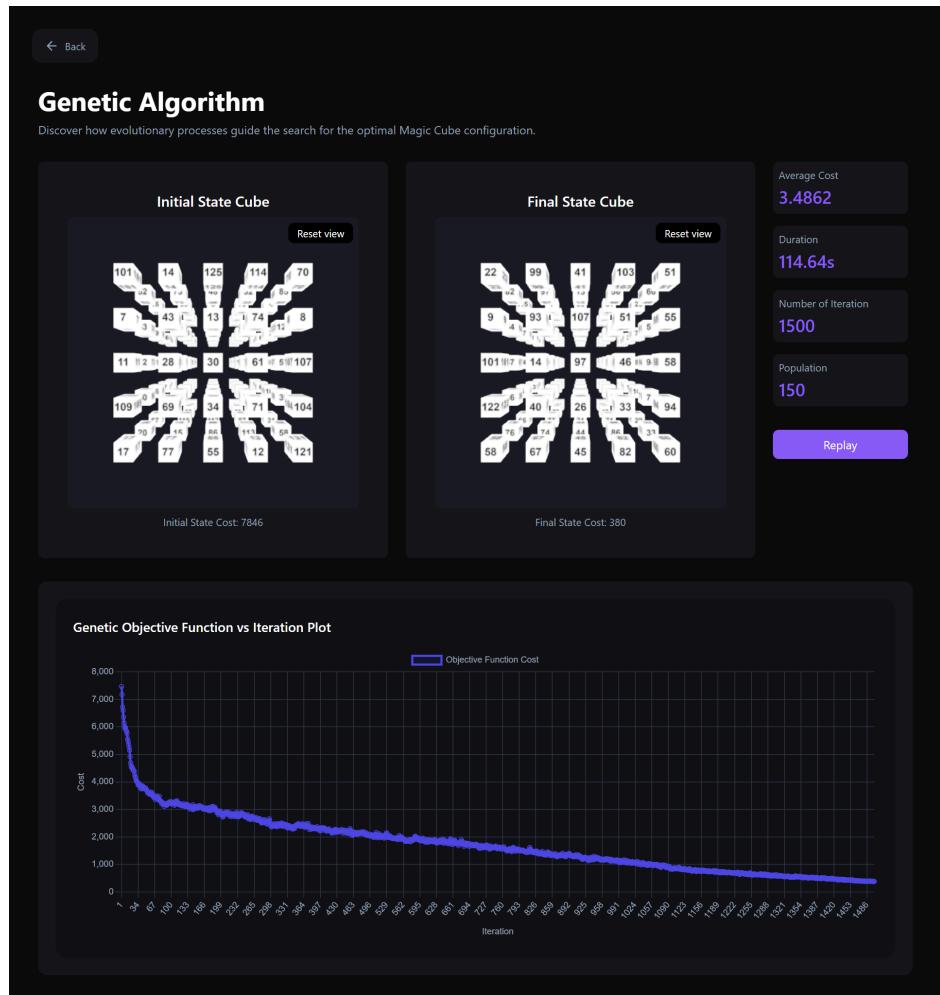
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga, yaitu 627 dengan *average cost* sebesar 5.7523. Seluruh proses ini membutuhkan waktu selama 155.49 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 100.



Gambar 2.22 *Genetic Algorithm* v (3) Eksperimen I

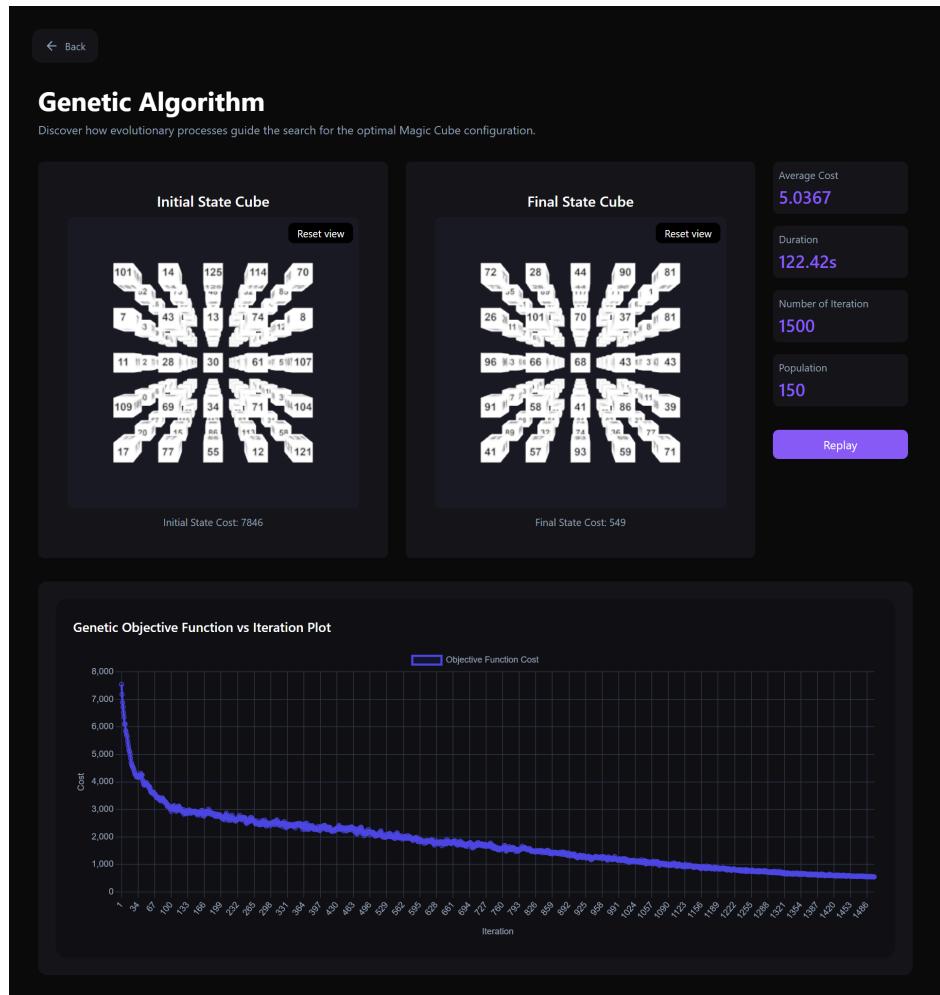
vi. **Iteration = 1500 Population = 150**

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama, yaitu 380 dengan *average cost* sebesar 3.4862 . Seluruh proses ini membutuhkan waktu selama 114.64 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 150.



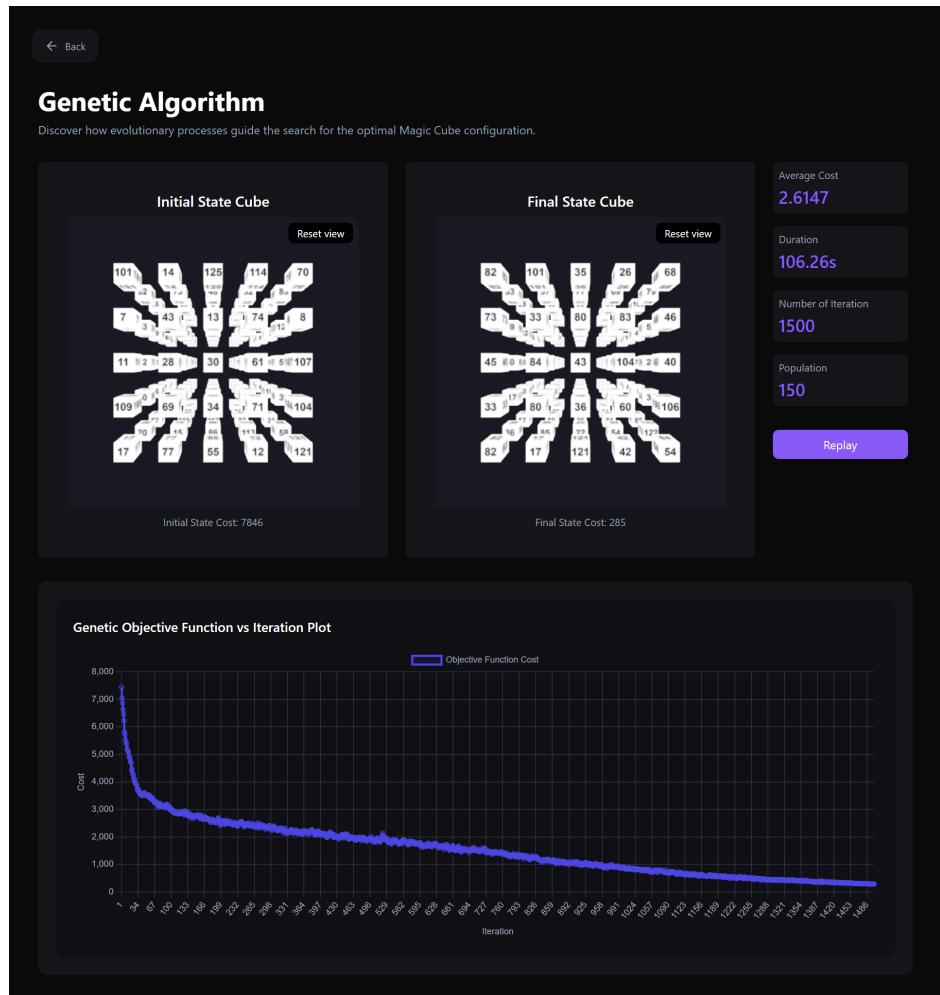
Gambar 2.23 *Genetic Algorithm* vi (1) Eksperimen I

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua, yaitu 549 dengan *average cost* sebesar 5.0367 . Seluruh proses ini membutuhkan waktu selama 122.42 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 150.



Gambar 2.24 *Genetic Algorithm* vi (2) Eksperimen I

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga, yaitu 285 dengan *average cost* sebesar 2.6147 . Seluruh proses ini membutuhkan waktu selama 106.26 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 150.

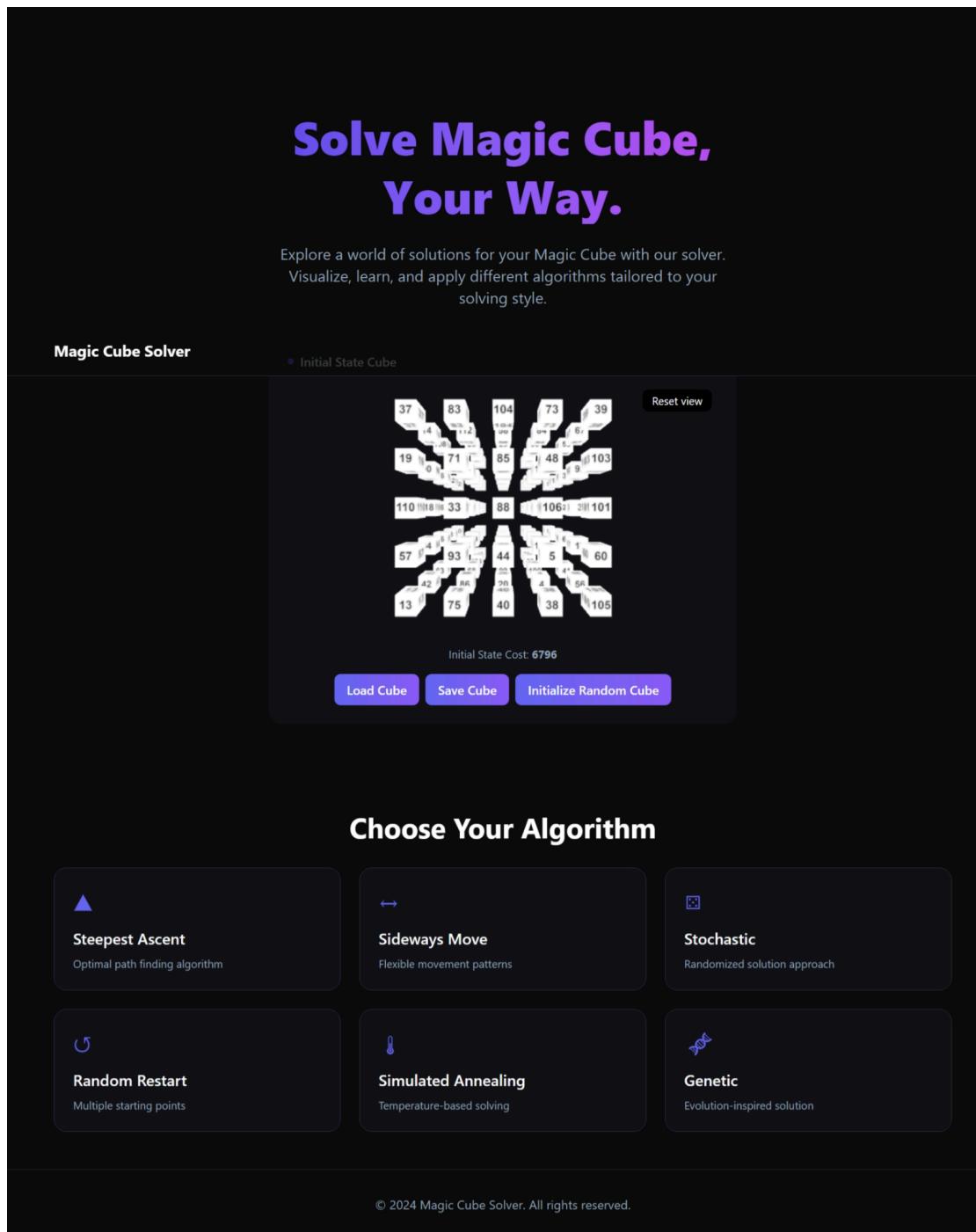


Gambar 2.25 *Genetic Algorithm* vi (3) Eksperimen I

2. Eksperimen II

a. Inisialisasi Cube

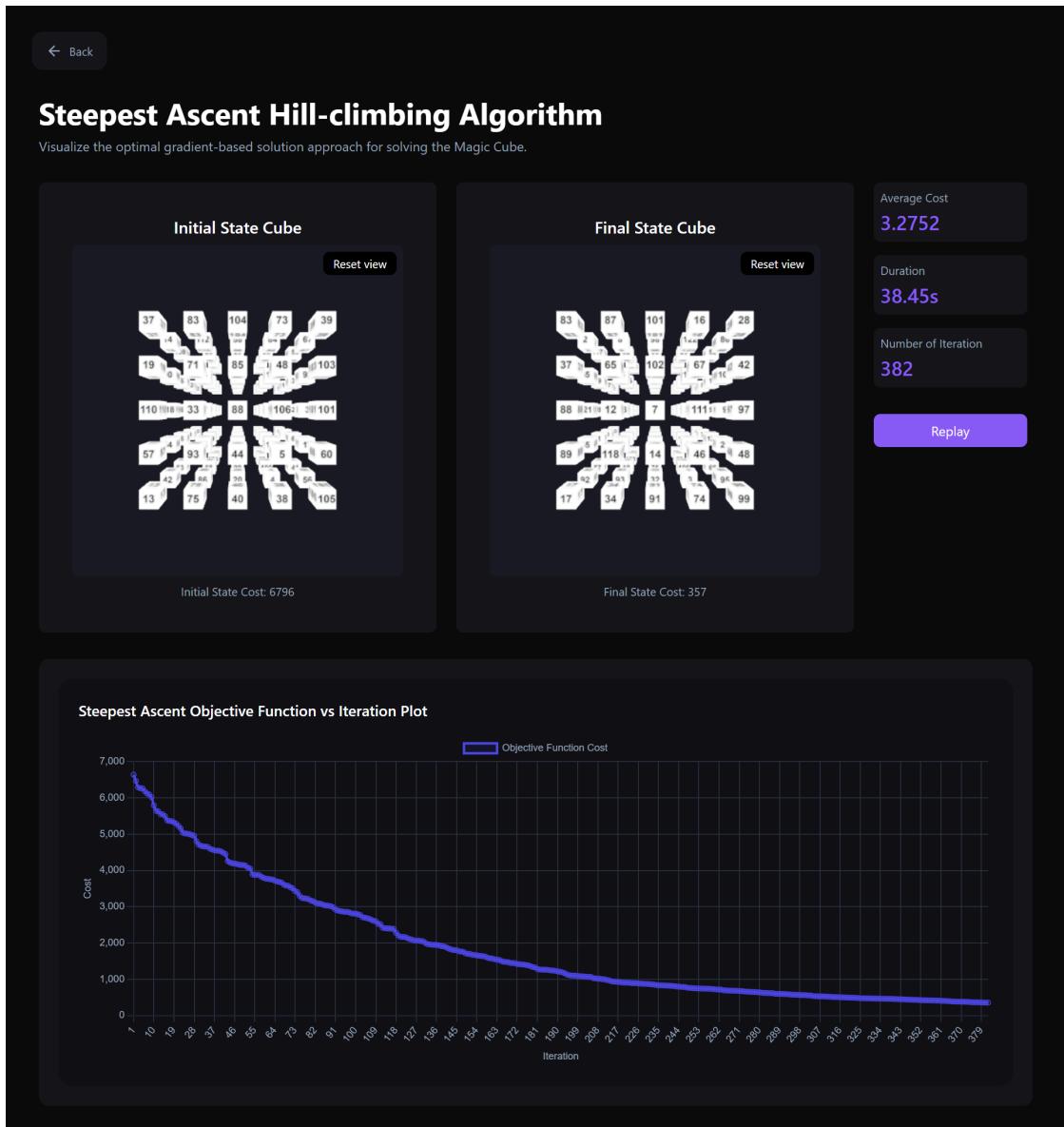
Pada eksperimen kedua, setelah dilakukan inisialisasi *random cube*, didapatkan bahwa *initial state cost* kubus adalah 6796. Kubus ini akan digunakan pada seluruh algoritma untuk menentukan seberapa dekat algoritma tersebut mendekati global optima.



Gambar 2.26 Inisialisasi Cube II

b. **Steepest Ascent Hill-climbing**

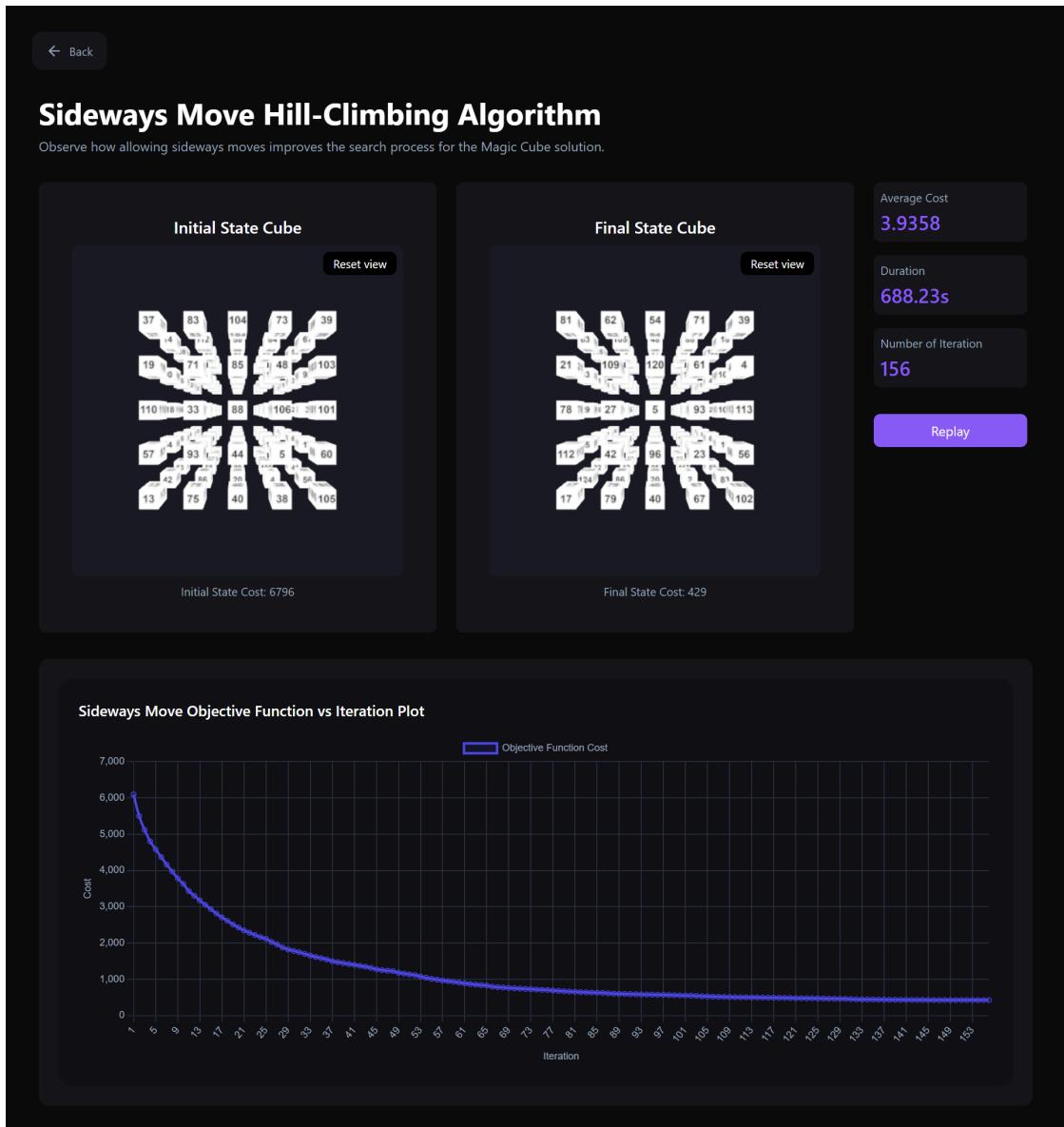
Pada algoritma *Steepest Ascent Hill-climbing*, didapatkan *final cost* yaitu 357 dengan *average cost* sebesar 3.2752. Seluruh proses ini membutuhkan waktu selama 38.45 detik dan dengan total iterasi yaitu 382.



Gambar 2.27 Steepest Ascent Hill-climbing Eksperimen II

c. Hill-climbing with Sideways Move

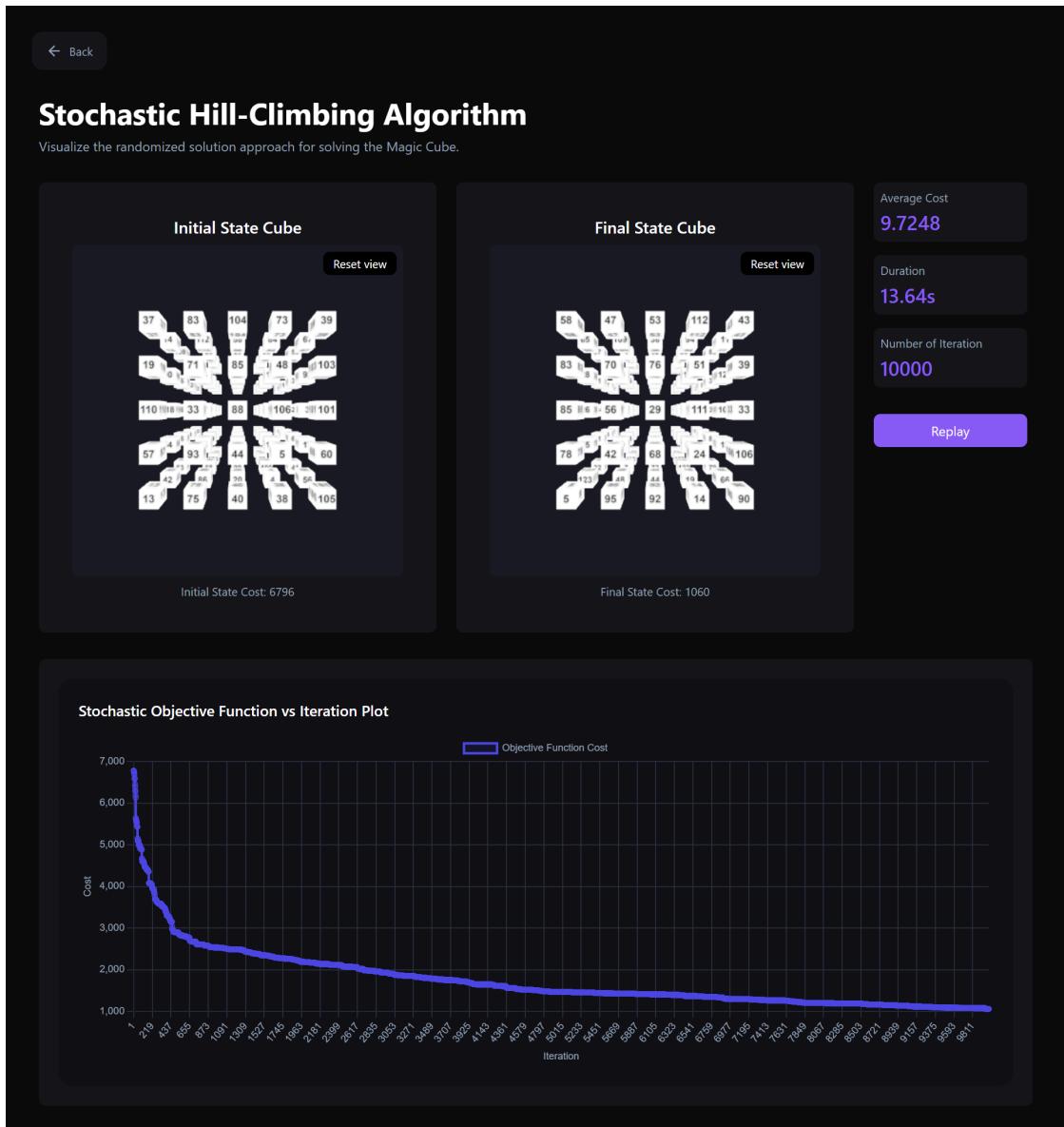
Pada algoritma *Hill-climbing with Sideways Move*, didapatkan *final cost* yaitu 429 dengan *average cost* sebesar 3.9358. Seluruh proses ini membutuhkan waktu selama 688.23 detik dan dengan total iterasi yaitu 156.



Gambar 2.28 *Hill-climbing with Sideways Move Eksperimen II*

d. Stochastic Hill-climbing

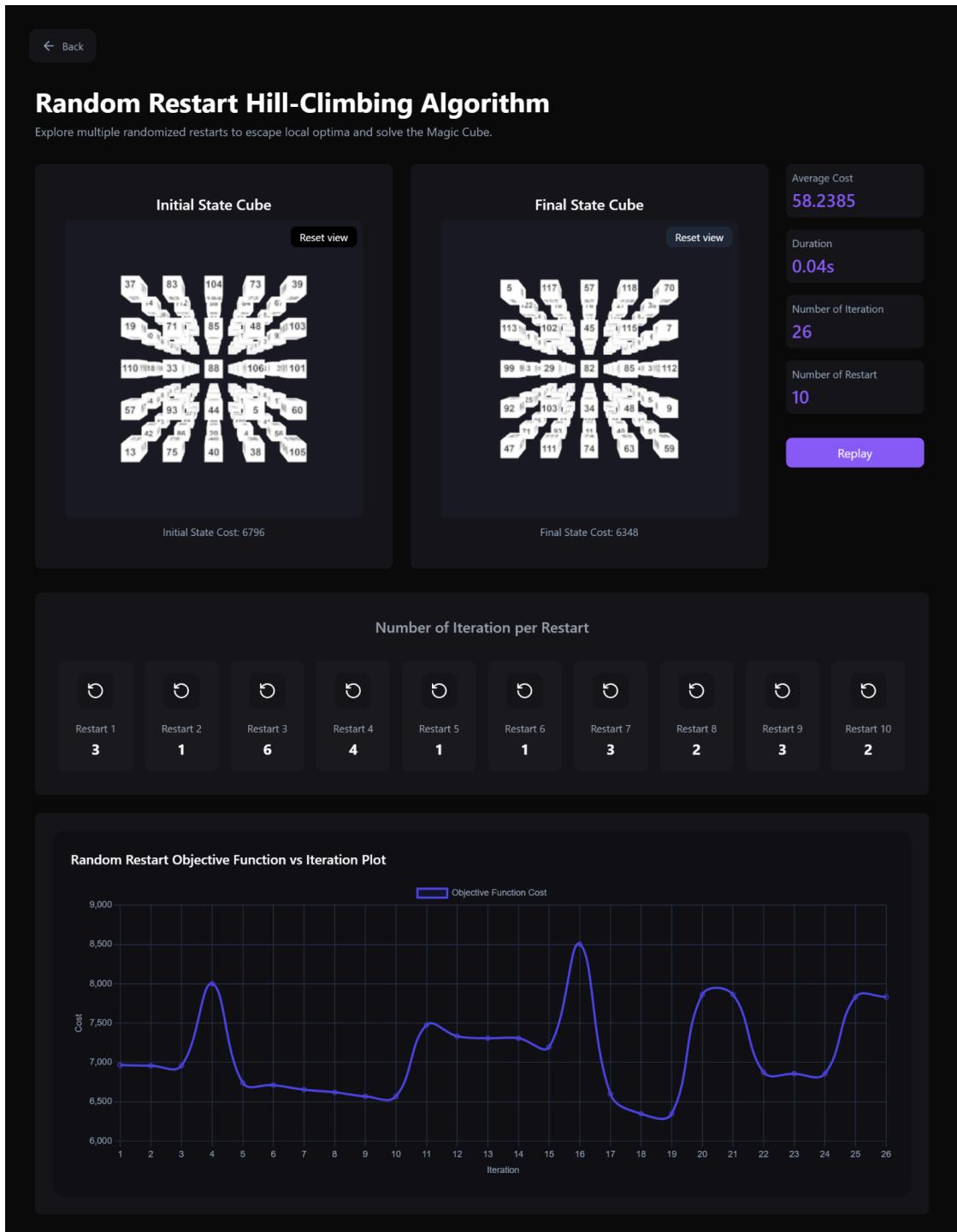
Pada algoritma *Stochastic Hill-climbing*, didapatkan *final cost* yaitu 1060 dengan *average cost* sebesar 9.7248. Seluruh proses ini membutuhkan waktu selama detik dan dengan total iterasi yaitu 10000.



Gambar 2.29 *Stochastic Hill-climbing* Eksperimen II

e. Random Restart Hill-climbing

Pada algoritma *Random Restart Hill-climbing*, didapatkan *final cost* yaitu 6348 dengan *average cost* sebesar 58.2385. Seluruh proses ini membutuhkan waktu selama 0.04 detik dan dengan total iterasi yaitu 26 dan total *restart* 10.

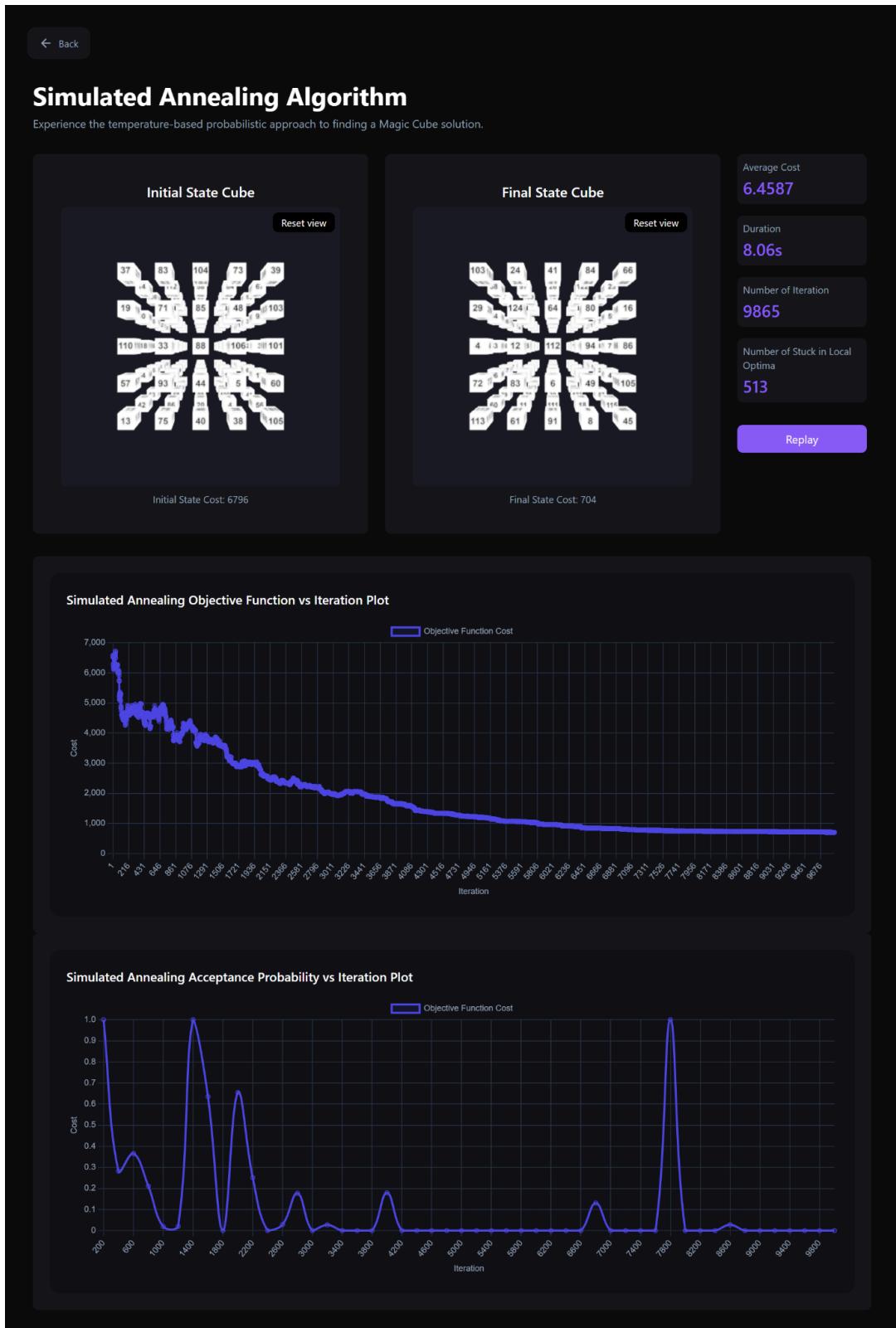


Gambar 2.30 Random Restart Hill-climbing Eksperimen II

f. Simulated Annealing

Pada algoritma *Simulated Annealing*, didapatkan *final cost* yaitu 704 dengan *average cost* sebesar 6.4587. Seluruh proses ini membutuhkan waktu selama

8.06 detik dan dengan total iterasi yaitu 9865 dan total *stuck* di lokal optimum yaitu 513.

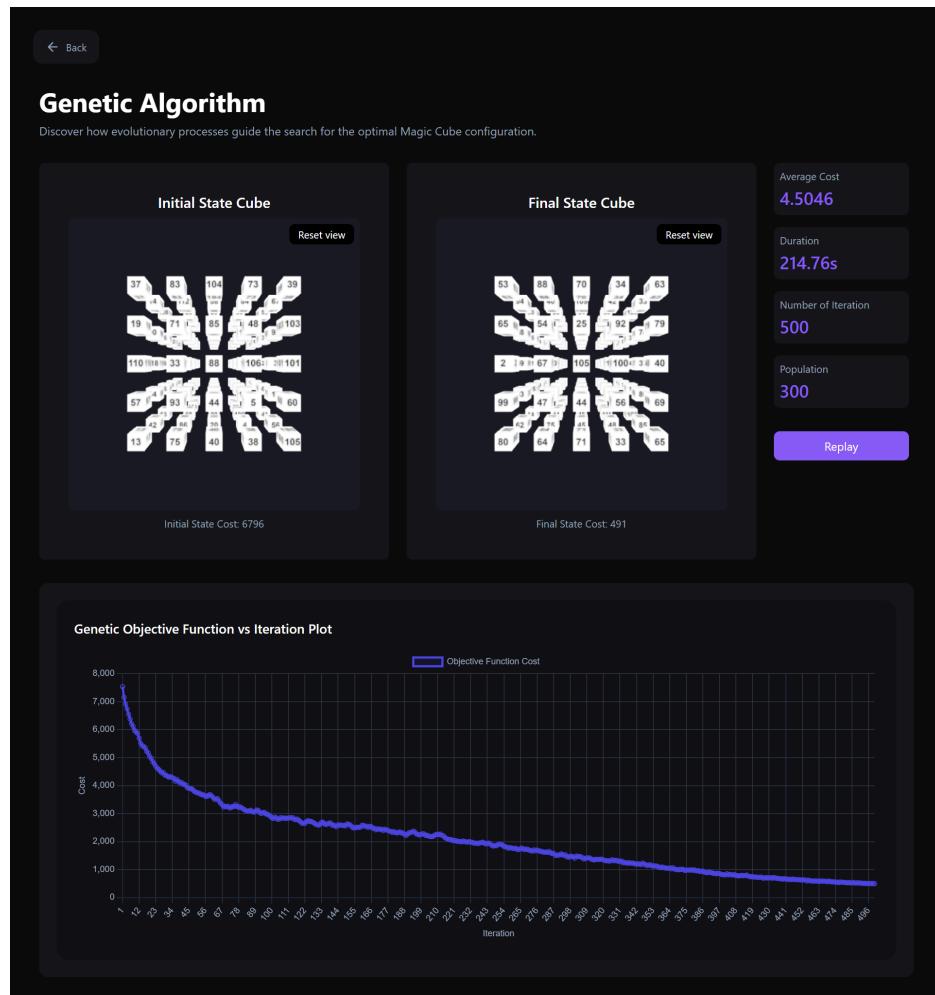


Gambar 2.31 Simulated Annealing Eksperimen II

g. Genetic Algorithm

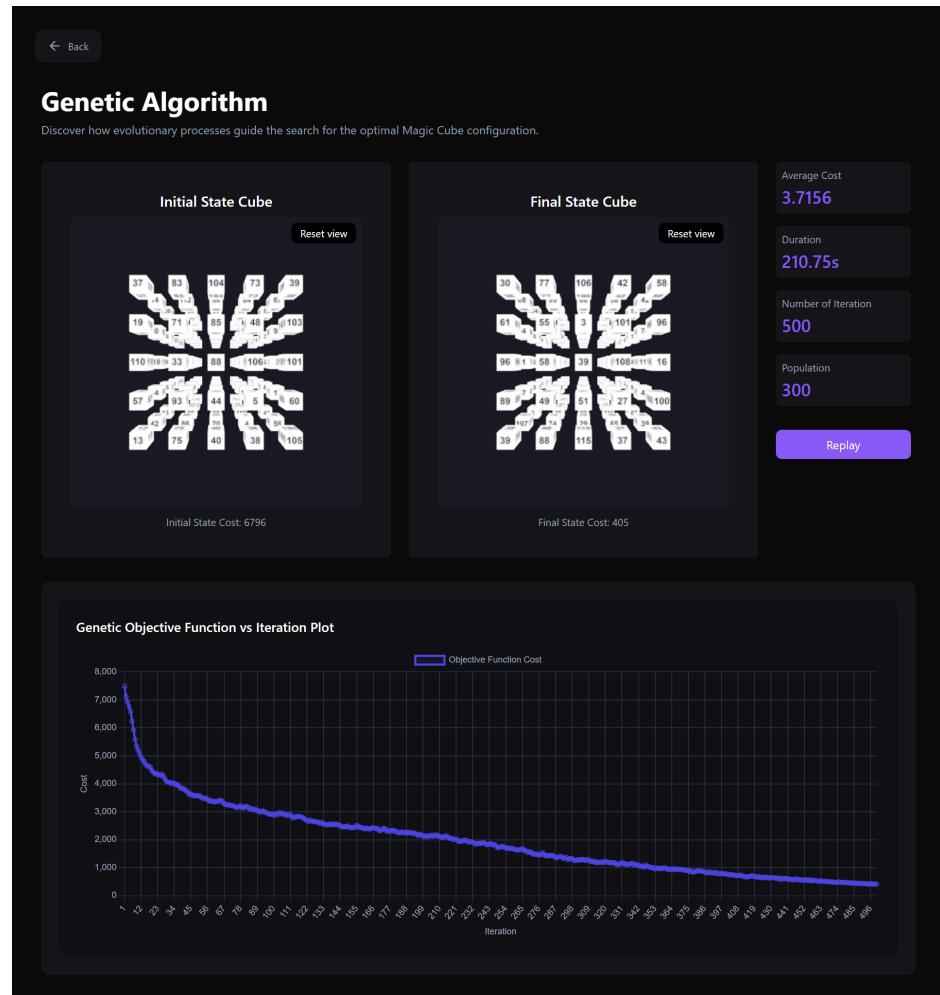
i. Population = 300 Iteration = 500

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama yaitu 491 dengan *average cost* sebesar 4.5046. Seluruh proses ini membutuhkan waktu selama 214.76 detik dan dengan total iterasi yaitu 500 dan jumlah populasi yaitu 300.



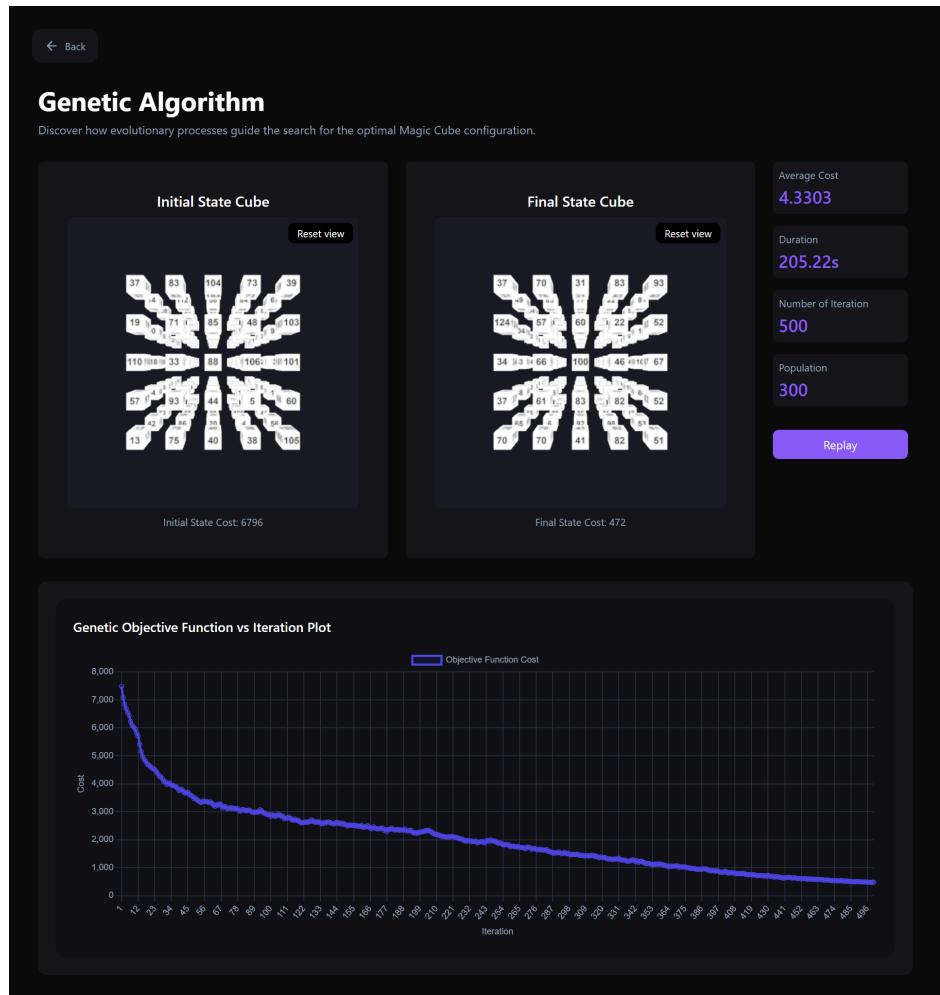
Gambar 2.32 *Genetic Algorithm* i (1) Eksperimen II

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 405 dengan *average cost* sebesar 3.7156. Seluruh proses ini membutuhkan waktu selama 210.75 detik dan dengan total iterasi yaitu 500 dan jumlah populasi yaitu 300.



Gambar 2.33 *Genetic Algorithm* i (2) Eksperimen II

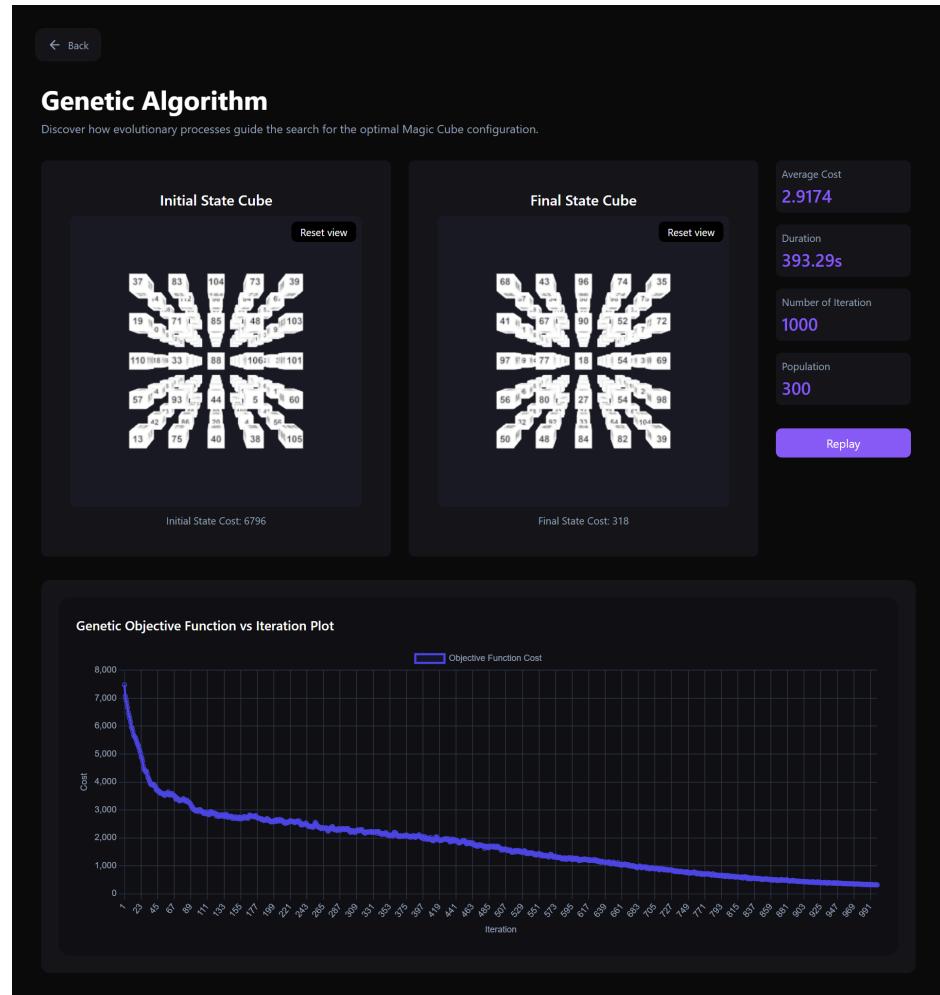
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 472 dengan *average cost* sebesar 4.3303. Seluruh proses ini membutuhkan waktu selama 205.22 detik dan dengan total iterasi yaitu 500 dan jumlah populasi yaitu 300.



Gambar 2.34 *Genetic Algorithm i (3) Eksperimen II*

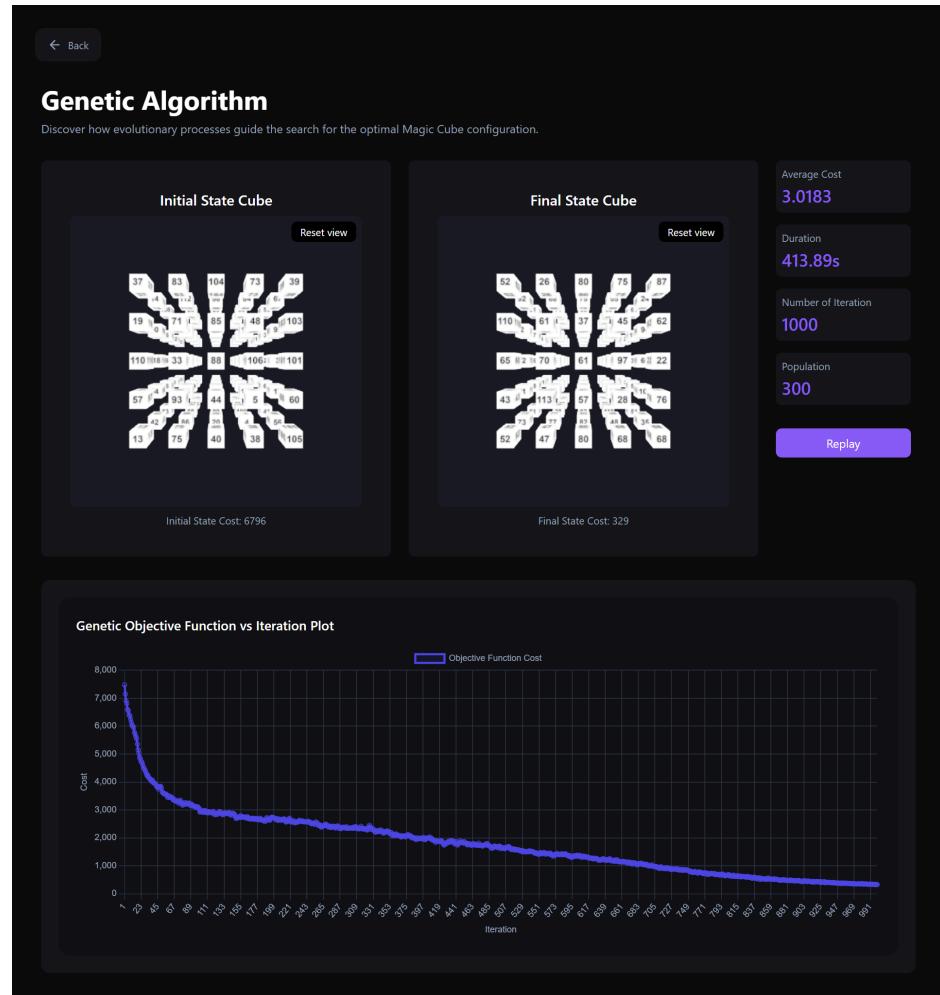
ii. Population = 300 Iteration = 1000

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama yaitu 318 dengan *average cost* sebesar 2.9174. Seluruh proses ini membutuhkan waktu selama 393.29 detik dan dengan total iterasi yaitu 1000 dan jumlah populasi yaitu 300.



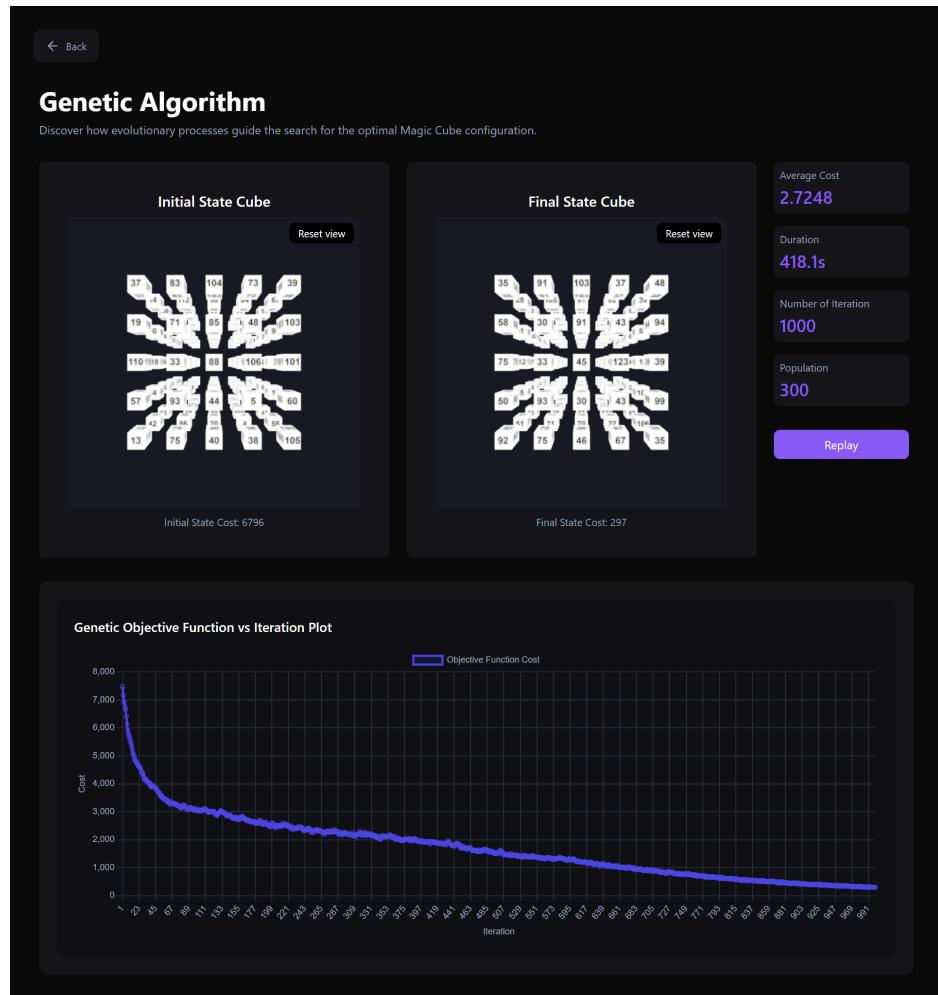
Gambar 2.35 *Genetic Algorithm* ii (1) Eksperimen II

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 329 dengan *average cost* sebesar 3.0183. Seluruh proses ini membutuhkan waktu selama 413.89 detik dan dengan total iterasi yaitu 1000 dan jumlah populasi yaitu 300.



Gambar 2.36 *Genetic Algorithm* ii (2) Eksperimen II

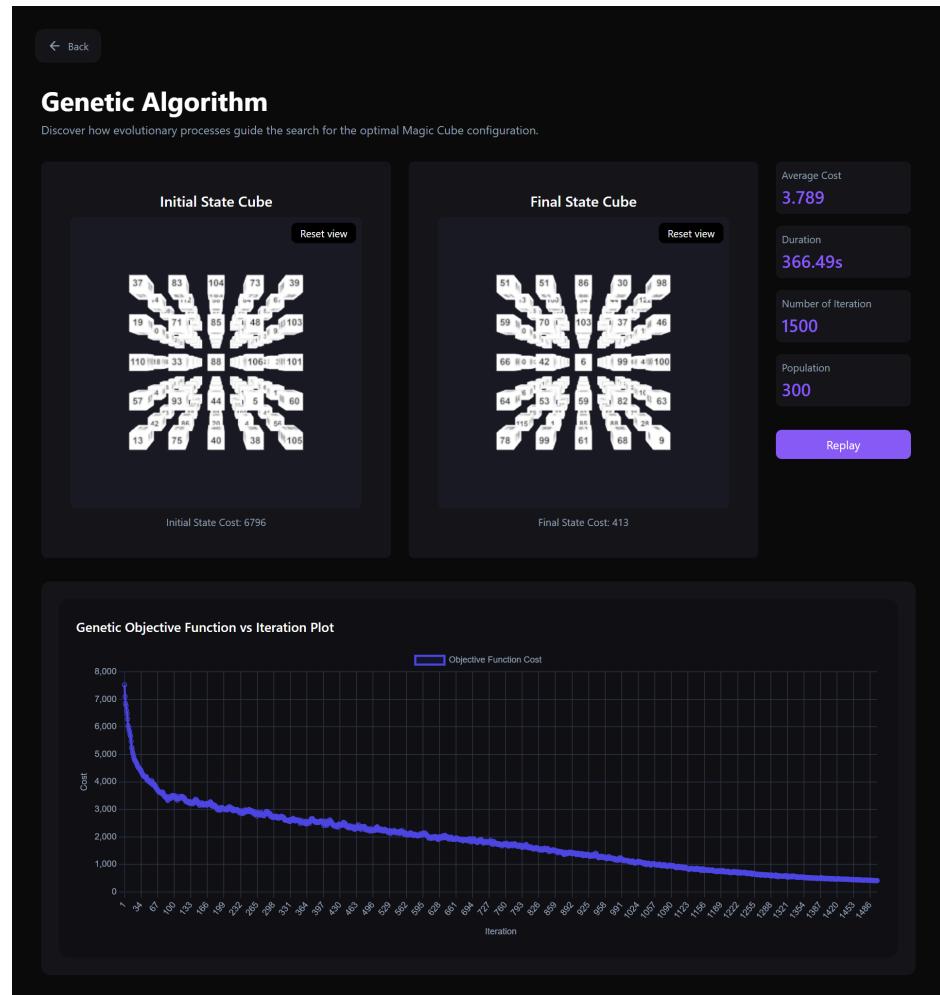
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 297 dengan *average cost* sebesar 2.7248. Seluruh proses ini membutuhkan waktu selama 418.1 detik dan dengan total iterasi yaitu 1000 dan jumlah populasi yaitu 300.



Gambar 2.37 *Genetic Algorithm* ii (3) Eksperimen II

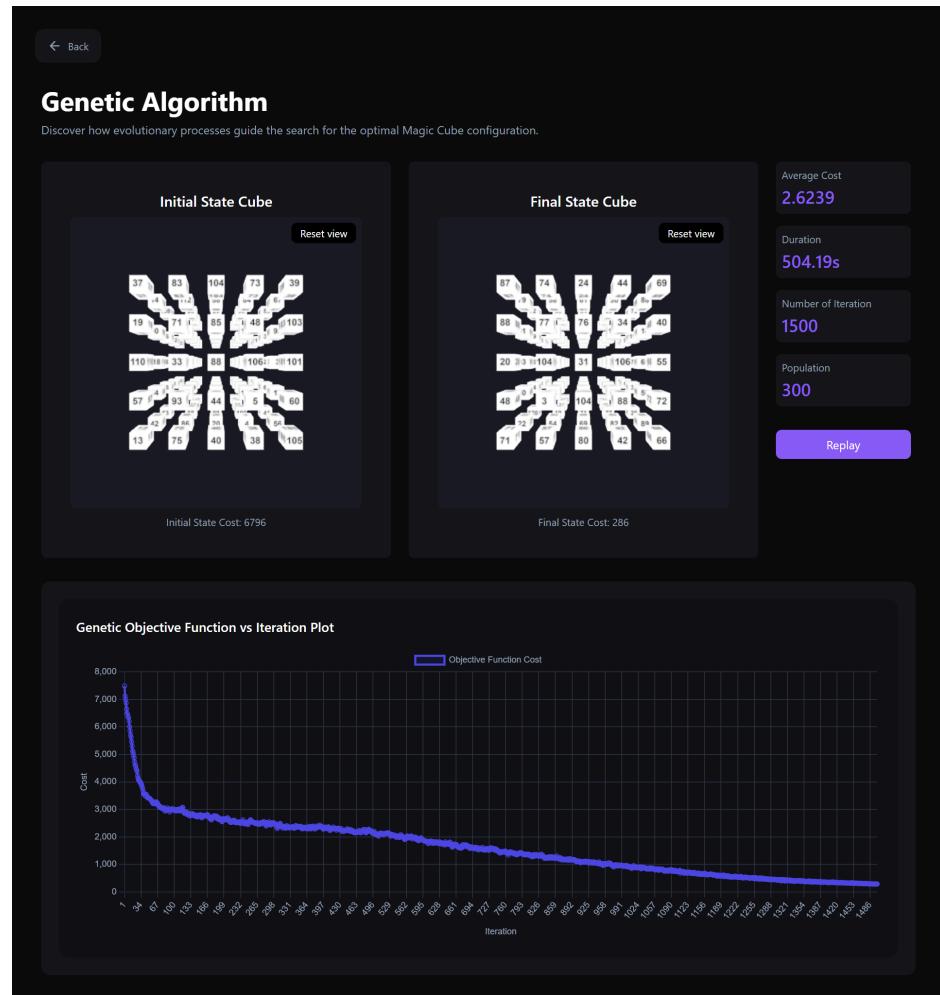
iii. Population = 300 Iteration = 1500

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 413 dengan *average cost* sebesar 3.789. Seluruh proses ini membutuhkan waktu selama 366.49 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 300.



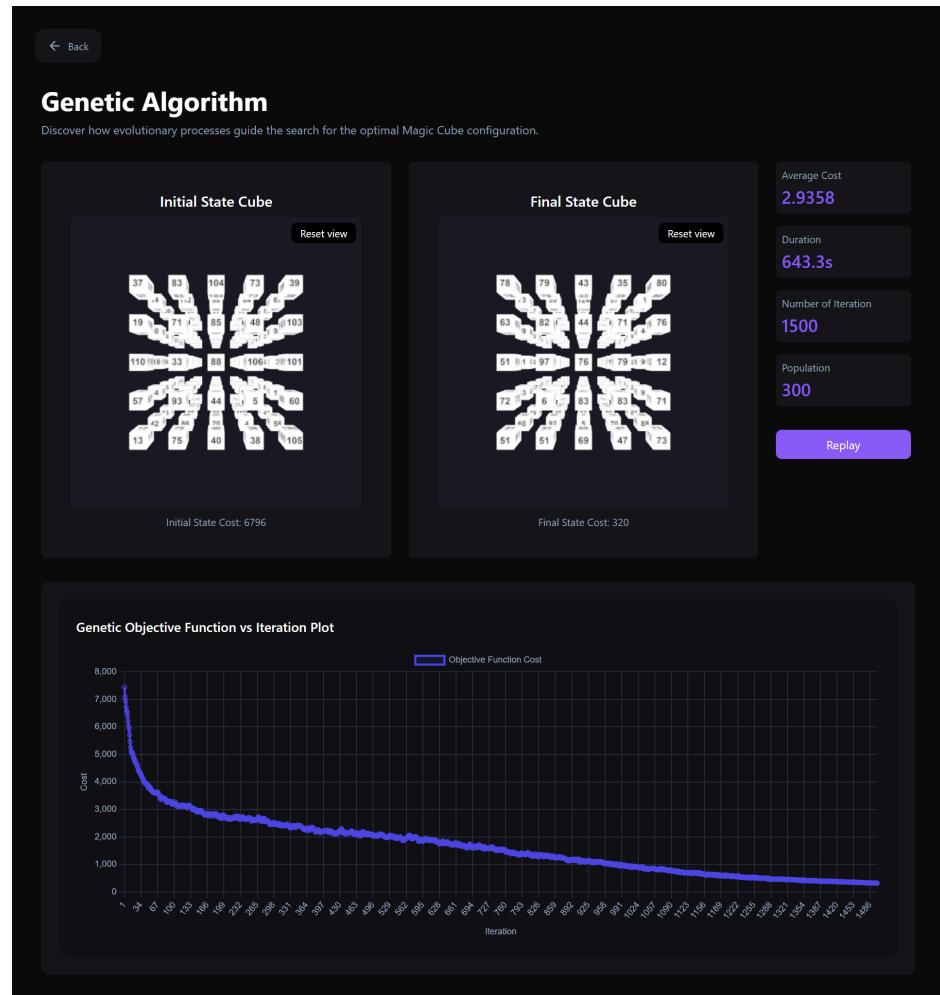
Gambar 2.38 *Genetic Algorithm* iii (1) Eksperimen III

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 286 dengan *average cost* sebesar 2.6239. Seluruh proses ini membutuhkan waktu selama 504.19 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 300.



Gambar 2.39 *Genetic Algorithm* iii (2) Eksperimen III

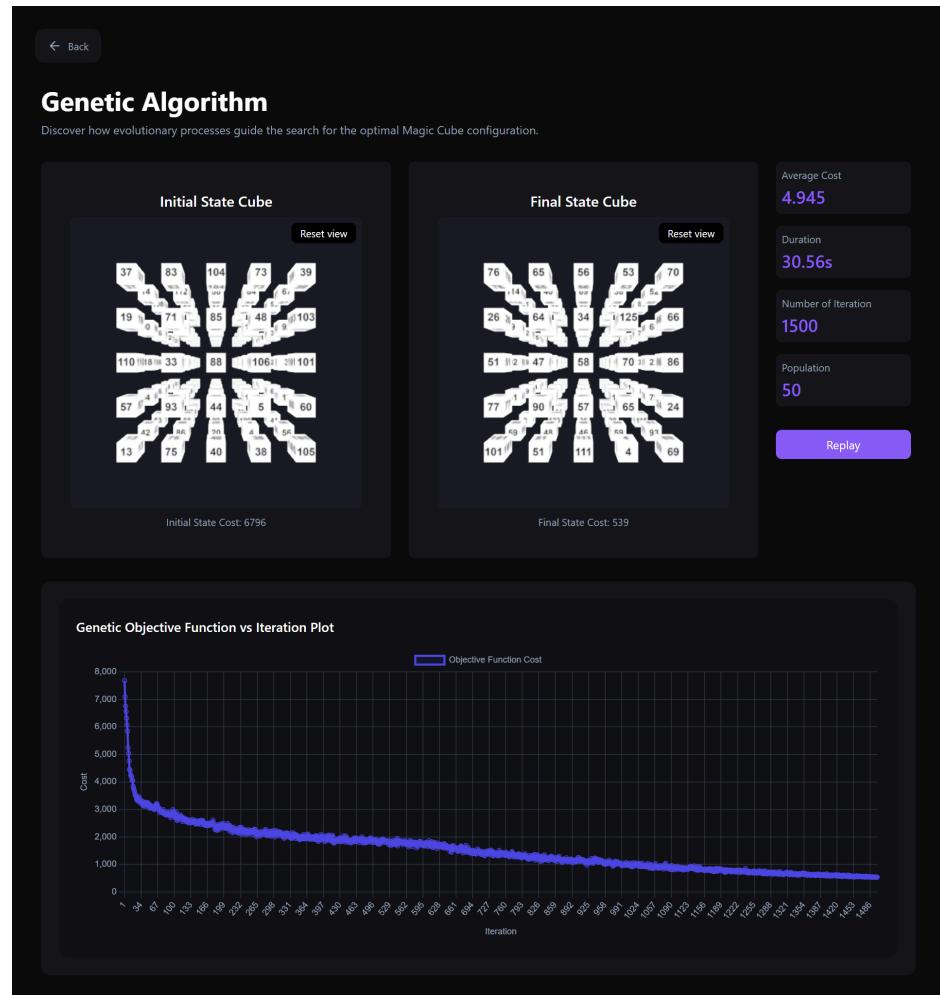
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 320 dengan *average cost* sebesar 2.9358. Seluruh proses ini membutuhkan waktu selama 643.3 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 300.



Gambar 2.40 *Genetic Algorithm* iii (3) Eksperimen III

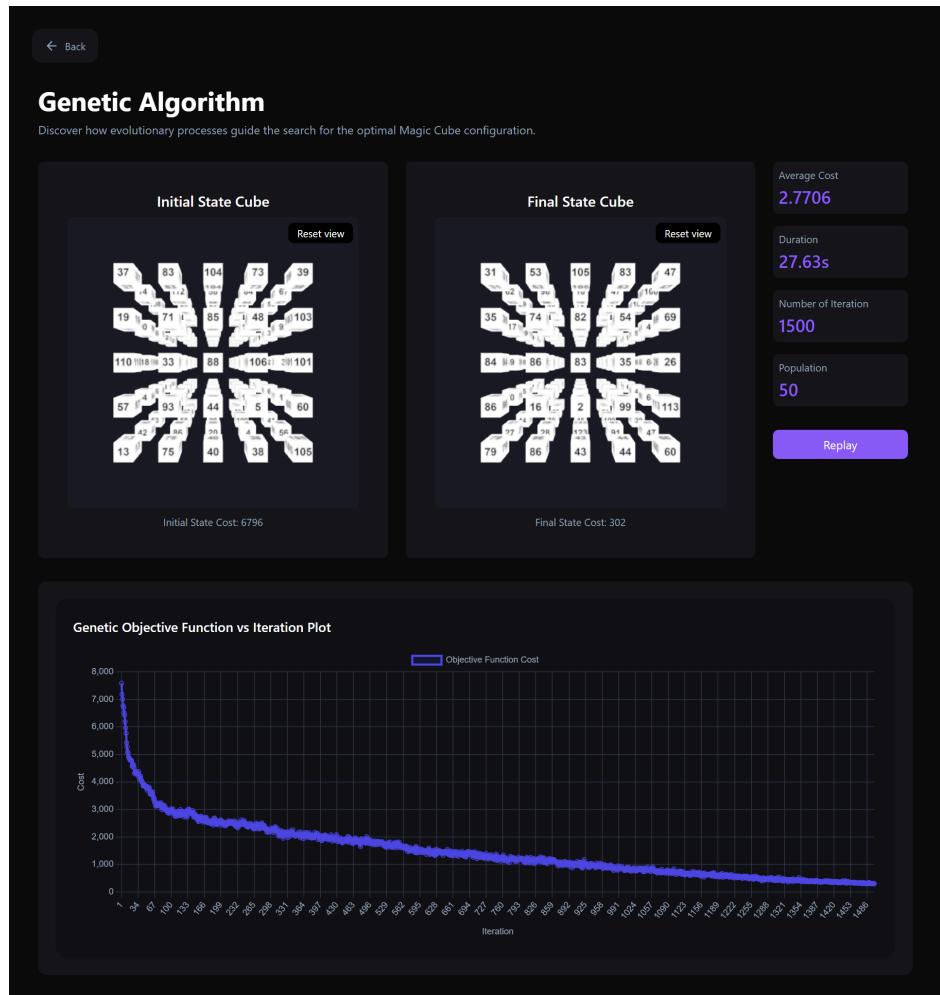
iv. **Iteration = 1500 Population = 50**

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama yaitu 539 dengan *average cost* sebesar 4.945. Seluruh proses ini membutuhkan waktu selama 30.56 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 50.



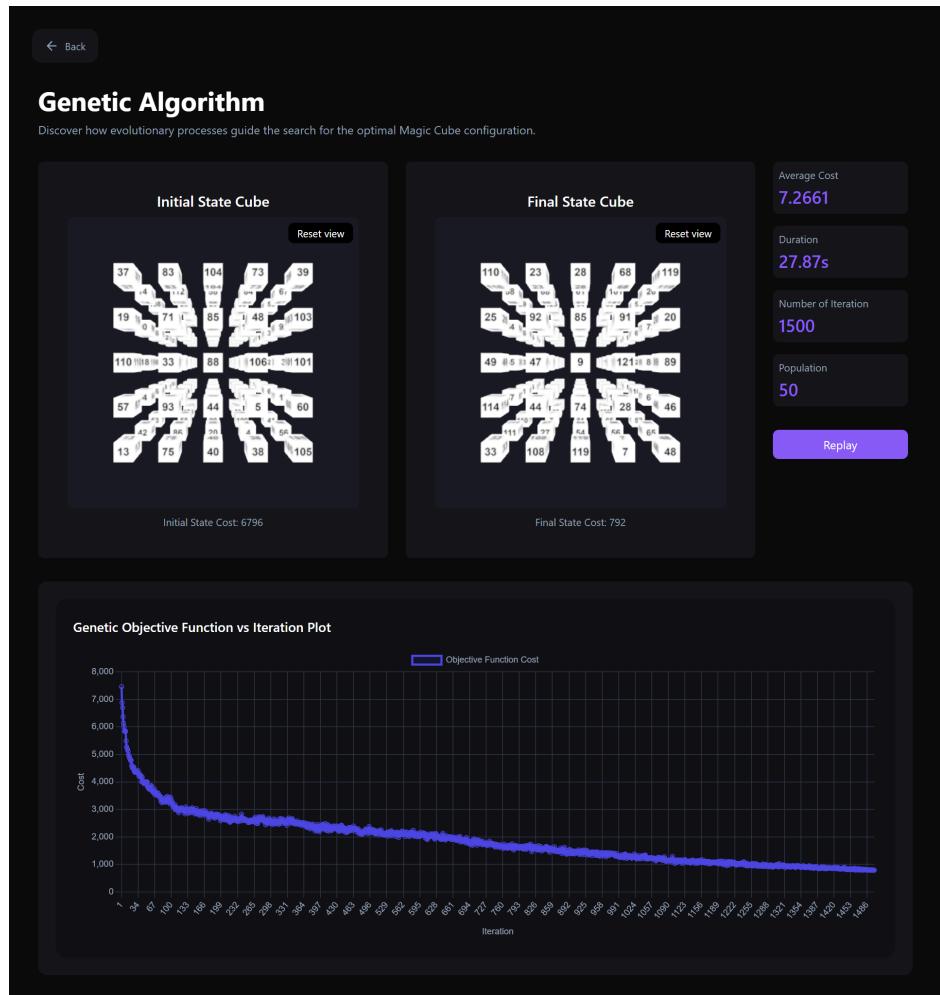
Gambar 2.41 *Genetic Algorithm iv (1)* Eksperimen I

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 302 dengan *average cost* sebesar 2.7706. Seluruh proses ini membutuhkan waktu selama 27.63 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 50.



Gambar 2.42 *Genetic Algorithm iv (2) Eksperimen I*

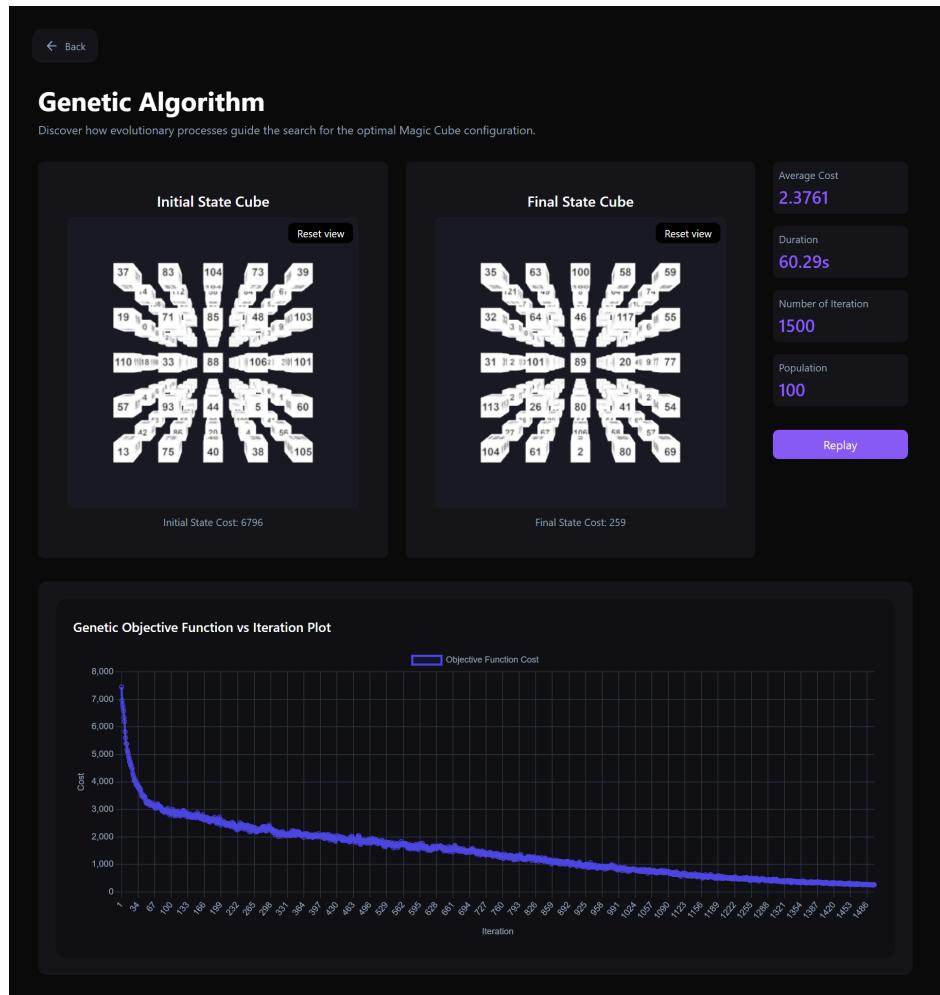
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 792 dengan *average cost* sebesar 7.2661. Seluruh proses ini membutuhkan waktu selama 27.87 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 50.



Gambar 2.43 *Genetic Algorithm iv (3) Eksperimen I*

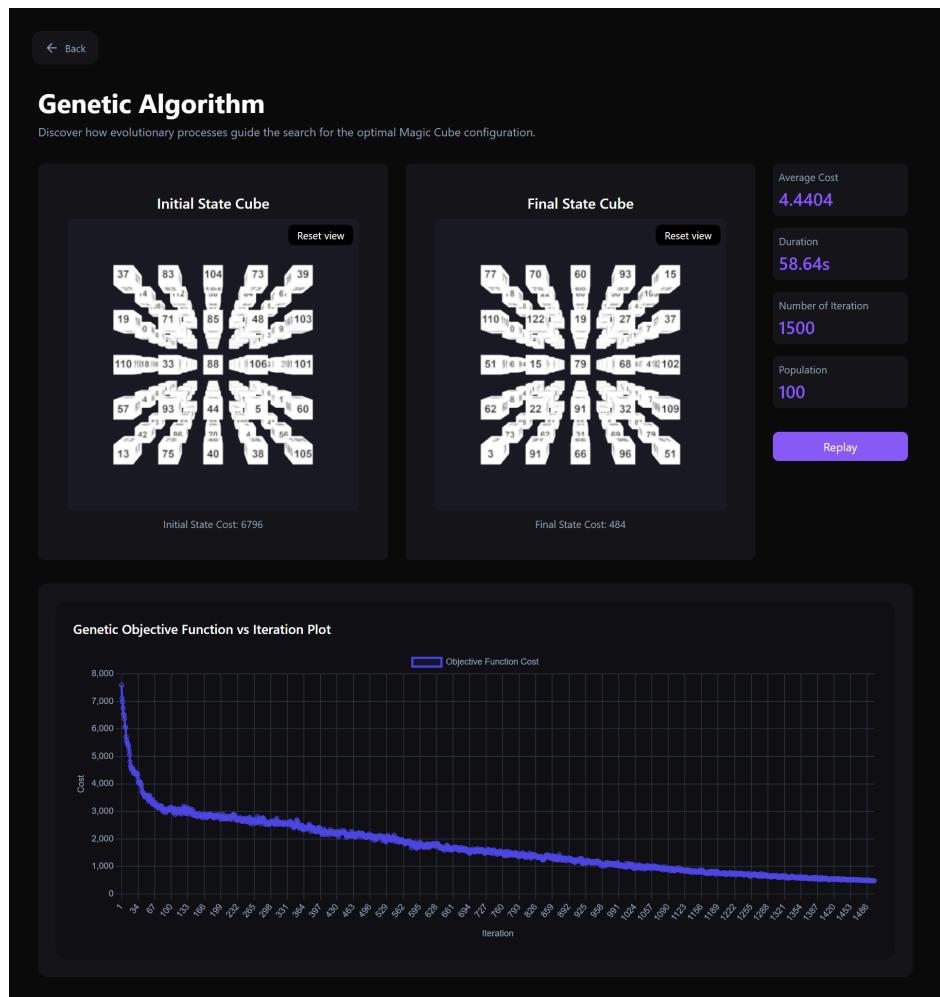
v. Iteration = 1500 Population = 100

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama yaitu 259 dengan *average cost* sebesar 2.3761. Seluruh proses ini membutuhkan waktu selama 60.29 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 100.



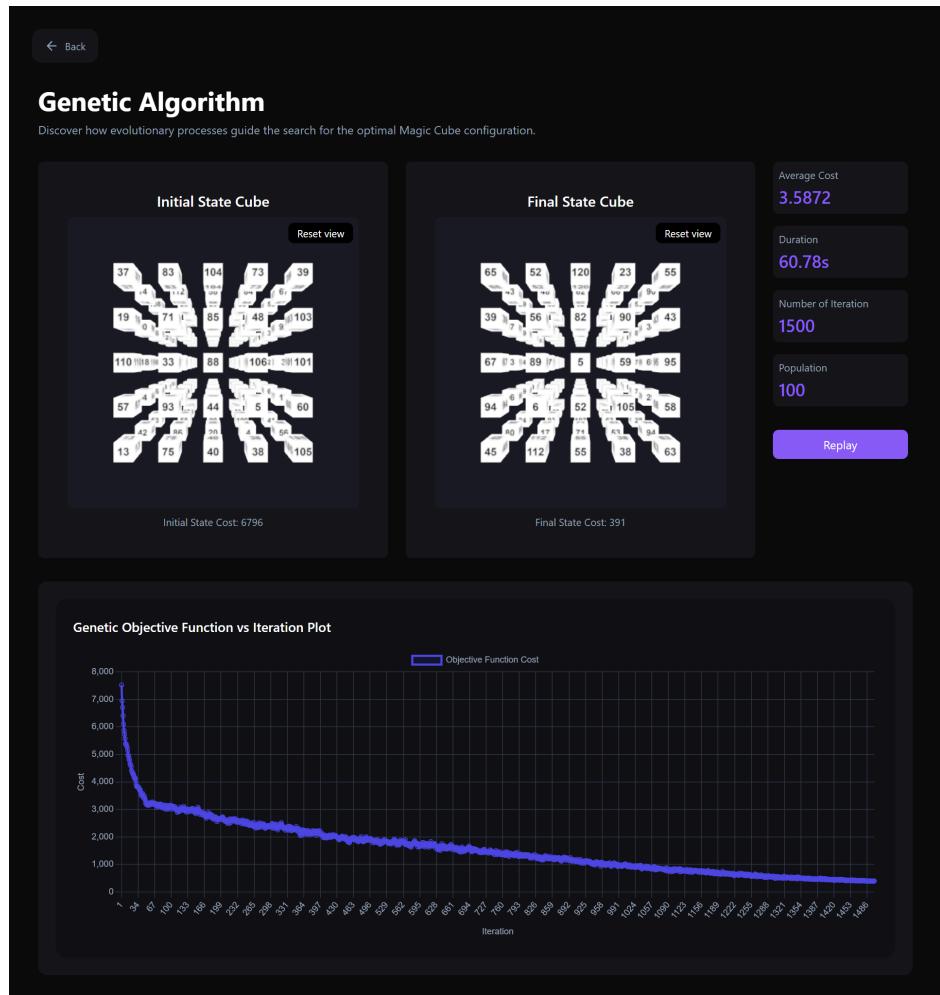
Gambar 2.44 *Genetic Algorithm* v (1) Eksperimen I

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 484 dengan *average cost* sebesar 4.4404. Seluruh proses ini membutuhkan waktu selama 58.64 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 100.



Gambar 2.45 *Genetic Algorithm* v (2) Eksperimen I

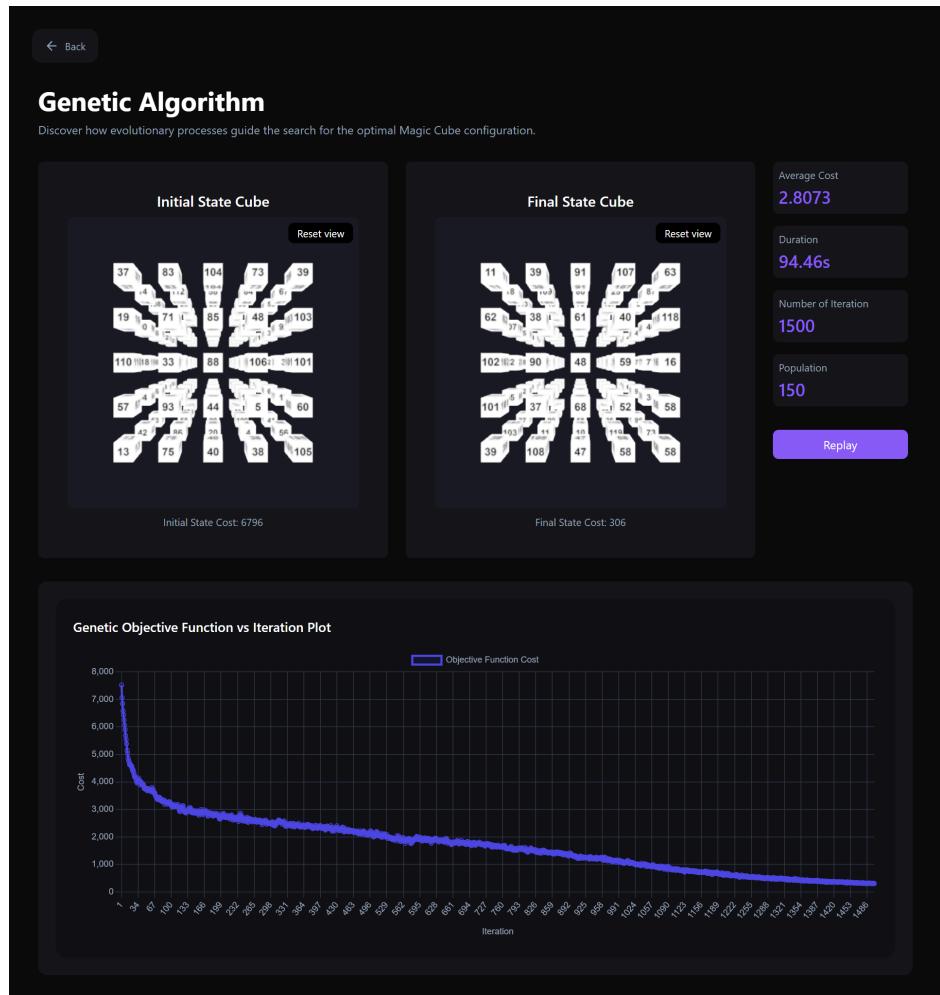
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 391 dengan *average cost* sebesar 3.5872. Seluruh proses ini membutuhkan waktu selama 60.78 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 100.



Gambar 2.46 *Genetic Algorithm* v (3) Eksperimen I

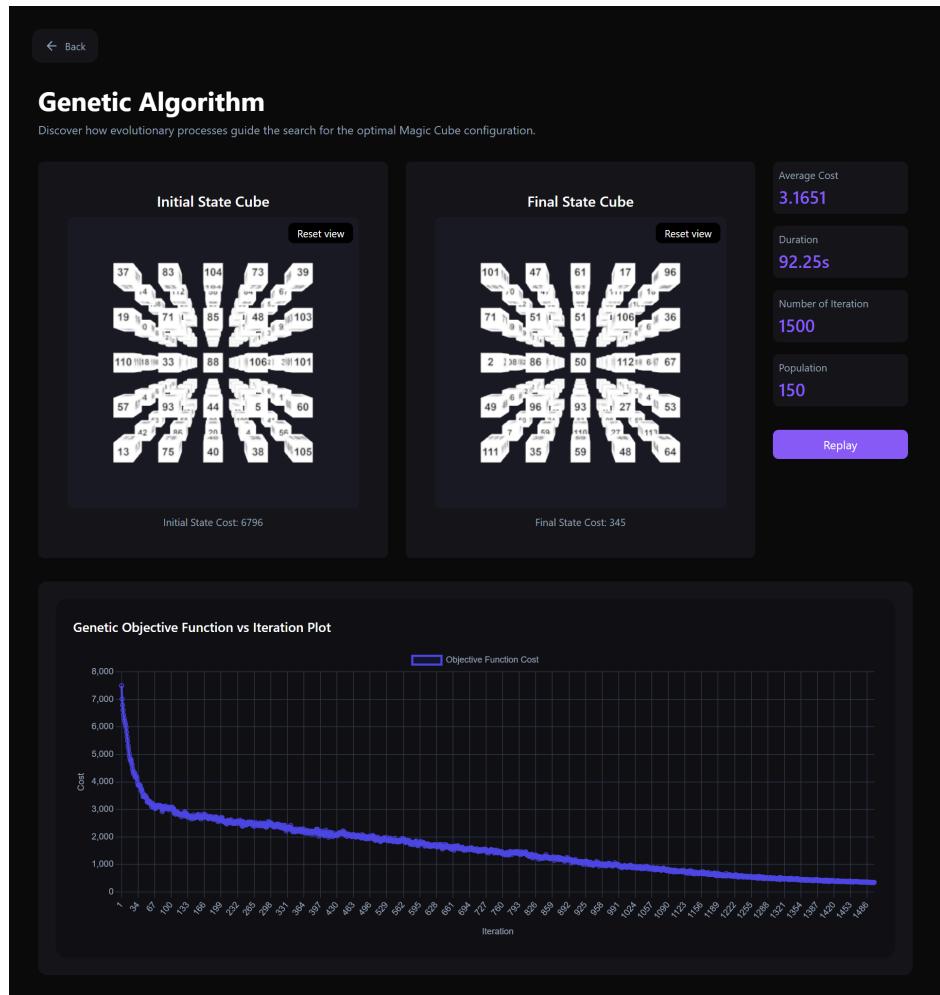
vi. **Iteration = 1500 Population = 150**

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama yaitu 306 dengan *average cost* sebesar 2.8073. Seluruh proses ini membutuhkan waktu selama 94.46 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 150.



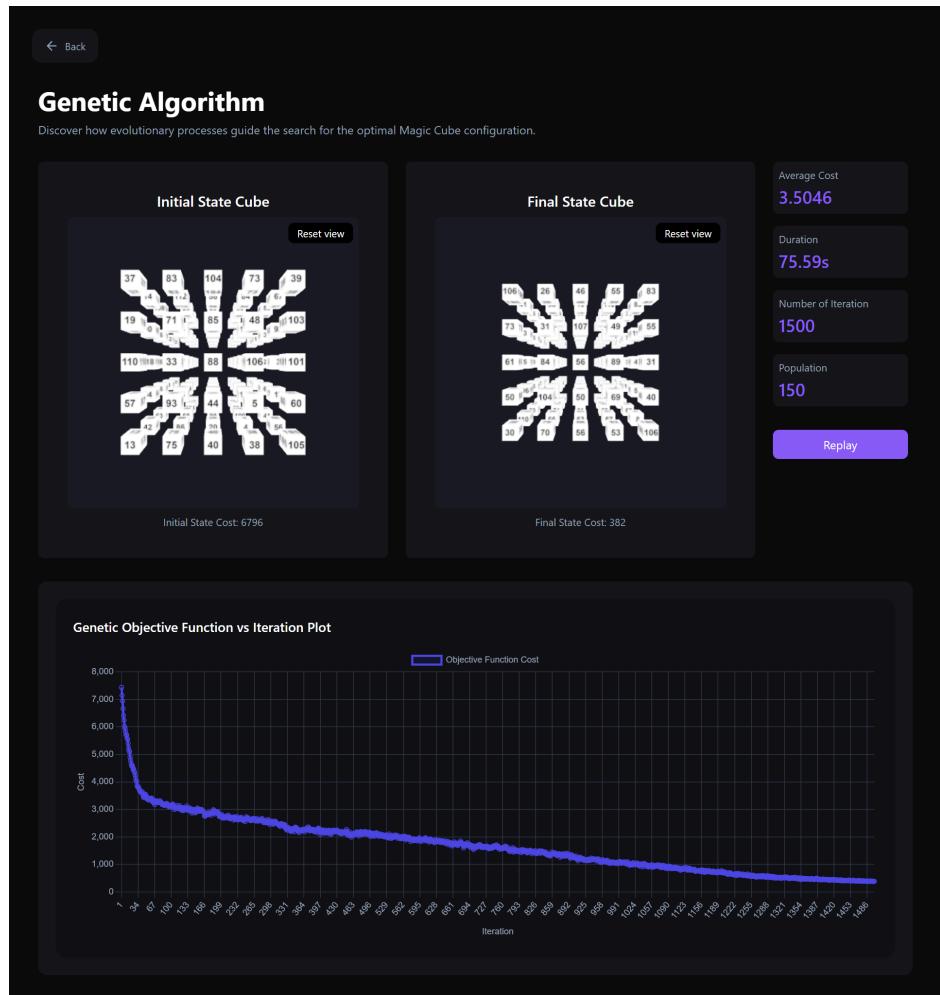
Gambar 2.47 *Genetic Algorithm* vi (1) Eksperimen II

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 345 dengan *average cost* sebesar 3.1651. Seluruh proses ini membutuhkan waktu selama 92.25 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 150.



Gambar 2.48 *Genetic Algorithm* vi (2) Eksperimen II

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 382 dengan *average cost* sebesar 3.5046. Seluruh proses ini membutuhkan waktu selama 75.59 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 150.

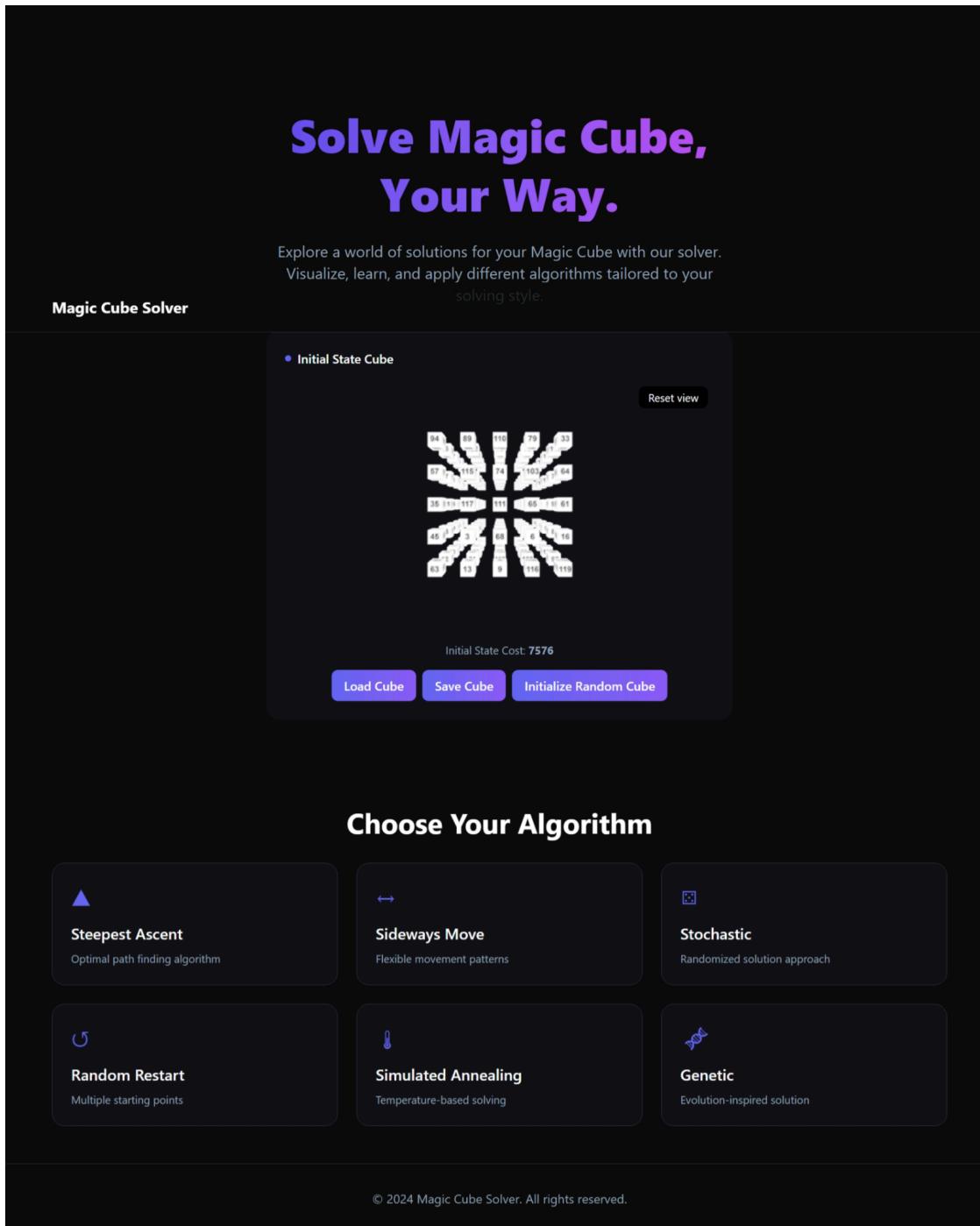


Gambar 2.49 *Genetic Algorithm* vi (3) Eksperimen II

3. Eksperimen III

a. Inisialisasi Cube

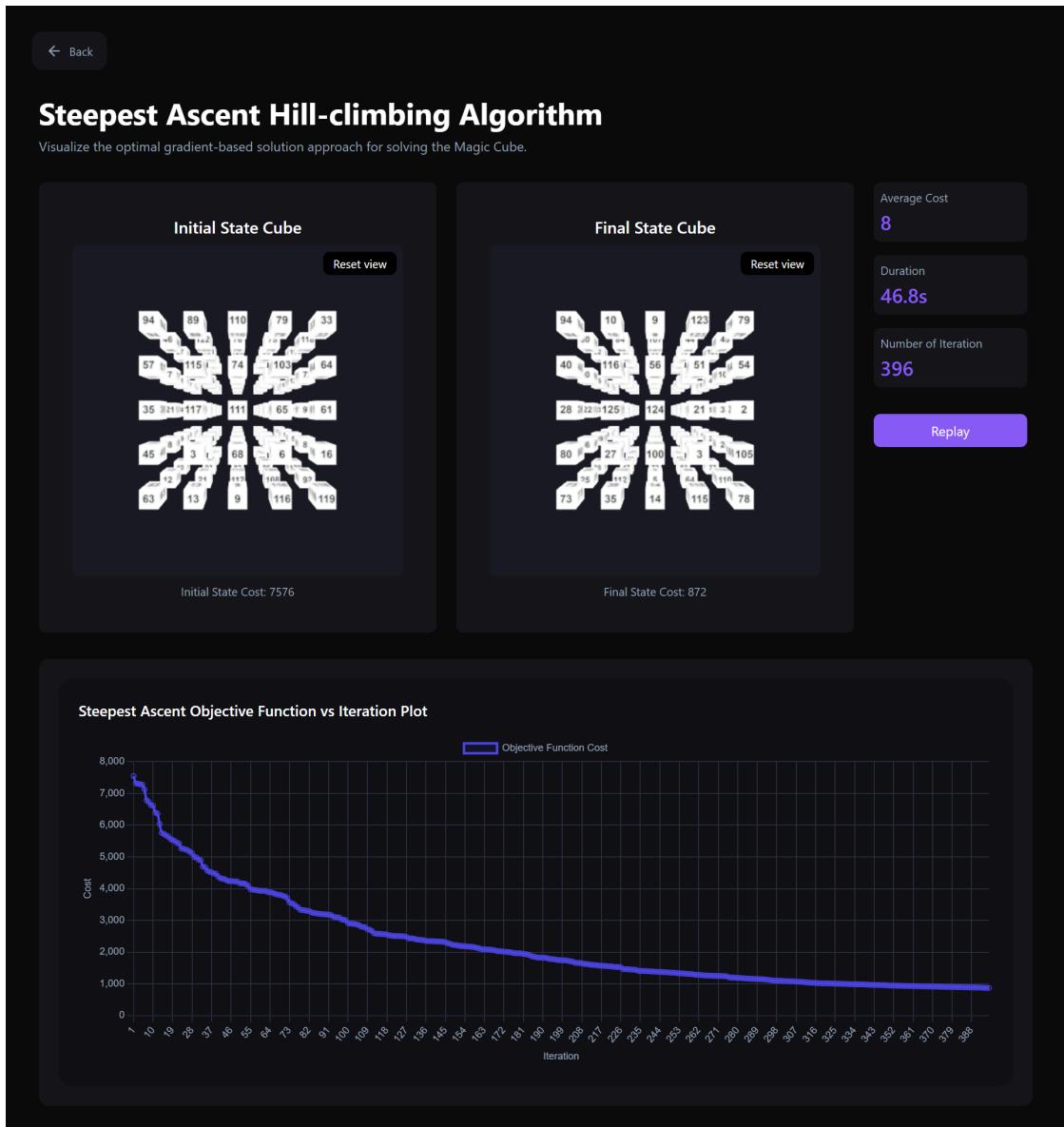
Pada eksperimen ketiga, setelah dilakukan inisialisasi *random cube*, didapatkan bahwa *initial state cost* kubus adalah 7576. Kubus ini akan digunakan pada seluruh algoritma untuk menentukan seberapa dekat algoritma tersebut mendekati global optima.



Gambar 2.50 Inisialisasi Cube III

b. Steepest Ascent Hill-climbing

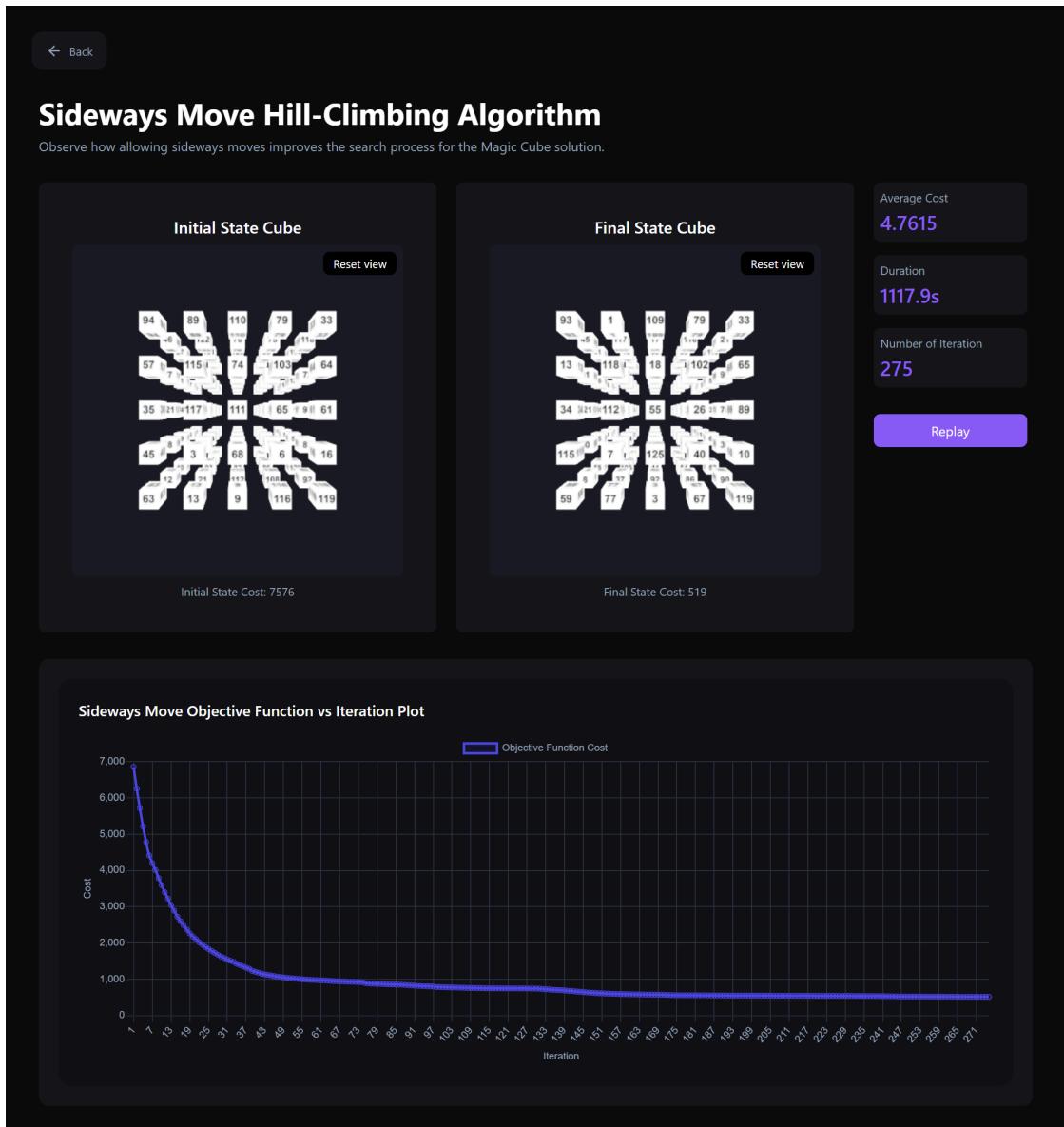
Pada algoritma *Steepest Ascent Hill-climbing*, didapatkan *final cost* yaitu 872 dengan *average cost* sebesar 8. Seluruh proses ini membutuhkan waktu selama 46.8s detik dan dengan total iterasi yaitu 396.



Gambar 2.51 Steepest Ascent Hill-climbing Eksperimen III

c. Hill-climbing with Sideways Move

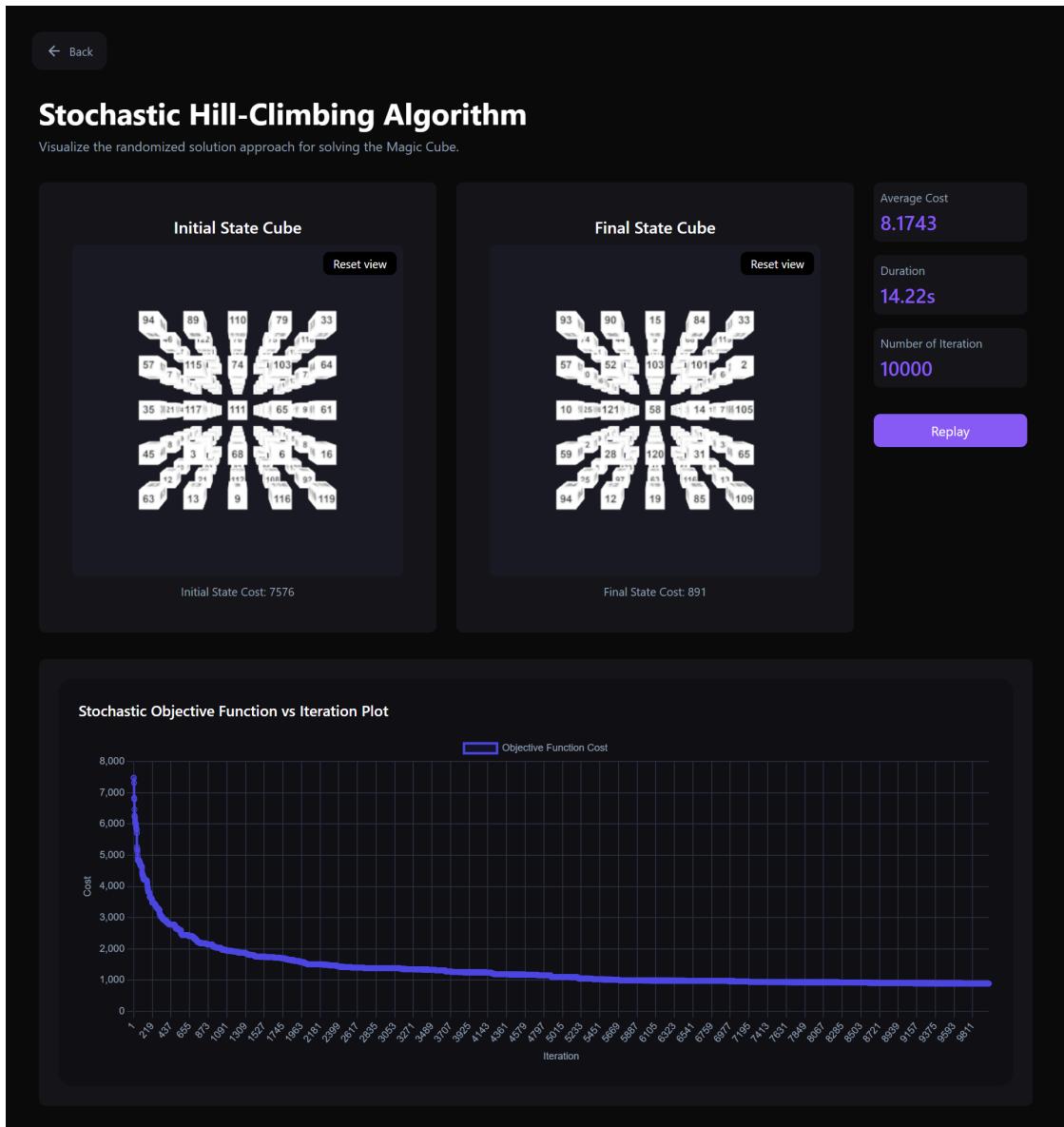
Pada algoritma *Hill-climbing with Sideways Move*, didapatkan *final cost* yaitu 519 dengan *average cost* sebesar 4.7615. Seluruh proses ini membutuhkan waktu selama 1117.9 detik dan dengan total iterasi yaitu 275.



Gambar 2.52 *Hill-climbing with Sideways Move Eksperimen III*

d. Stochastic Hill-climbing

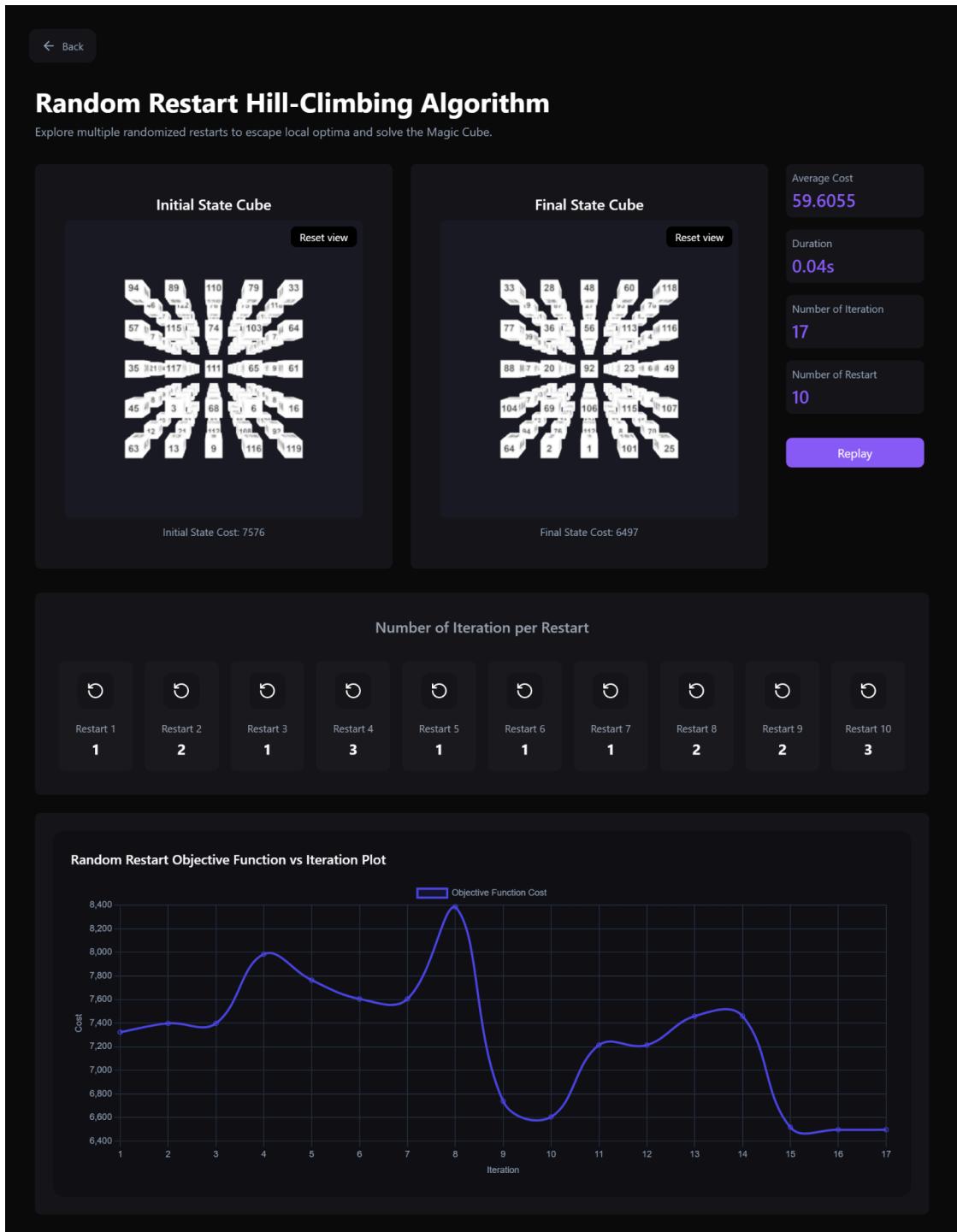
Pada algoritma *Stochastic Hill-climbing*, didapatkan *final cost* yaitu 891 dengan *average cost* sebesar 8.1743. Seluruh proses ini membutuhkan waktu selama 14.22 detik dan dengan total iterasi yaitu 10000.



Gambar 2.53 Stochastic Hill-climbing Eksperimen III

e. Random Restart Hill-climbing

Pada algoritma *Random Restart Hill-climbing*, didapatkan *final cost* yaitu 6497 dengan *average cost* sebesar 59.6055. Seluruh proses ini membutuhkan waktu selama 0.04 detik dan dengan total iterasi 17 dan total *restart* 10.

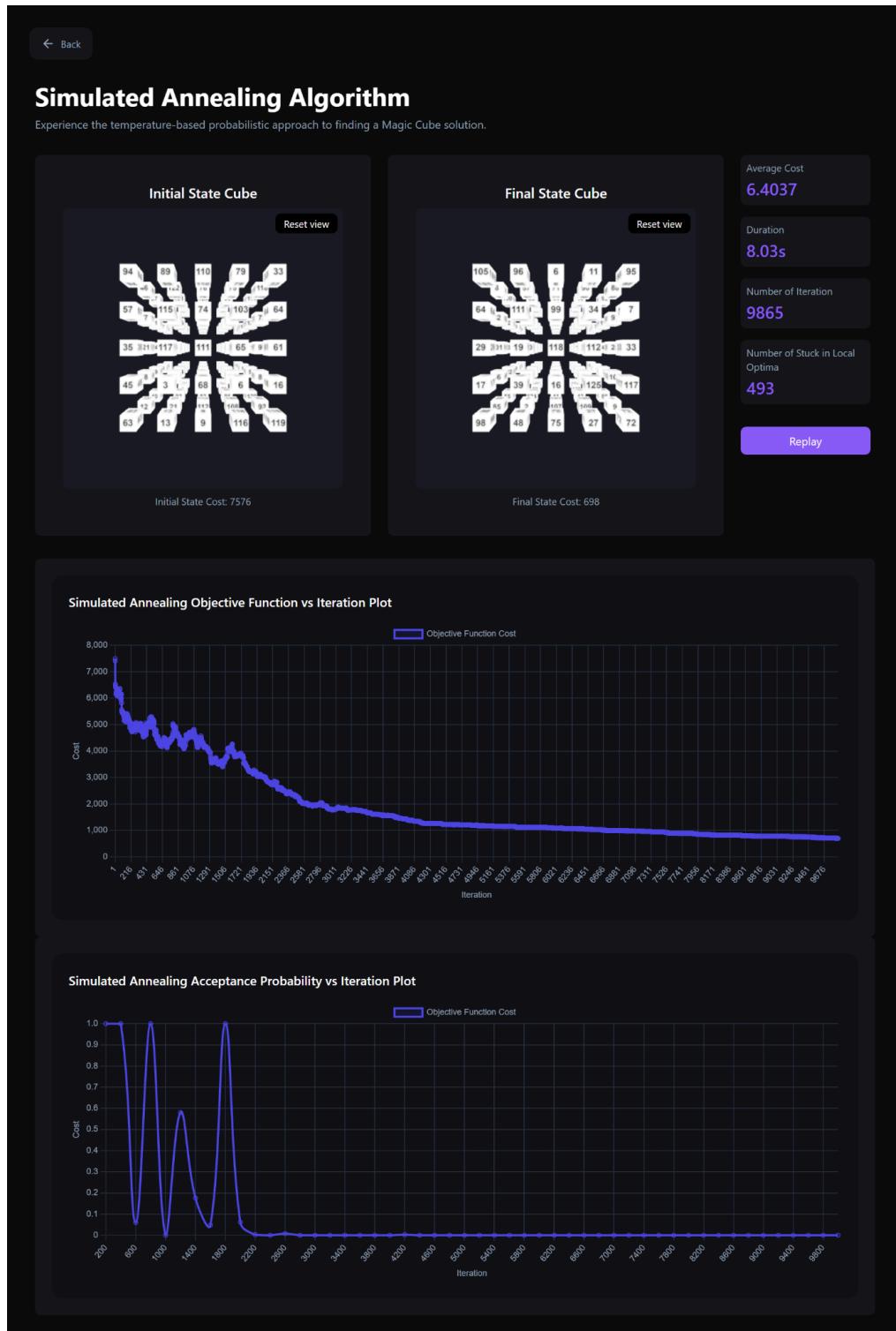


Gambar 2.54 Random Restart Hill-climbing Eksperimen II

f. Simulated Annealing

Pada algoritma *Simulated Annealing*, didapatkan *final cost* yaitu 698 dengan *average cost* sebesar 6.4037. Seluruh proses ini membutuhkan waktu selama

8.03 detik dan dengan total iterasi yaitu 9865 dan total stuck di lokal optimum yaitu 493.

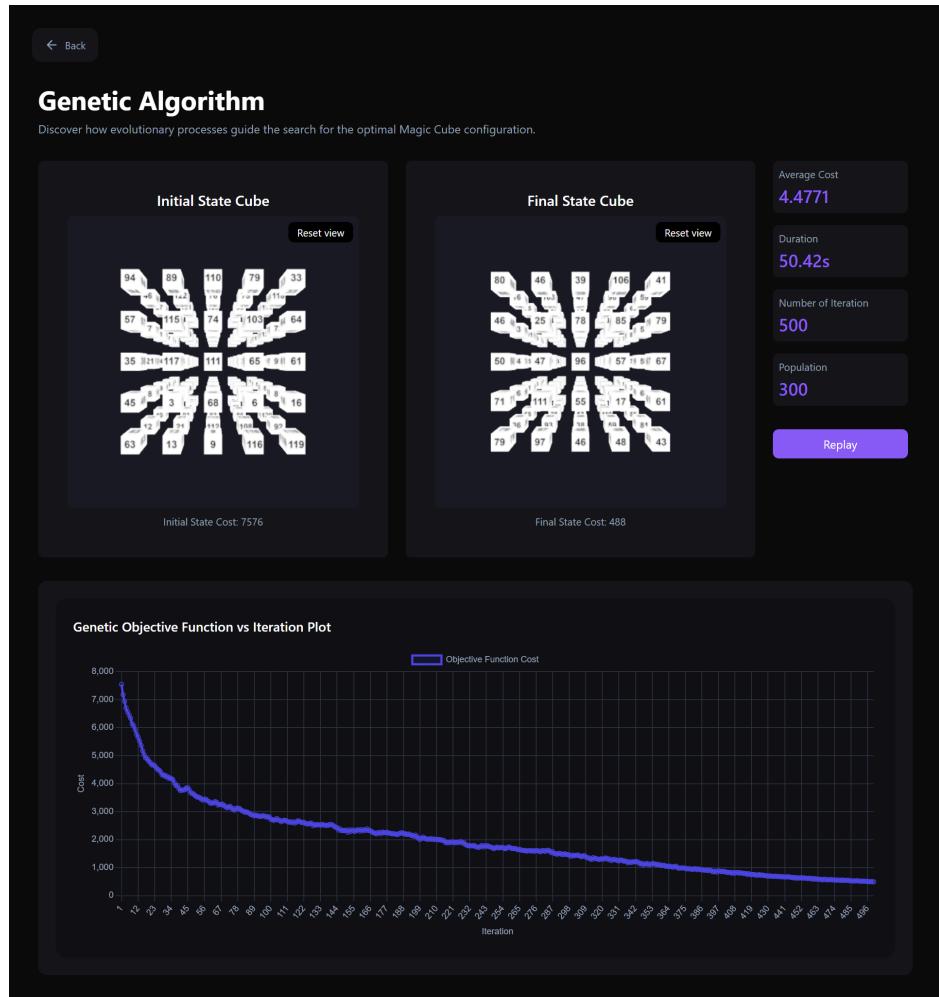


Gambar 2.55 Simulated Annealing Eksperimen III

g. Genetic Algorithm

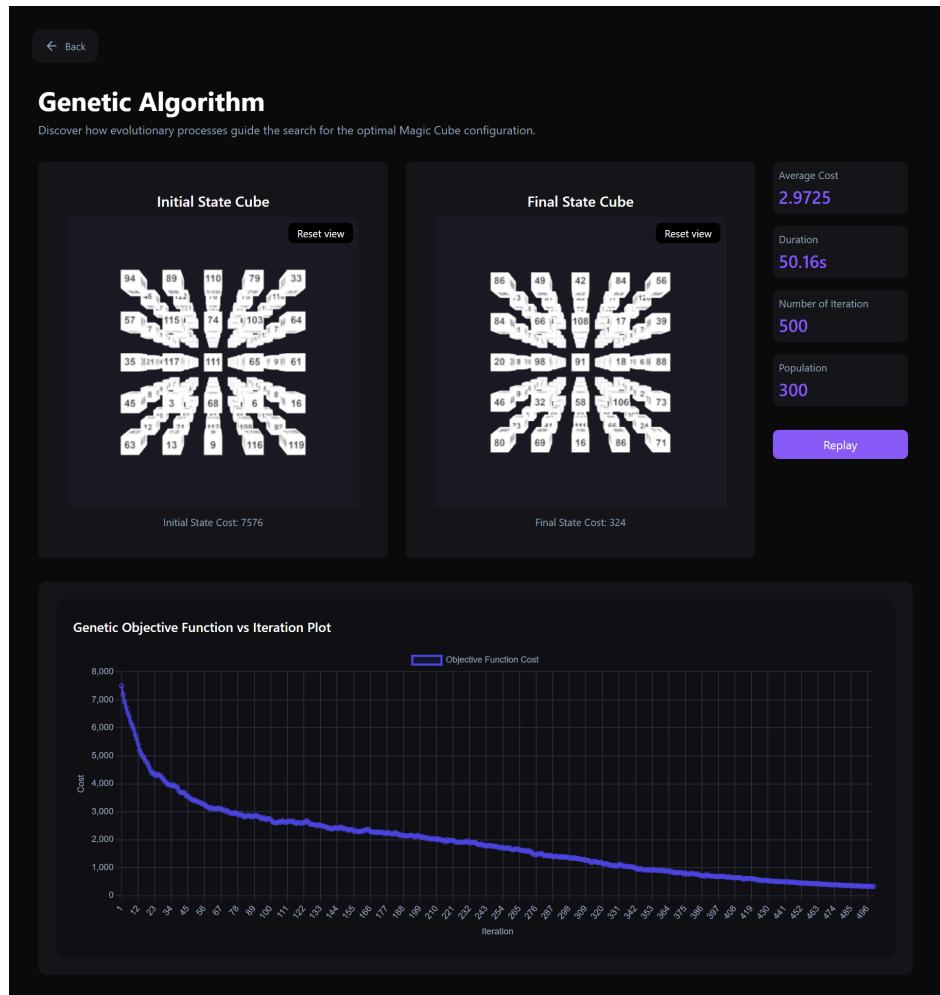
i. Population = 300 Iteration = 500

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama yaitu 488 dengan *average cost* sebesar 4.4771. Seluruh proses ini membutuhkan waktu selama 50.42 detik dan dengan total iterasi yaitu 500 dan jumlah populasi yaitu 300.



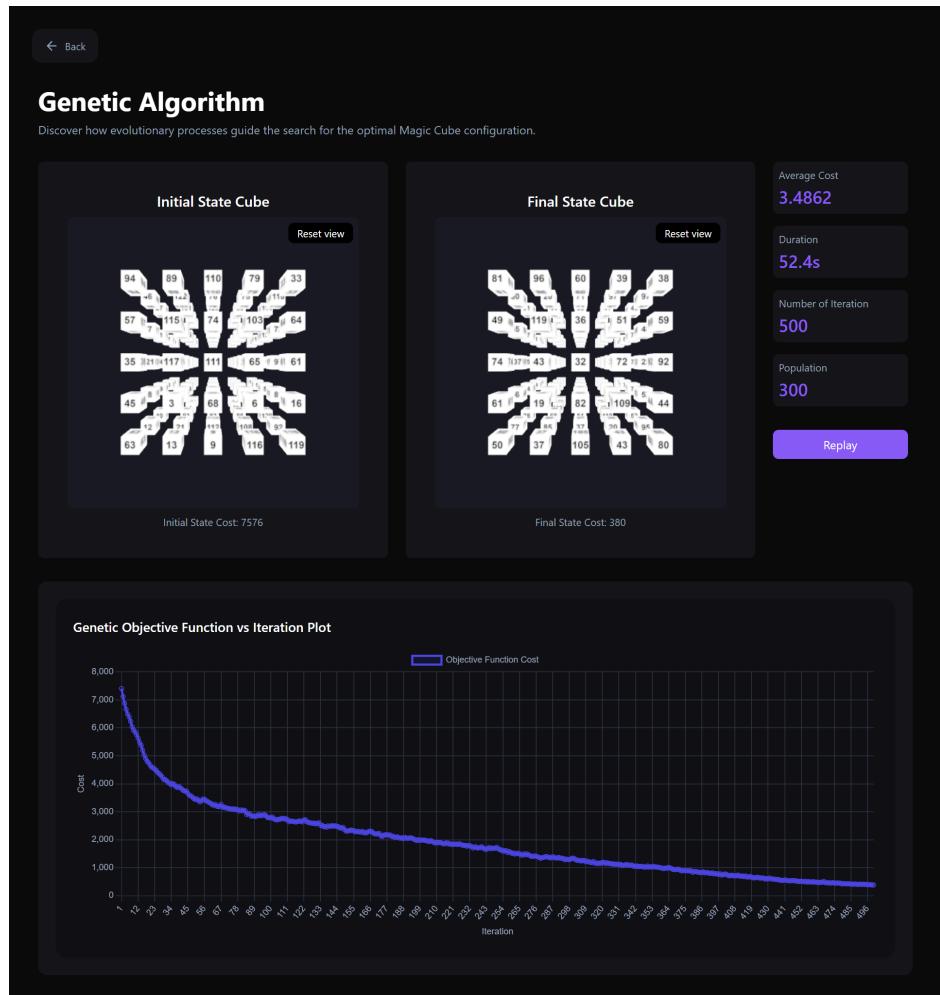
Gambar 2.56 *Genetic Algorithm* i (1) Eksperimen III

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 324 dengan *average cost* sebesar 2.9725. Seluruh proses ini membutuhkan waktu selama 50.16 detik dan dengan total iterasi yaitu 500 dan jumlah populasi yaitu 300.



Gambar 2.57 *Genetic Algorithm i (2) Eksperimen III*

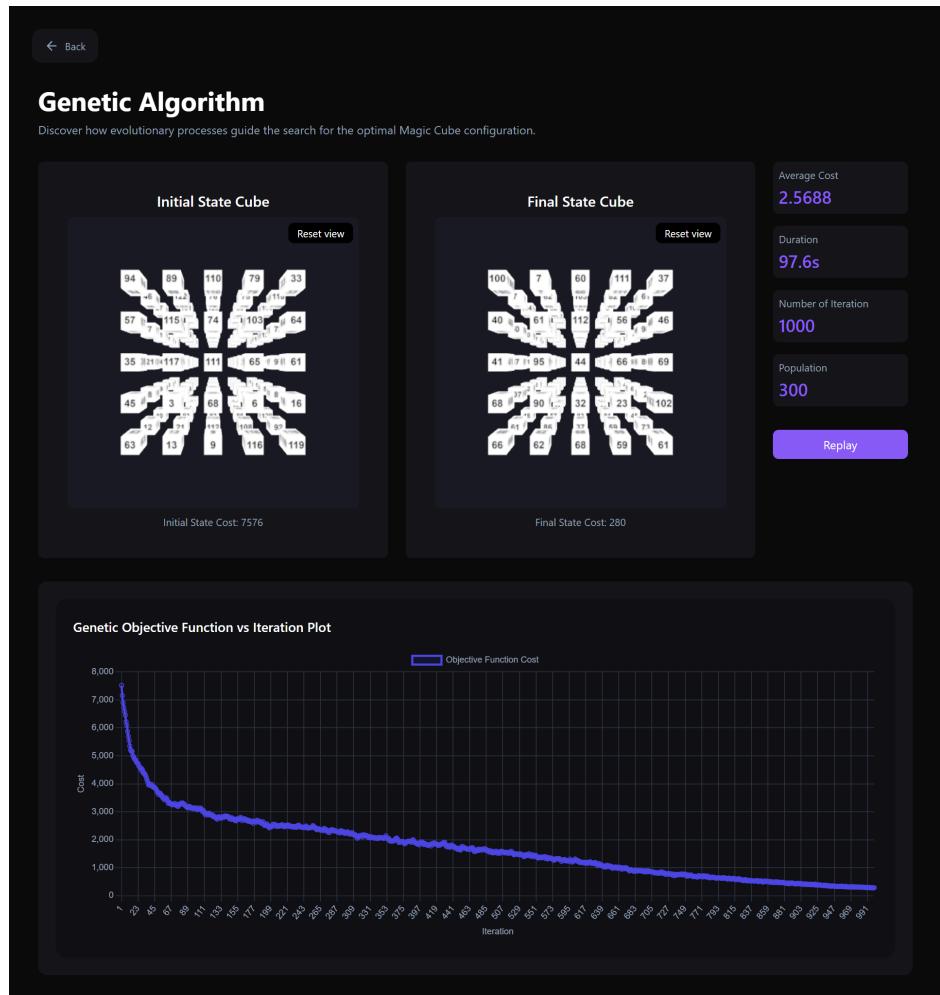
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 380 dengan *average cost* sebesar 3.4862. Seluruh proses ini membutuhkan waktu selama 52.4 detik dan dengan total iterasi yaitu 500 dan jumlah populasi yaitu 300.



Gambar 2.58 *Genetic Algorithm i (3) Eksperimen III*

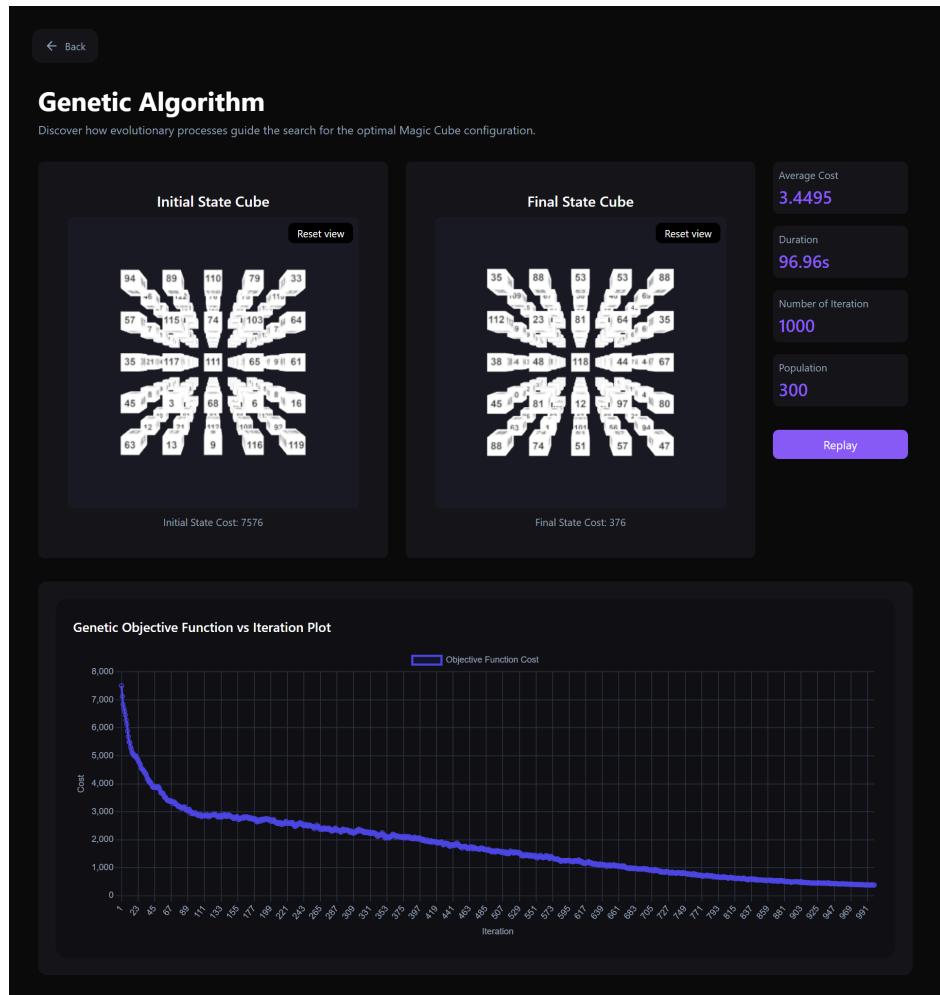
ii. Population = 300 Iteration = 1000

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama yaitu 280 dengan *average cost* sebesar 2.5688. Seluruh proses ini membutuhkan waktu selama 97.6 detik dan dengan total iterasi yaitu 1000 dan jumlah populasi yaitu 300.



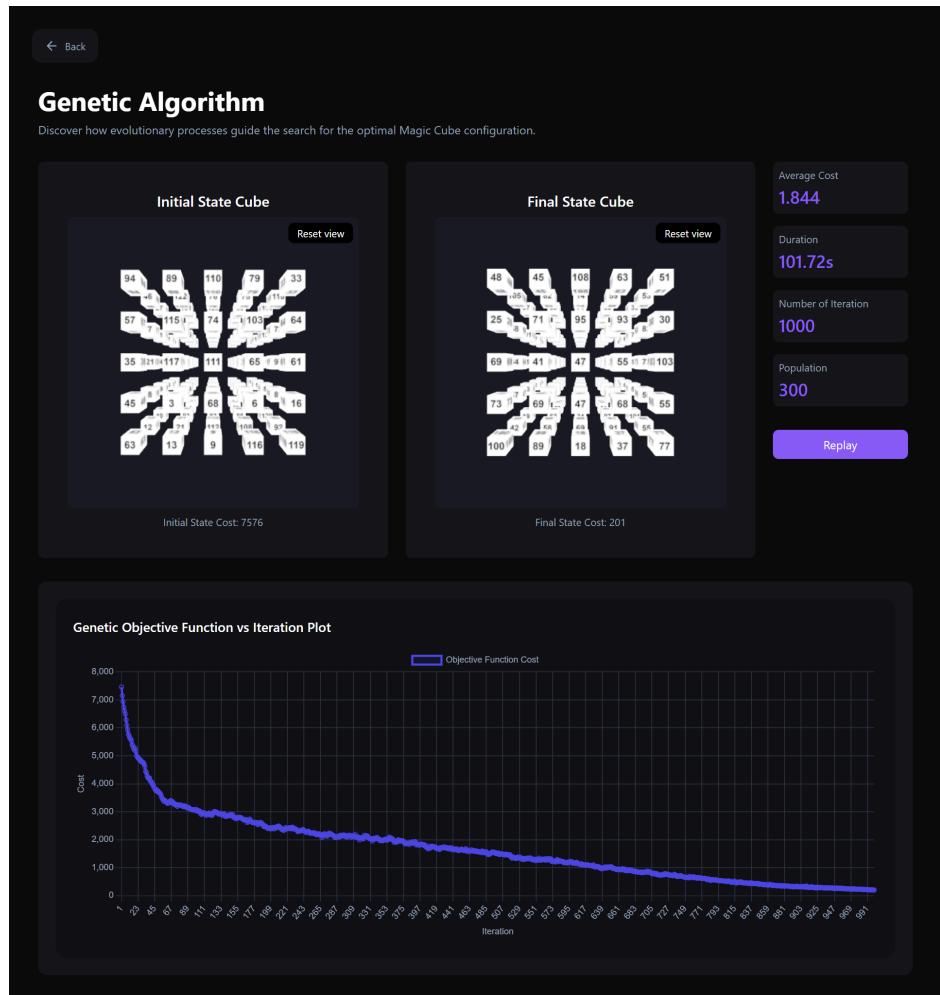
Gambar 2.59 *Genetic Algorithm* ii (1) Eksperimen III

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 376 dengan *average cost* sebesar 3.4495. Seluruh proses ini membutuhkan waktu selama 96.96 detik dan dengan total iterasi yaitu 1000 dan jumlah populasi yaitu 300.



Gambar 2.60 *Genetic Algorithm* ii (2) Eksperimen III

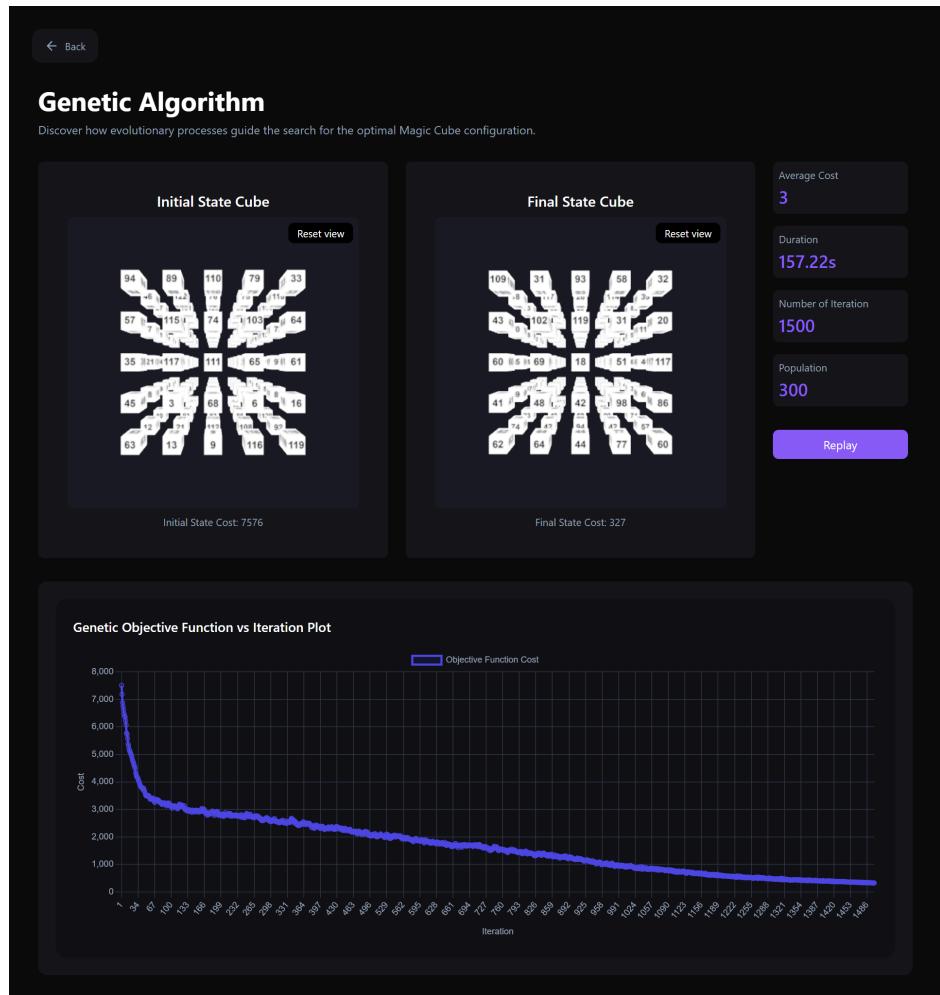
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 201 dengan *average cost* sebesar 1.844. Seluruh proses ini membutuhkan waktu selama 101.72 detik dan dengan total iterasi yaitu 1000 dan jumlah populasi yaitu 300.



Gambar 2.61 *Genetic Algorithm ii (3)* Eksperimen III

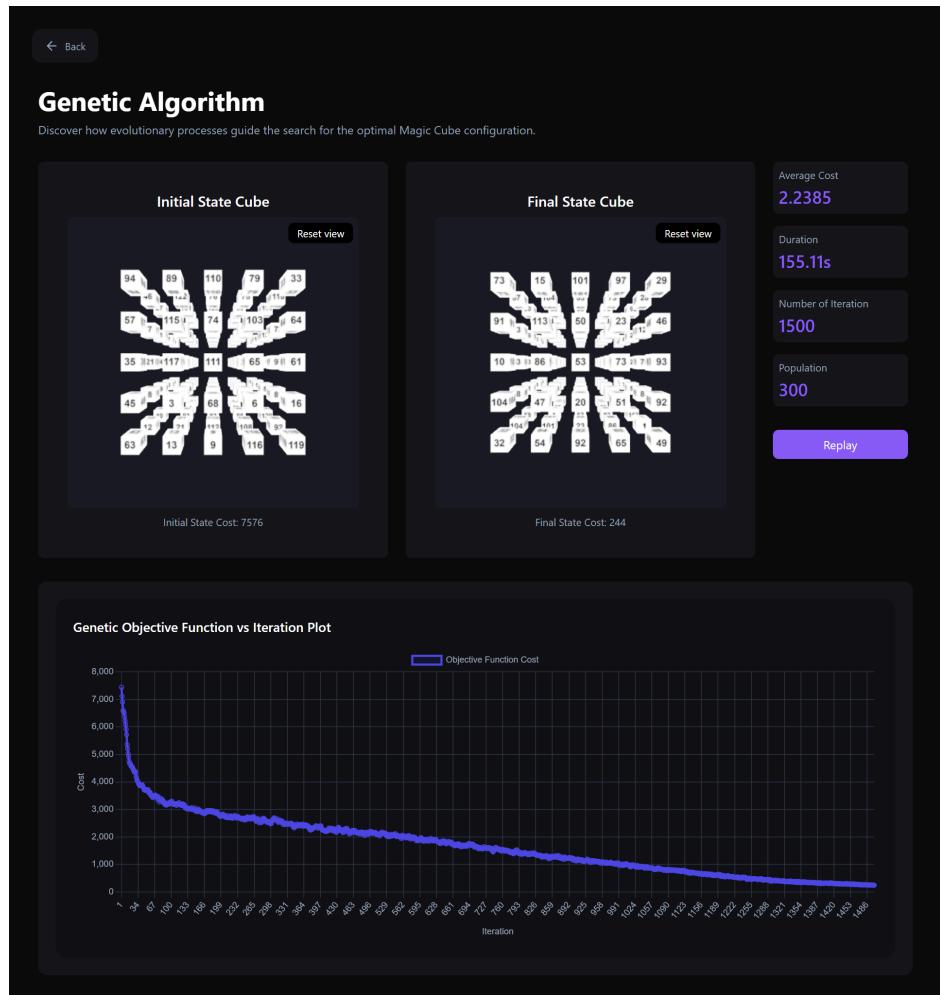
iii. Population = 300 Iteration = 1500

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama yaitu 327 dengan *average cost* sebesar 3. Seluruh proses ini membutuhkan waktu selama 157.22 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 300.



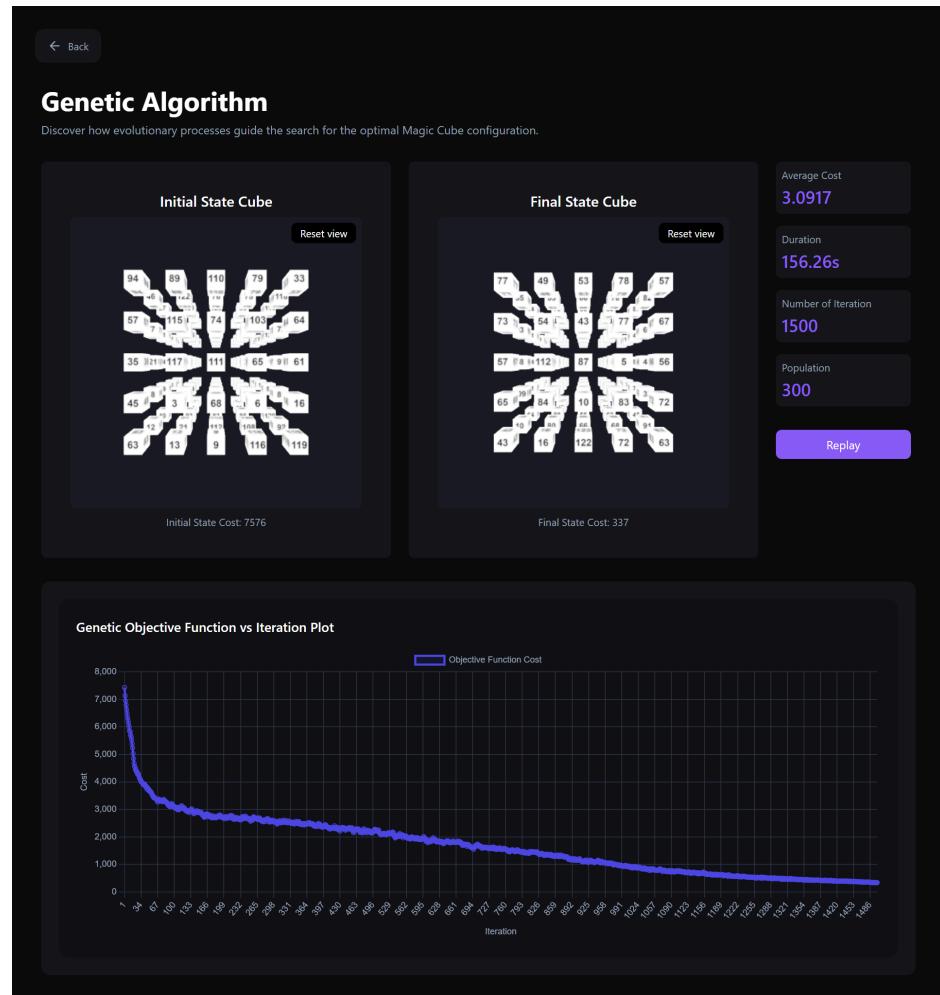
Gambar 2.62 *Genetic Algorithm* iii (1) Eksperimen III

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 244 dengan *average cost* sebesar 2.2385. Seluruh proses ini membutuhkan waktu selama 155.1 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 300.



Gambar 2.63 *Genetic Algorithm* iii (2) Eksperimen III

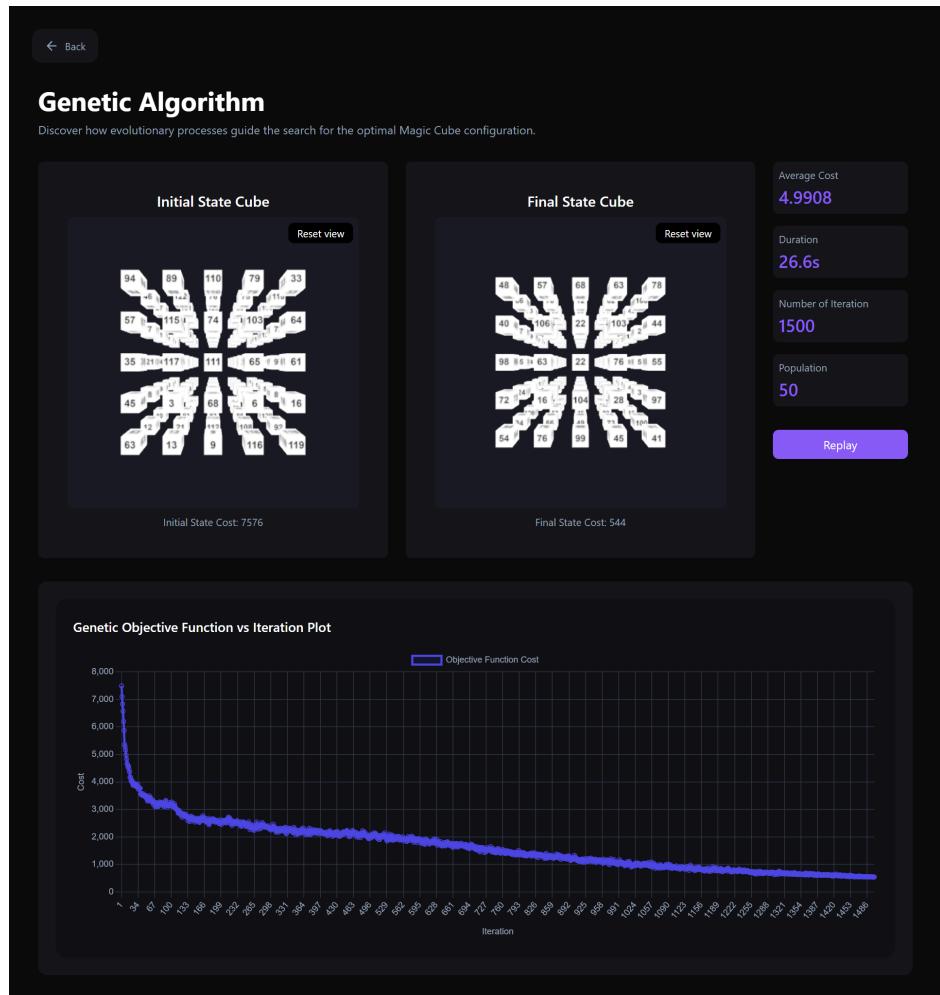
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 337 dengan *average cost* sebesar 3.0917. Seluruh proses ini membutuhkan waktu selama 156.26 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 300.



Gambar 2.64 Genetic Algorithm iii (3) Eksperimen III

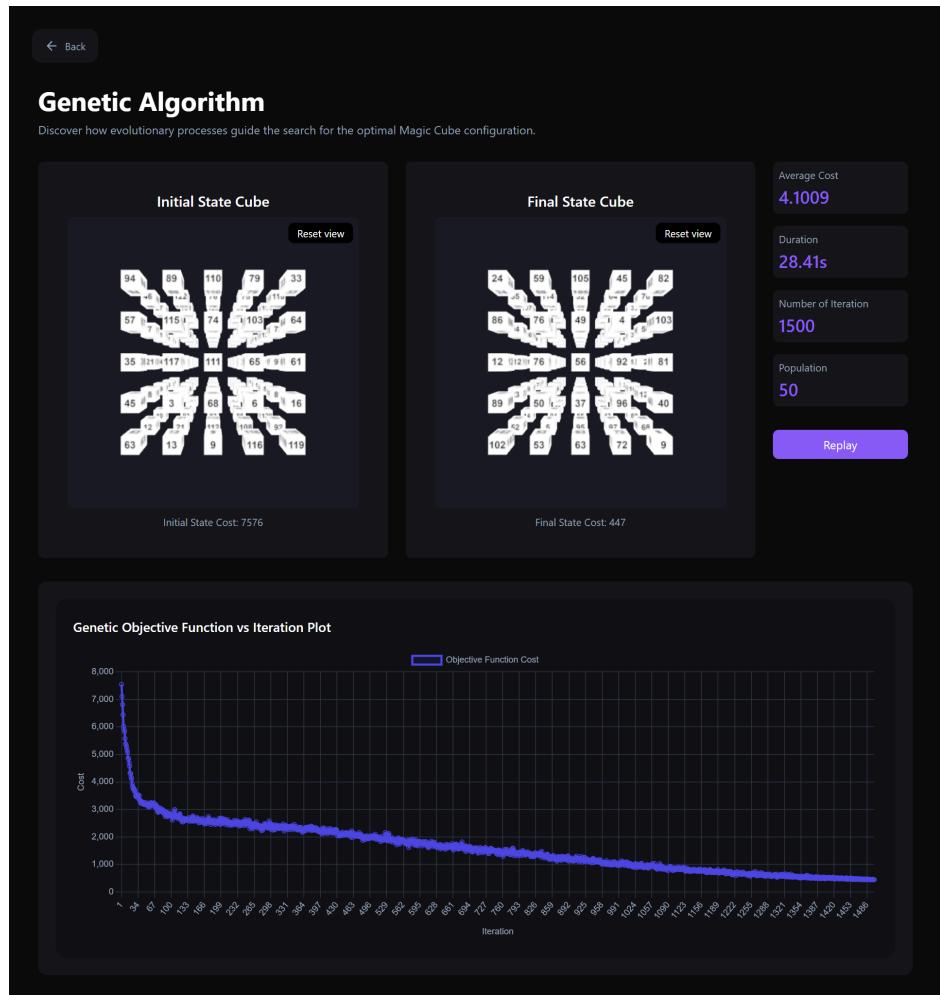
iv. Iteration = 1500 Population = 50

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama yaitu 544 dengan *average cost* sebesar 4.9908. Seluruh proses ini membutuhkan waktu selama 26.6 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 50.



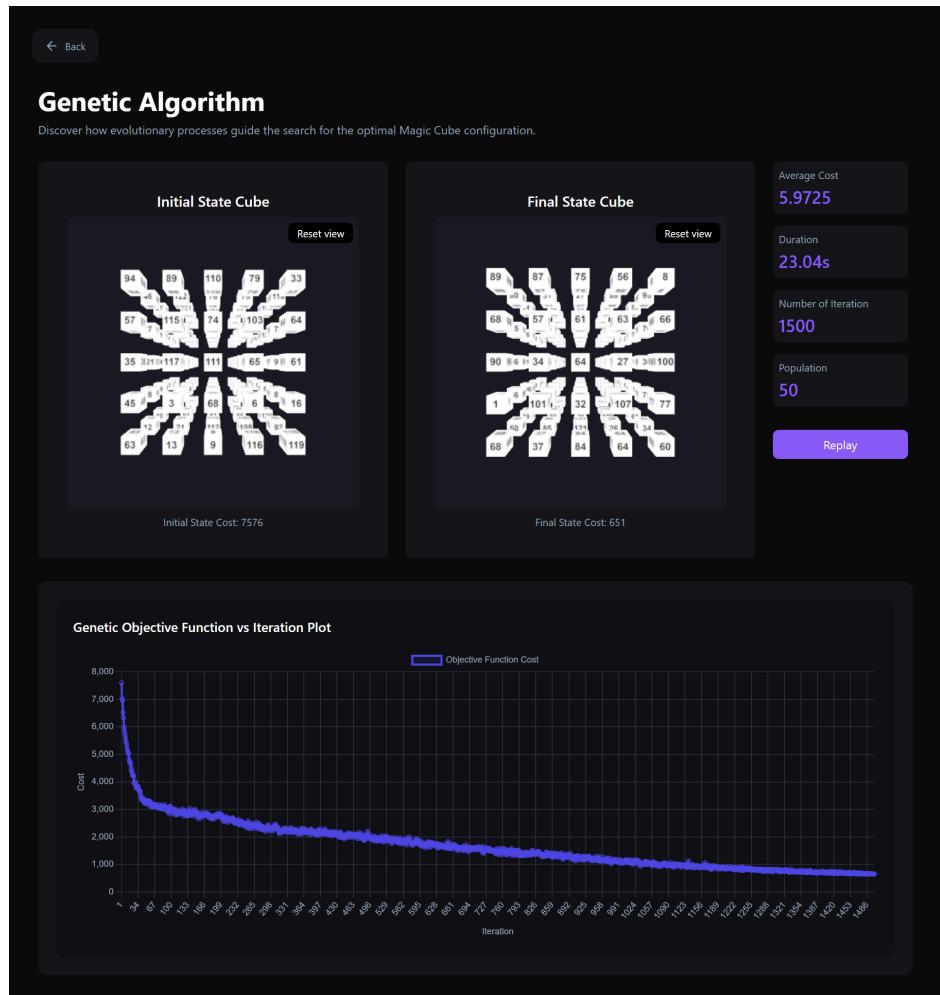
Gambar 2.65 *Genetic Algorithm* iv (1) Eksperimen III

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 447 dengan *average cost* sebesar 4.1109. Seluruh proses ini membutuhkan waktu selama 28.41 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 50.



Gambar 2.66 Genetic Algorithm iv (2) Eksperimen III

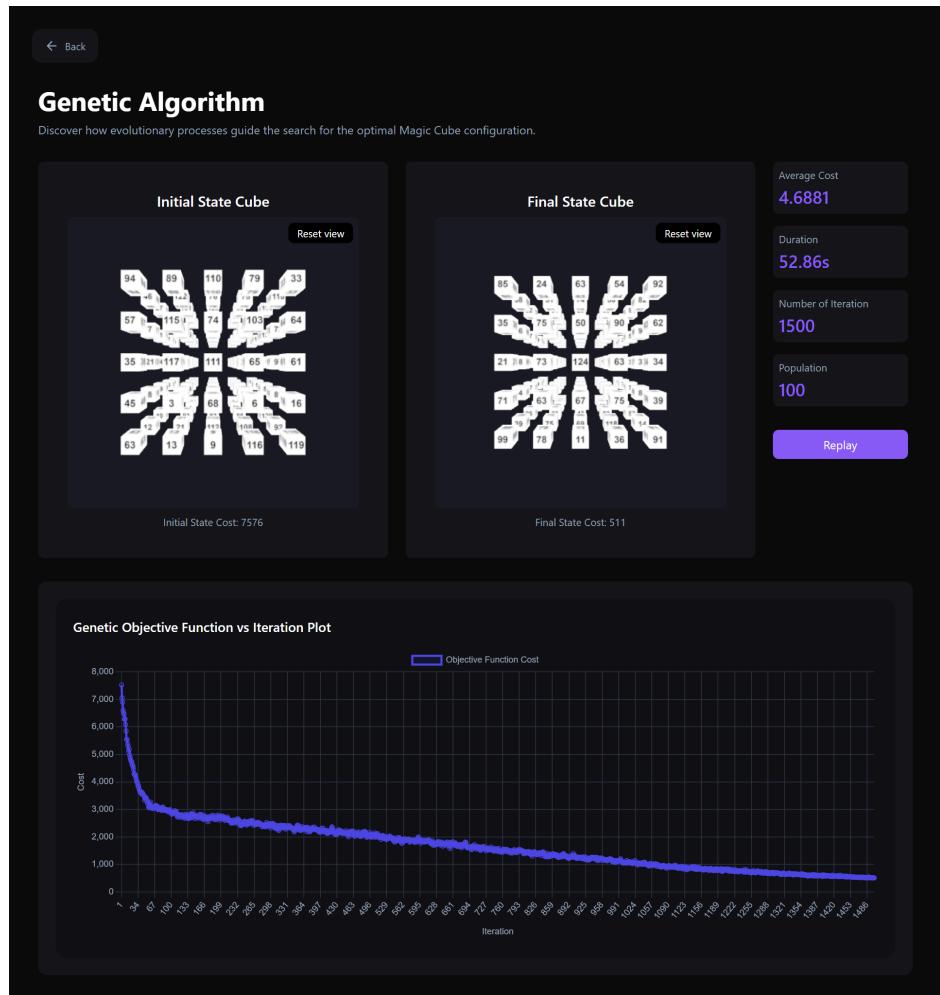
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 651 dengan *average cost* sebesar 5.9725. Seluruh proses ini membutuhkan waktu selama 23.04 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 50.



Gambar 2.67 *Genetic Algorithm* iv (3) Eksperimen III

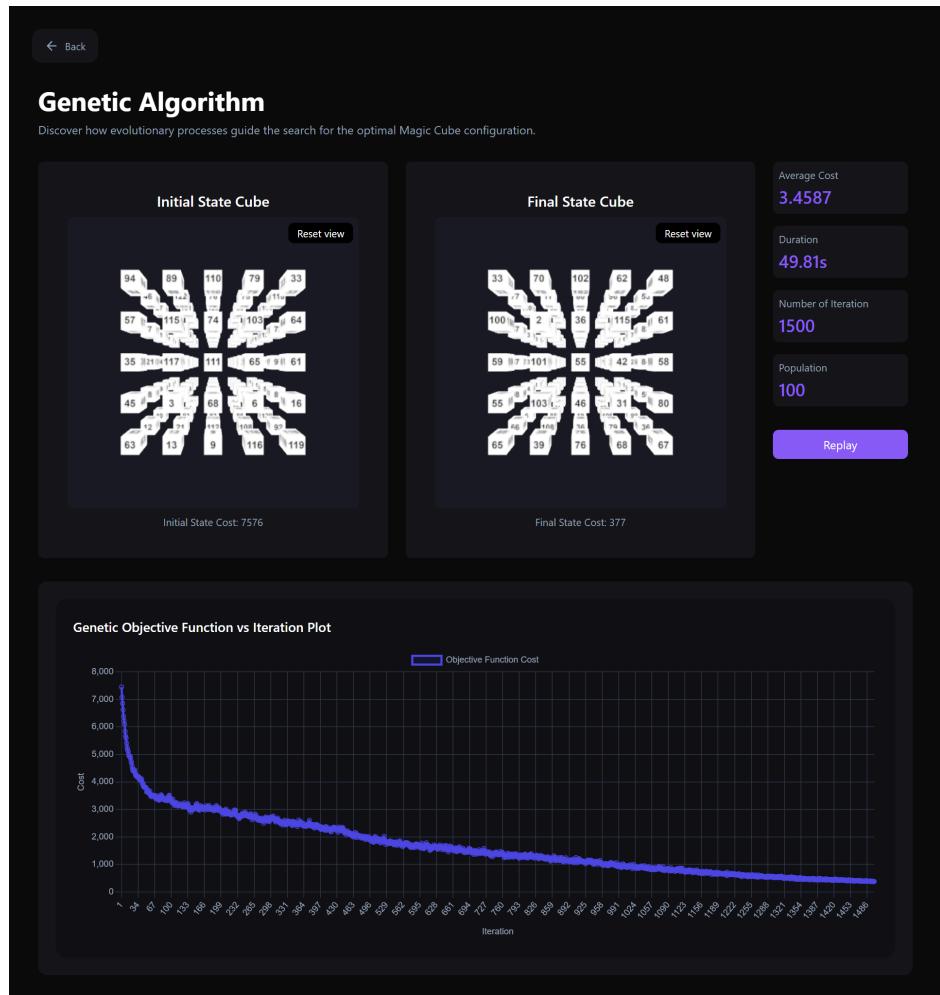
v. Iteration = 1500 Population = 100

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama yaitu 511 dengan *average cost* sebesar 4.6881. Seluruh proses ini membutuhkan waktu selama 52.86 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 100.



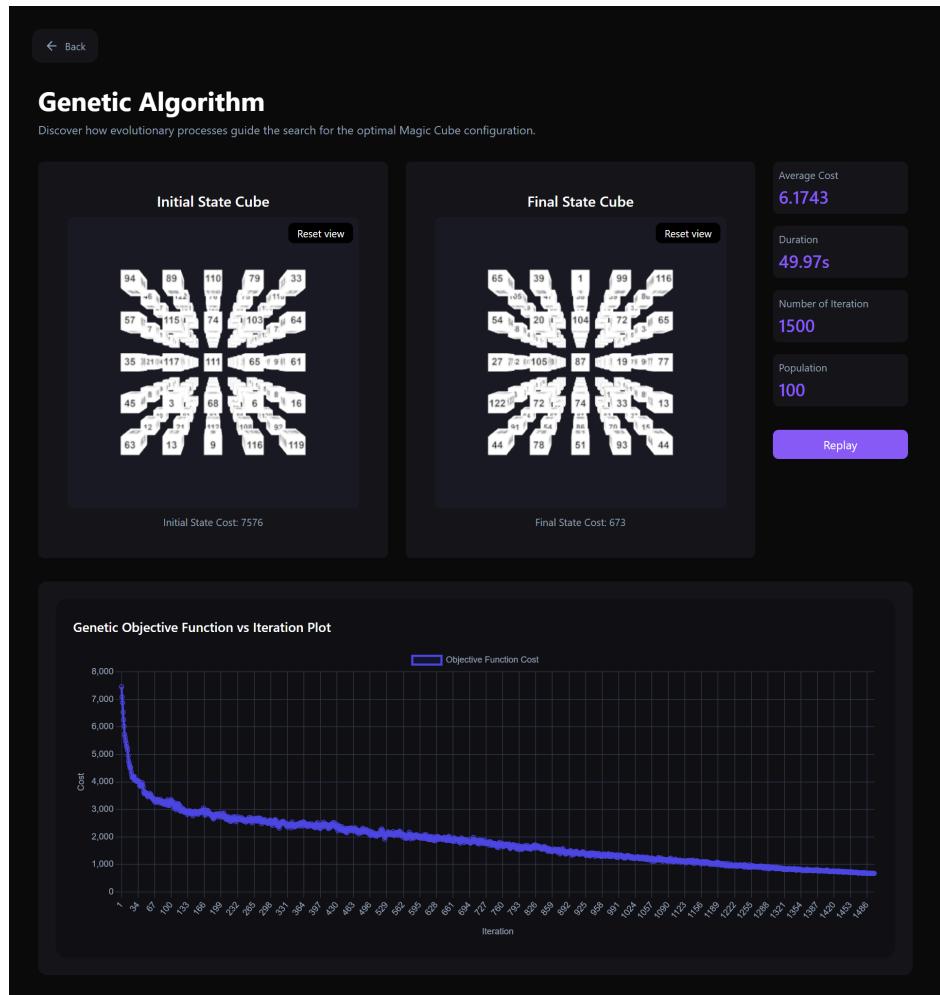
Gambar 2.68 *Genetic Algorithm v (1)* Eksperimen III

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 377 dengan *average cost* sebesar 3.4587. Seluruh proses ini membutuhkan waktu selama 49.81 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 100.



Gambar 2.69 *Genetic Algorithm v (2) Eksperimen III*

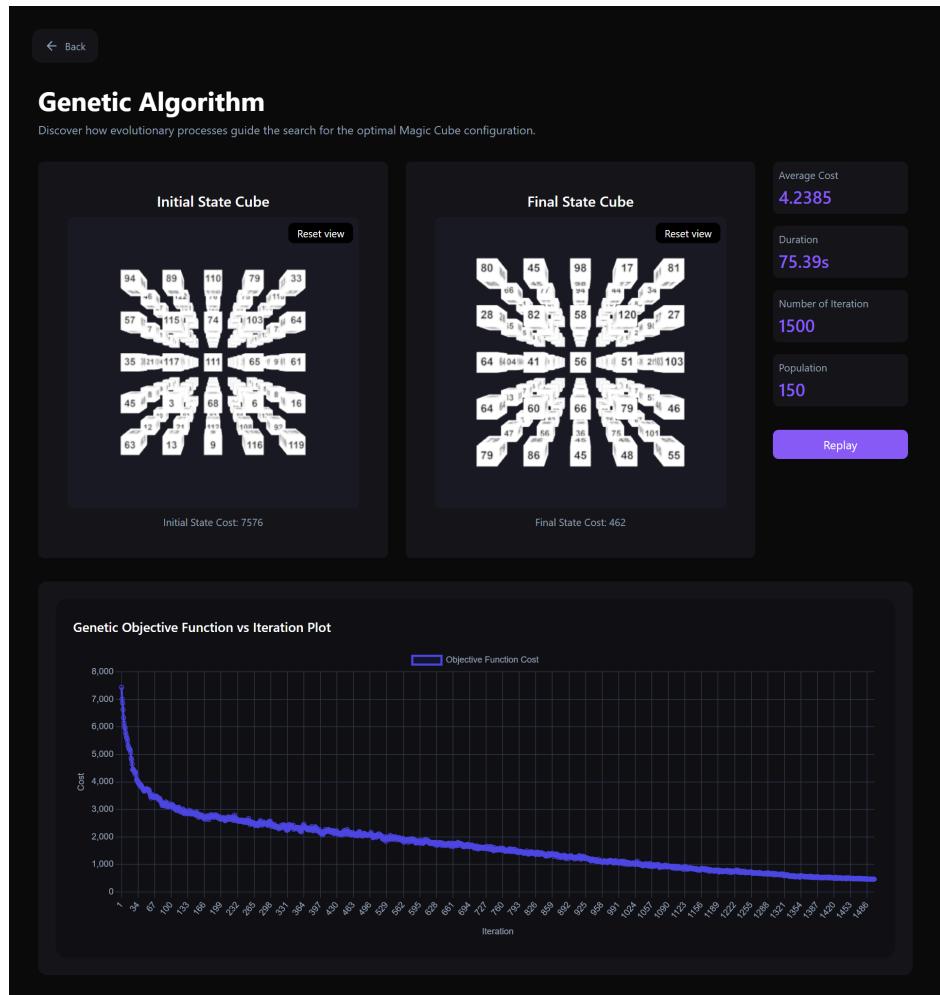
Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 673 dengan *average cost* sebesar 6.1743. Seluruh proses ini membutuhkan waktu selama 49.97 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 100.



Gambar 2.70 *Genetic Algorithm v (3)* Eksperimen III

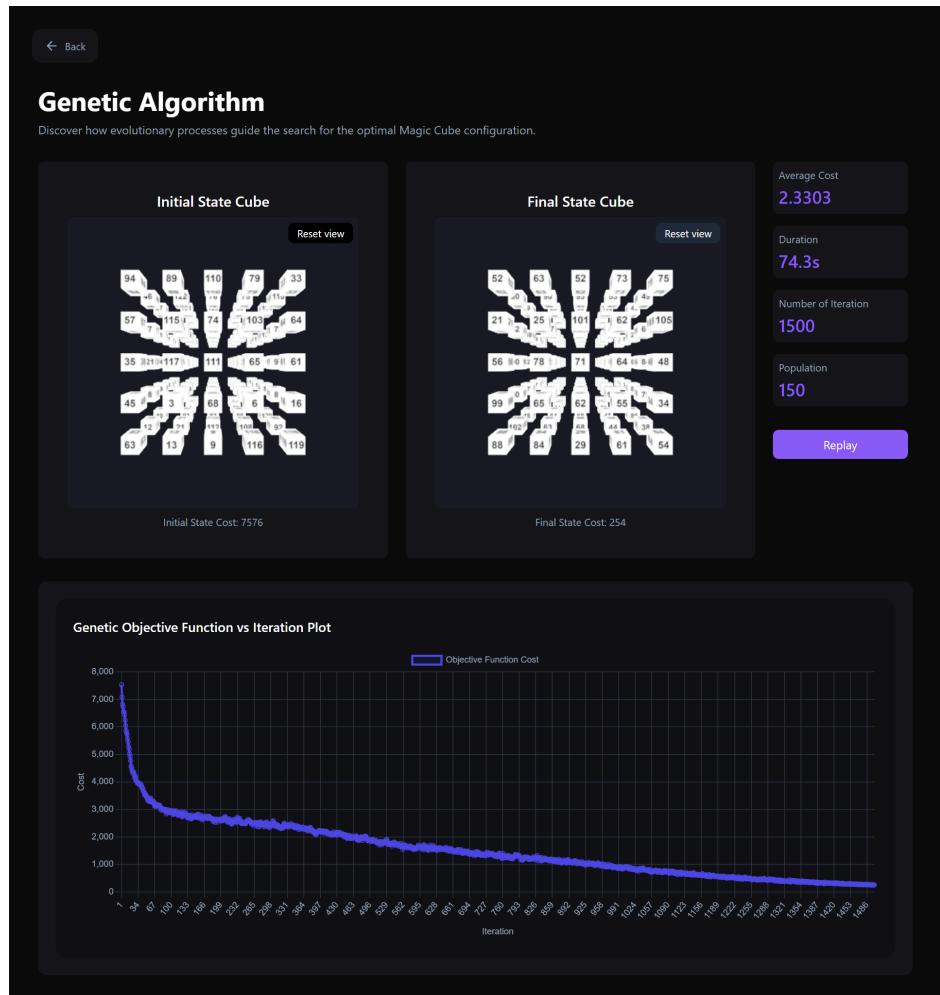
vi. Iteration = 1500 Population = 150

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan pertama yaitu 462 dengan *average cost* sebesar 4.2385. Seluruh proses ini membutuhkan waktu selama 75.39 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 150.



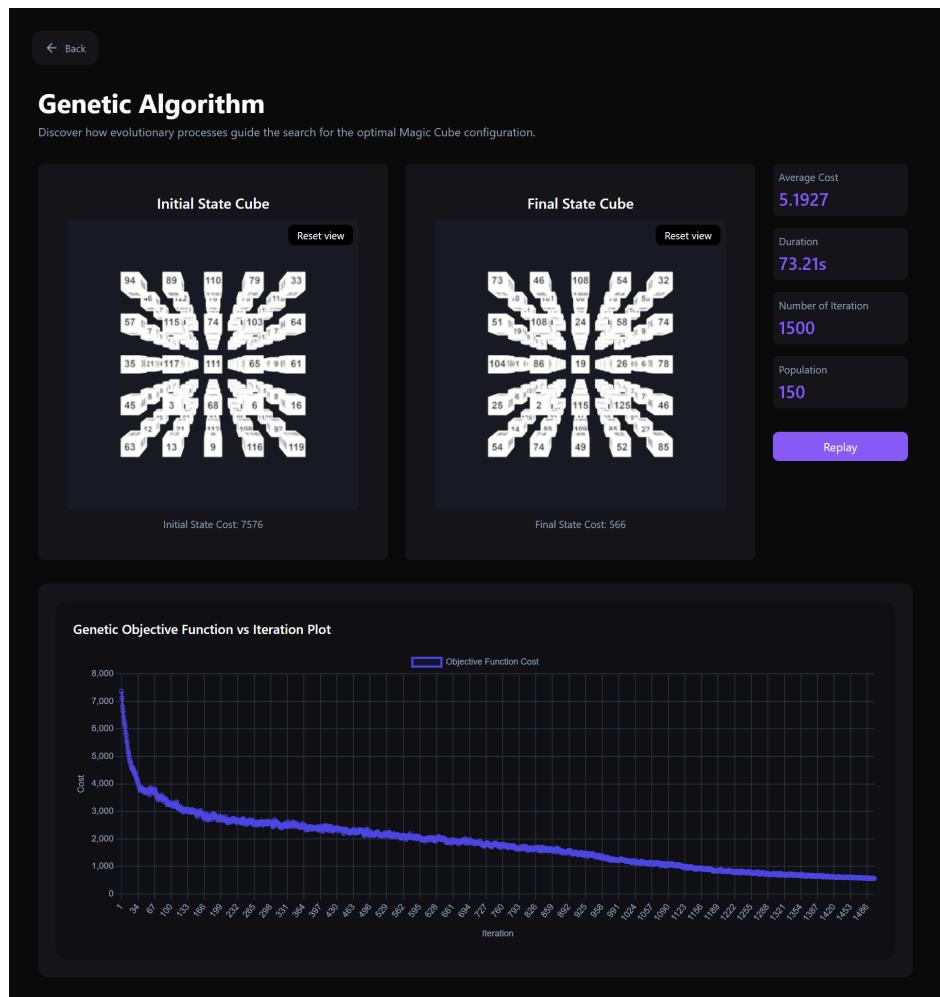
Gambar 2.71 *Genetic Algorithm* vi (1) Eksperimen III

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan kedua yaitu 254 dengan *average cost* sebesar 2.3303. Seluruh proses ini membutuhkan waktu selama 74.3 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 150.



Gambar 2.72 *Genetic Algorithm* vi (2) Eksperimen III

Pada algoritma *Genetic Algorithm*, didapatkan *final cost* pada percobaan ketiga yaitu 566 dengan *average cost* sebesar 5.1927. Seluruh proses ini membutuhkan waktu selama 73.21 detik dan dengan total iterasi yaitu 1500 dan jumlah populasi yaitu 150.



Gambar 2.73 Genetic Algorithm vi (3) Eksperimen III

Dari hasil yang diperoleh setelah melakukan pengujian pada tiap algoritma dengan tiga konfigurasi kubus, dilakukan perbandingan dengan menggunakan *best cost*, *duration*, serta *iteration* yang didapatkan.

Pada kubus pertama dengan *initial cost* sebesar 7856, didapatkan nilai *best cost*, *duration*, serta *iteration* tiap algoritma sebagai berikut.

Tabel 2.1 Perbandingan Algoritma Eksperimen I

Kriteria	Algoritma					
	Steepest	Sideways	Stochastic	Random	Simulated	Genetic
Best Cost	732	565	913	6246	689	222
Iteration	344	199	10000	21	9865	1000

Kriteria	Algoritma					
	Steepest	Sideways	Stochastic	Random	Simulated	Genetic
Duration	30.22	808.11	317	0.04	8.46	153.31

Pada Tabel 2.1, algoritma *Genetic* menunjukkan performa yang terbaik dalam hal kualitas solusi dengan menghasilkan *best cost* yang paling rendah sebesar 222. Hal ini menunjukkan bahwa algoritma ini berhasil mendekati solusi optimal lebih baik daripada algoritma lain. Sementara itu, algoritma *Random Restart Hill-climbing* memiliki keunggulan dalam kriteria *iteration* dengan nilai sebesar 21 dan *duration* tercepat yaitu 0.04 detik, tetapi *best cost* yang dihasilkan bernilai paling besar. Dari eksperimen pertama, didapatkan bahwa *Genetic* merupakan algoritma yang memberikan *best cost* terbaik.

Pada kubus kedua dengan *initial cost* sebesar 6796, didapatkan nilai *best cost*, *duration*, serta *iteration* tiap algoritma sebagai berikut.

Tabel 2.2 Perbandingan Algoritma Eksperimen II

Kriteria	Algoritma					
	Steepest	Sideways	Stochastic	Random	Simulated	Genetic
Best Cost	357	429	1060	6348	704	259
Iteration	382	156	10000	26	9865	1500
Duration	38.45	688.23	1364	0.04	8.06	60.29

Pada Tabel 2.2, algoritma *Genetic* juga menunjukkan performa yang terbaik dalam hal kualitas solusi dengan menghasilkan *best cost* yang paling rendah sebesar 259. Hal ini menunjukkan bahwa algoritma ini berhasil mendekati solusi optimal lebih baik daripada algoritma lain. Sementara itu, algoritma *Random Restart Hill-climbing* memiliki keunggulan kembali dalam kriteria *iteration* dengan nilai sebesar 26 dan *duration* tercepat yaitu 0.04 detik, tetapi *best cost* yang dihasilkan bernilai paling besar. Algoritma *Hill-climbing with Sideways Move* memiliki nilai *best cost* terbaik kedua, tetapi nilai *duration* yang dihasilkan merupakan yang terbesar. Dari eksperimen kedua, didapatkan bahwa *Genetic* merupakan algoritma yang memberikan *best cost* terbaik dengan durasi pencarian yang masih jauh lebih kecil dibandingkan *Hill-climbing with Sideways Move*.

Pada kubus ketiga dengan *initial cost* sebesar 7576, didapatkan nilai *best cost*, *duration*, serta *iteration* tiap algoritma sebagai berikut.

Tabel 2.3 Perbandingan Algoritma Eksperimen III

Kriteria	Algoritma					
	Steepest	Sideways	Stochastic	Random	Simulated	Genetic
Best Cost	872	519	891	6497	698	201
Iteration	396	275	10000	17	9865	1000
Duration	46.8	1117.9	14.22	0.04	8.03	101.72

Pada Tabel 2.3, algoritma *Genetic* juga menunjukkan performa yang terbaik dalam hal kualitas solusi dengan menghasilkan *best cost* yang paling rendah sebesar 201. Hal ini menunjukkan bahwa algoritma ini berhasil mendekati solusi optimal lebih baik daripada algoritma lain. Sementara itu, algoritma *Random Restart Hill-climbing* memiliki keunggulan kembali dalam kriteria *iteration* dengan nilai sebesar 17 dan *duration* tercepat yaitu 0.04 detik, tetapi *best cost* yang dihasilkan bernilai paling besar. Algoritma *Hill-climbing with Sideways Move* memiliki nilai *best cost* terbaik kedua, tetapi nilai *duration* yang dihasilkan merupakan yang terbesar. Dari eksperimen ketiga, didapatkan bahwa *Genetic* merupakan algoritma yang memberikan *best cost* terbaik dengan durasi pencarian yang masih jauh lebih kecil dibandingkan *Hill-climbing with Sideways Move*.

Berdasarkan hasil yang telah ditampilkan pada tabel, setiap algoritma menunjukkan tingkat kedekatan yang berbeda terhadap *global optima* yang dapat dilihat dari nilai *best cost* yang dihasilkan. Secara konsisten, algoritma **Genetic** mencapai nilai *best cost* terendah dibandingkan dengan algoritma lain. Hal ini disebabkan oleh kemampuan algoritma *Genetic* untuk mengeksplorasi ruang solusi yang lebih luas melalui mekanisme *crossover* dan *mutasi*, sehingga lebih efektif dalam menghindari jebakan di local optima dan mencari solusi yang lebih baik. Sebaliknya, algoritma Random memberikan hasil yang jauh dari optimal, karena sifatnya yang bergantung pada pencarian acak.

Algoritma *Genetic* mampu menghasilkan nilai *best cost* yang lebih baik di setiap eksperimen dan lebih unggul dalam mengatasi masalah *local optima*. Sementara itu, algoritma *Steepest Ascent Hill-climbing* dan *Hill-climbing with Sideways Move*, meskipun dapat mendekati solusi yang baik, masih berada di belakang *Genetic* dalam kinerjanya. Algoritma *Simulated Annealing* juga menunjukkan hasil yang cukup baik, tetapi masih lebih rendah dari *Genetic*, meskipun mekanisme *annealing schedule* memungkinkan pencarian yang lebih eksplorasi. Di sisi lain, algoritma *Stochastic* dan *Random Restart* menghasilkan nilai *best cost* yang jauh lebih tinggi, menunjukkan ketidakmampuan mereka dalam mencapai *global optima* secara konsisten.

Pada hal kecepatan atau durasi yang dibutuhkan, algoritma *Random Restart* memiliki waktu eksekusi tercepat di setiap eksperimen karena tidak menggunakan evaluasi *fitness* yang kompleks atau mekanisme eksplorasi yang canggih. Sementara itu, algoritma *Genetic*

membutuhkan waktu yang jauh lebih lama dibandingkan *Random Restart*, namun tetap lebih unggul dalam kualitas solusi yang dihasilkan. Algoritma *Hill-climbing with Sideways Move* dan *Stochastic* memakan waktu jauh lebih lama karena jumlah iterasi yang tinggi, sedangkan *Steepest Ascent Hill-climbing* berada di tengah-tengah dalam hal durasi. *Simulated Annealing* memiliki waktu eksekusi yang relatif singkat, tetapi lebih lama dari *Random Restart* karena membutuhkan evaluasi berulang untuk setiap perubahan suhu.

Dari segi konsistensi, algoritma *Genetic* menunjukkan hasil yang stabil di seluruh eksperimen, selalu mencapai nilai *best cost* yang rendah. Algoritma *Steepest Ascent Hill-climbing* dan *Hill-climbing with Sideways Move* juga menunjukkan hasil yang cukup konsisten, tetapi nilainya bergantung pada titik awal yang dipilih. Sementara itu, algoritma *Random Restart* dan *Stochastic* menunjukkan ketidakstabilan dalam hasil, dengan *best cost* yang sangat bervariasi yang mengindikasikan kurangnya kontrol dalam eksplorasi ruang solusi.

Pengaruh jumlah iterasi dan ukuran populasi pada algoritma *Genetic* cukup signifikan terhadap hasil akhir. Jumlah iterasi yang lebih banyak memungkinkan algoritma untuk mengeksplorasi lebih banyak kemungkinan solusi yang dapat membantu mendekati *global optima*, meskipun dengan biaya waktu yang lebih tinggi. Ukuran populasi yang besar memberikan variasi yang lebih kaya dalam proses seleksi yang meningkatkan peluang untuk menghasilkan individu-individu baru yang mendekati solusi optimal. Namun, peningkatan jumlah iterasi dan populasi secara berlebihan dapat menyebabkan waktu eksekusi yang jauh lebih lama tanpa perbaikan signifikan pada *best cost*, sehingga diperlukan keseimbangan dalam menetapkan parameter ini untuk mendapatkan hasil terbaik.

F. Bonus

1. Seluruh Algoritma

2. Video Player

i. Source Code

`Player.jsx` adalah komponen React yang berfungsi sebagai video player untuk memvisualisasikan perubahan state atau konfigurasi pada magic cube. Fitur-fitur utama yang ada dalam komponen ini adalah kontrol pemutaran, pengaturan kecepatan pemutaran, dan penampilan jumlah iterasi serta *restart* jika diperlukan.

Berikut rincian fitur yang digunakan dalam kode ini:

1. Inisialisasi State dan Ref:

- `isPlaying`, `playbackSpeed`, dan `elapsedTime` mengatur status pemutaran video, kecepatan playback, dan waktu yang sudah dilalui.
- `requestRef` digunakan untuk menangani `requestAnimationFrame` guna animasi pemutaran yang halus.

- `restartCount` dan `iterationRestart` adalah mekanisme untuk menghitung jumlah `restart` berdasarkan iterasi tertentu yang disimpan dalam array `iterationRestart`.

2. Logika Kontrol Pemutaran

- **Play/Pause:** `handlePlayPause` mengubah status `isPlaying` antara play dan pause.
- **Progress Bar:** Pengguna bisa menggeser *progress bar* untuk mengatur posisi *state* yang sedang ditampilkan pada magic cube. Fungsi `handleProgressChange` mengubah `elapsedTime` berdasarkan nilai yang dipilih.

3. Menghitung State saat ini:

- `currentCubeStateIndex` menghitung *state* magic cube yang ditampilkan berdasarkan `elapsedTime` dan `playbackSpeed`.
- `currentCubeState` memanggil data *state* yang sesuai dari `states` untuk ditampilkan di layar.

4. Animasi dengan `requestAnimationFrame`

- `animateProgress` adalah fungsi yang memperbarui `elapsedTime` secara berkala ketika video sedang dimainkan. Ini memastikan animasi berjalan dengan kecepatan yang tepat.

5. Render Komponen Cube:

- `MemoizedCube` adalah versi memoized dari komponen `Cube`, yang memastikan hanya berubah ketika *props* berubah, sehingga mengurangi rendering berulang.

6. Tampilan dan Kontrol:

- Komponen ini menggunakan beberapa tombol untuk mengontrol kecepatan playback (`playbackSpeeds`) dan juga menunjukkan informasi iterasi dan `restart`.

```
import React, { useState, useRef, useEffect, memo } from
"react";
import Cube from "./Cube";
import { X, Play, Pause } from "lucide-react";

const MemoizedCube = memo(Cube);

export default function Player({
  onClose,
  states,
  playbackSpeeds = [0.5, 1, 2, 4],
```

```

iteration,
restart = null,
population = null,
iterationRestart = [], // Array containing iteration
thresholds for each restart
)) {
  const [isPlaying, setIsPlaying] = useState(false);
  const [playbackSpeed, setPlaybackSpeed] = useState(1);
  const [elapsedTime, setElapsedTime] = useState(0); // Track
time in seconds
  const requestRef = useRef(null);
  const [restartCount, setRestartCount] = useState(0); // Initialize restart count

  const maxIterations = Math.min(states.length, 10000);
  const totalDuration = (maxIterations / 10000) * 1000;
  const stateDuration = 1000 / 10000;

  // Calculate the current state index based on elapsedTime
and playbackSpeed
  const currentCubeStateIndex = Math.min(
    Math.floor(elapsedTime / (stateDuration /
playbackSpeed)),
    maxIterations - 1
  );

  const currentCubeState = states[currentCubeStateIndex];

  useEffect(() => {
    // Check if currentCubeStateIndex has crossed the next
threshold in iterationRestart
    if (iterationRestart.length > 0 && currentCubeStateIndex
>= iterationRestart[0]) {
      setRestartCount((prev) => prev + 1);
      iterationRestart.shift(); // Remove the first element
as it's reached
    }
  }, [currentCubeStateIndex, iterationRestart]);

```

```

const handlePlayPause = () => {
  setIsPlaying((prev) => !prev);
};

const handleProgressChange = (e) => {
  const newElapsed = parseFloat(e.target.value) * totalDuration / 100;
  setElapsedTime(newElapsed);
};

const handleSpeedChange = (speed) => {
  setPlaybackSpeed(speed);
};

const animateProgress = () => {
  setElapsedTime((prev) => {
    const newElapsed = prev + playbackSpeed * 0.1;
    if (newElapsed >= totalDuration) {
      setIsPlaying(false);
      return totalDuration;
    }
    return newElapsed;
  });
  requestRef.current =
  requestAnimationFrame(animateProgress);
};

useEffect(() => {
  if (isPlaying) {
    requestRef.current =
    requestAnimationFrame(animateProgress);
  } else {
    cancelAnimationFrame(requestRef.current);
  }
  return () => cancelAnimationFrame(requestRef.current);
}, [isPlaying, playbackSpeed]);

return (
<div className="fixed inset-0 bg-[rgba(10,10,10,0.8)]"

```

```

flex justify-center items-center z-50">
  <div className="relative bg-[#16181d] p-6 rounded-lg
w-[85%] max-w-[1000px] flex flex-col items-center">
    <button
      onClick={onClose}
      className="absolute top-0 right-0 mt-2 mr-2 z-50
text-white hover:text-gray-400 transition"
    >
      <x className="w-6 h-6" />
    </button>

    <div className="relative w-full h-[400px]
bg-[#1a1d24] rounded-lg flex justify-center items-center
overflow-hidden mb-4">
      <MemoizedCube
        key={`player-cube-${currentCubeStateIndex}`}
        magic_cube={currentCubeState} />
    </div>

    <input
      type="range"
      min="0"
      max="100"
      value={(elapsedTime / totalDuration) * 100}
      onChange={handleProgressChange}
      className="w-full mb-2"
    />

    <div className="flex items-center justify-center
gap-4 mt-4">
      <button
        onClick={handlePlayPause}
        className="p-3 bg-[#8b5cf6] rounded-full
text-white hover:bg-[#7a4bd1] transition"
      >
        {isPlaying ? <Pause className="w-5 h-5" /> :
<Play className="w-5 h-5" />}
      </button>
    </div>
  </div>

```

```

        {playbackSpeeds.map((speed) => (
            <button
                key={speed}
                onClick={() => handleSpeedChange(speed)}
                className={`px-3 py-1 rounded-full ${(
                    playbackSpeed === speed ? "bg-[#8b5cf6]"
                    : "bg-[#1a1d24] text-[#94a3b8]"
                ) transition`}
            >
                {speed}x
            </button>
        ))}
    </div>

    <div className="absolute bottom-4 left-4 text-white text-sm">
        Iteration: {currentCubeStateIndex}
    </div>
    {population !== null && (
        <div className="absolute bottom-4 right-4 text-white text-sm">
            Population: {population}
        </div>
    )}

    {restart !== null && (
        <div className="absolute bottom-4 right-4 text-white text-sm">
            Restart: {restartCount}
        </div>
    )}
</div>
);
}

```

ii. Eksperimen

Pada eksperimen dalam gambar, **Player.jsx** digunakan untuk memutar kembali konfigurasi dari magic cube pada kecepatan yang berbeda. Pengguna

dapat mengontrol visualisasi melalui tombol *play*, *pause*, dan pengaturan kecepatan playback.

1. Tampilan Cube

Cube di tengah menunjukkan keadaan yang sesuai dengan iterasi yang diputar saat itu. Pengguna dapat melihat setiap perubahan pada posisi angka dalam magic cube sesuai dengan iterasi yang berlangsung.

2. Progress Bar

Di bagian bawah, terdapat progress bar yang menampilkan posisi iterasi saat ini. Pada gambar, iterasi berada di angka **0** menandakan awal dari pemutaran.

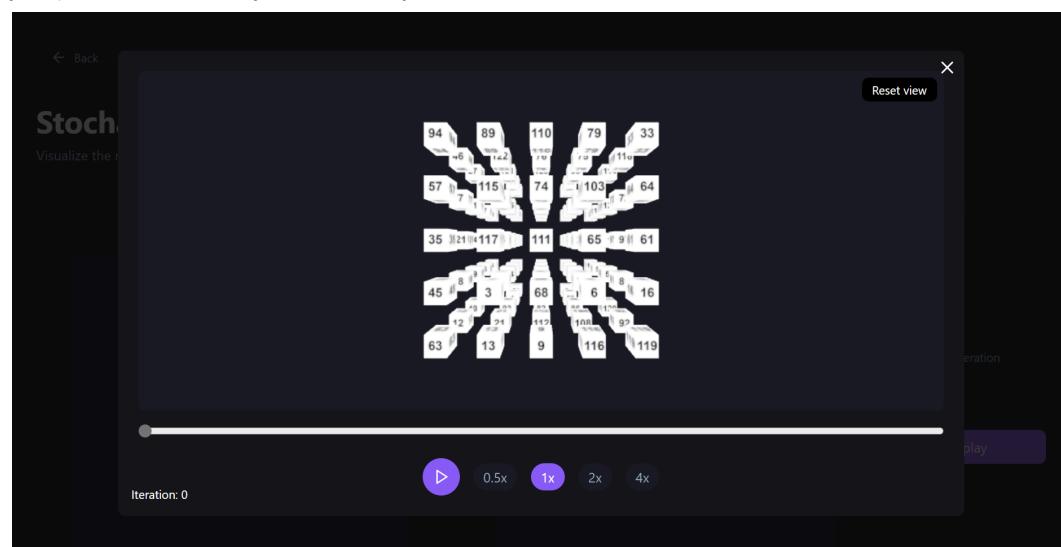
3. Pengaturan Playback Speed

Terdapat beberapa opsi kecepatan seperti 0.5x, 1x, 2x, dan 4x. Kecepatan ini memungkinkan pengguna untuk mempercepat atau memperlambat visualisasi perubahan konfigurasi magic cube.

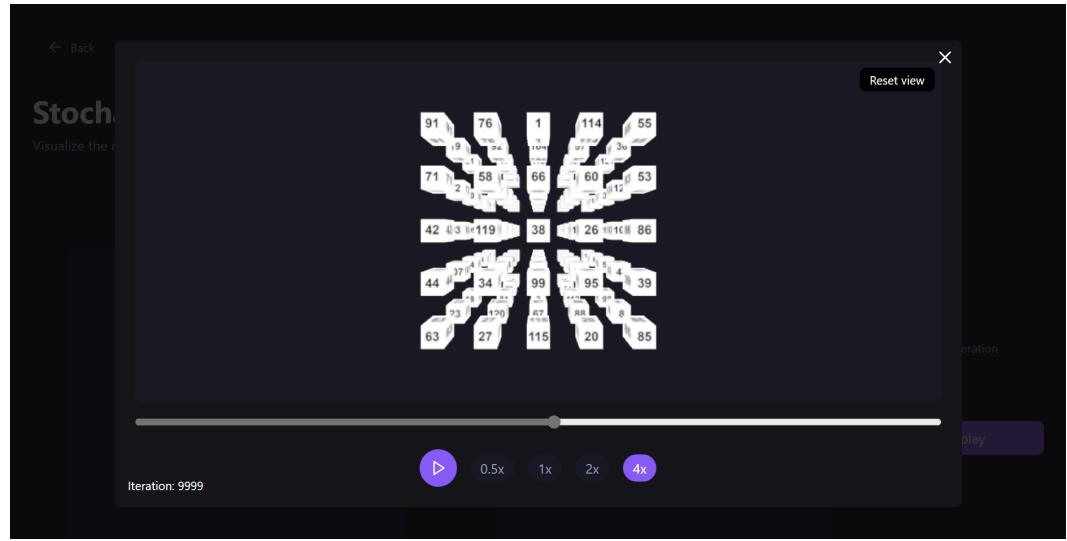
4. Informasi Iterasi

Di bagian kiri bawah dari player, pengguna dapat melihat iterasi yang sedang ditampilkan (pada gambar menunjukkan "Iteration: 0").

Eksperimen ini menunjukkan bahwa pengguna bisa menyesuaikan iterasi dan kecepatan playback dengan mudah, memfasilitasi visualisasi proses penyelesaian atau perubahan pada konfigurasi magic cube.



Gambar 2.74 Eksperimen Video Player (1)



Gambar 2.75 Eksperimen Video Player (2)

3. Save and Load

i. Source Code

Source code pada file **SaveLoad.jsx** ini adalah komponen React yang menyediakan fungsi untuk menyimpan dan memuat data konfigurasi "magic cube" ke dan dari backend. Komponen ini mengelola proses berikut:

1. Inisialisasi dan Fetch Data

Ketika komponen pertama kali dimuat, ia memanggil fungsi `fetchSavedFiles` yang melakukan permintaan `GET` ke backend untuk mendapatkan daftar file yang sudah disimpan. Data file disimpan dalam `state savedFiles` untuk ditampilkan pada panel.

2. Menyimpan Konfigurasi Cube

- Fungsi `handleSave` bertugas untuk mengirimkan data konfigurasi magic cube ke backend melalui `POST` request. Pada fungsi ini, data cube yang ingin disimpan dikirimkan dalam bentuk JSON dengan nama file yang ditentukan oleh pengguna.
- Backend akan menyimpan data ini sebagai file JSON, dan setelah tersimpan, komponen akan memperbarui daftar file yang tersimpan melalui pemanggilan ulang `fetchSavedFiles`.

3. Memuat Konfigurasi Cube

- Fungsi `handleLoad` menerima nama file dan memuat konfigurasi cube yang tersimpan dari backend. Permintaan `GET` dikirim ke backend dengan parameter nama file.
- Setelah file cube diambil dari backend, data konfigurasi ini dikirimkan kembali ke komponen utama untuk ditampilkan di aplikasi. Mau dicopas di mana?

```

import React, { useState, useEffect } from 'react';
import axios from 'axios';
import { saveAs } from 'file-saver';

const API_URL = 'http://localhost:8000';

export default function SaveLoadPanel({ cubeData, onLoadCube }) {
    const [fileName, setFileName] = useState('');
    const [savedFiles, setSavedFiles] = useState([]);
    const [isPanelVisible, setIsPanelVisible] =
        useState(false);

    // Fetch saved files from backend when component mounts
    useEffect(() => {
        fetchSavedFiles();
    }, []);

    const fetchSavedFiles = async () => {
        try {
            const response = await
                axios.get(`${API_URL}/saved_cubes`);
            setSavedFiles(response.data);
        } catch (error) {
            console.error("Error fetching saved files:", error);
        }
    };

    const handleSave = async () => {
        if (!fileName) {
            alert("Please enter a file name.");
            return;
        }
        try {
            await axios.post(` ${API_URL}/save_cube`, {
                file_name: fileName,
                cube: cubeData
            });
        }
    };
}

```

```

        setFileName('');
        fetchSavedFiles(); // Refresh the saved files list
        alert("Cube saved successfully.");
    } catch (error) {
        console.error("Error saving cube:", error);
    }
};

const handleLoad = async (file) => {
    try {
        const response = await axios.get(` ${API_URL}/load_cube/${file}`);
        onLoadCube(response.data.cube); // Send loaded data
        back to parent component
        setIsPanelVisible(false); // Close panel after loading
    } catch (error) {
        console.error("Error loading cube:", error);
    }
};

return (
    <div className="bg-[#111318] p-4 rounded-lg">
        <div className="flex justify-between items-center">
            <h3 className="text-lg font-semibold mb-2">Save />
            Load Cube</h3>
            <button
                onClick={() => setIsPanelVisible(!isPanelVisible)}
                className="bg-[#8b5cf6] text-white px-2 py-1
                rounded-lg font-medium hover:opacity-90 transition-opacity"
            >
                {isPanelVisible ? 'Close' : 'Open'}
            </button>
        </div>
        {isPanelVisible && (
            <div className="mt-4">
                {/* Save Cube Section */}
                <div className="mb-4">
                    <h4 className="font-medium">Save Cube</h4>
                    <input

```

```

        type="text"
        placeholder="Enter filename"
        value={fileName}
        onChange={(e) => setFileName(e.target.value)}
        className="w-full mt-2 p-2 rounded bg-[#1a1d24]
text-white"
      />
      <button
        onClick={handleSave}
        className="mt-2 bg-[#4ade80] text-white px-4
py-2 rounded-lg font-medium hover:opacity-90
transition-opacity"
      >
        Save Cube
      </button>
    </div>

    {/* Load Cube Section */}
    <div>
      <h4 className="font-medium">Load Cube</h4>
      <ul className="text-sm text-[#94a3b8] mt-2">
        {savedFiles.map((file, index) => (
          <li key={index} className="flex
justify-between items-center py-2">
            <span>{file}</span>
            <button
              onClick={() => handleLoad(file)}
              className="bg-[#8b5cf6] text-white px-2
py-1 rounded-lg font-medium hover:opacity-90
transition-opacity"
            >
              Load
            </button>
          </li>
        )) }
      </ul>
    </div>
  </div>
)

```

```
    </div>
)
}
```

```
@app.post("/save_cube")
async def save_cube(cube_data: CubeData):
    file_path = os.path.join(SAVE_DIR,
f"{cube_data.file_name}.json")
    with open(file_path, "w") as file:
        json.dump({"magic_cube": cube_data.cube}, file)
    return {"message": "Cube saved successfully"}

@app.post("/load_cube", response_model=CubeInitResponse)
async def load_cube(file: UploadFile = File(...)):
    try:
        content = await file.read()
        data = json.loads(content)
        magic_cube = data.get("magic_cube")
        if not magic_cube or not isinstance(magic_cube,
list):
            raise HTTPException(status_code=400,
detail="Invalid cube format")

        # Check if the loaded cube meets the magic cube
        objective
        cost = objective_function(magic_cube)
        return {
            "initial_cube": magic_cube,
            "initial_cost": cost
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Failed
to load cube: {str(e)}")
```

ii. Eksperimen

Pada eksperimen yang terlihat di gambar, pengguna melakukan beberapa proses untuk menguji fitur **Load Cube** dan **Save Cube**. Berikut langkah-langkah yang dilakukan:

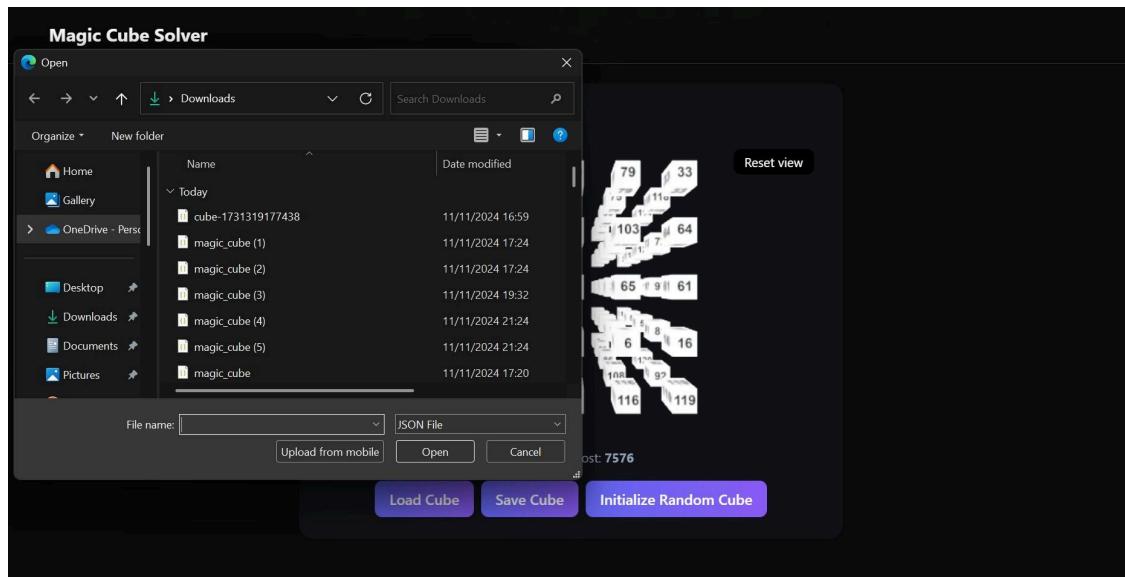
1. Menyimpan Cube

Pada gambar pertama, pengguna menyimpan beberapa konfigurasi magic cube dengan nama file yang berbeda, seperti `magic_cube`, `magic_cube (1)`, `magic_cube (2)`, dll.

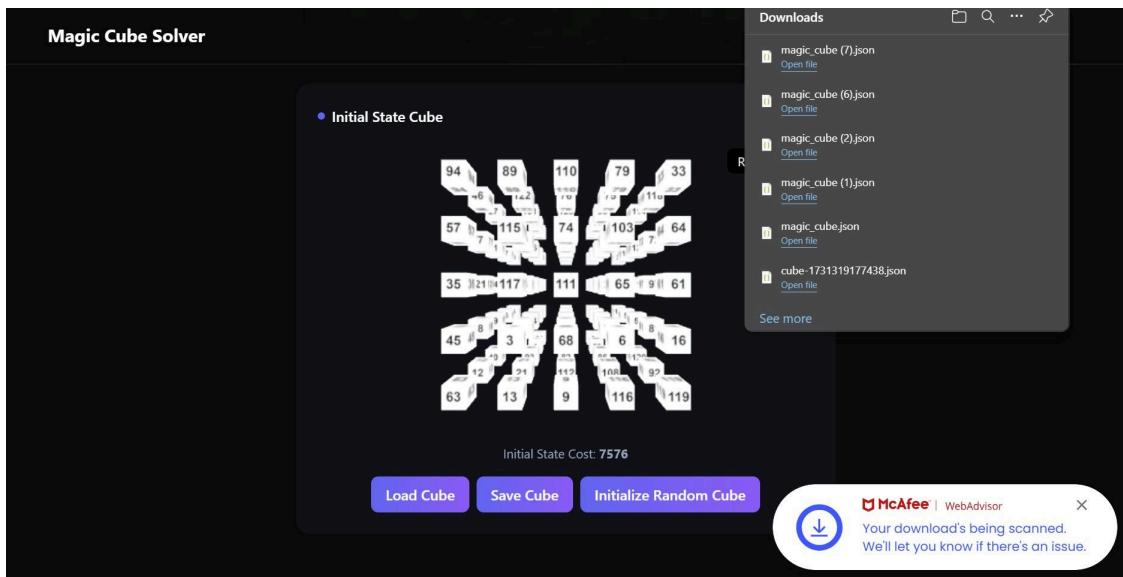
2. Memuat Cube dengan Cost 0

Pada gambar ketiga, terlihat bahwa salah satu konfigurasi cube yang dimuat memiliki `Initial State Cost` sebesar **0**. Ini menunjukkan bahwa cube tersebut adalah konfigurasi yang benar-benar memenuhi syarat *magic cube* tanpa adanya kesalahan atau deviasi dari target magic number.

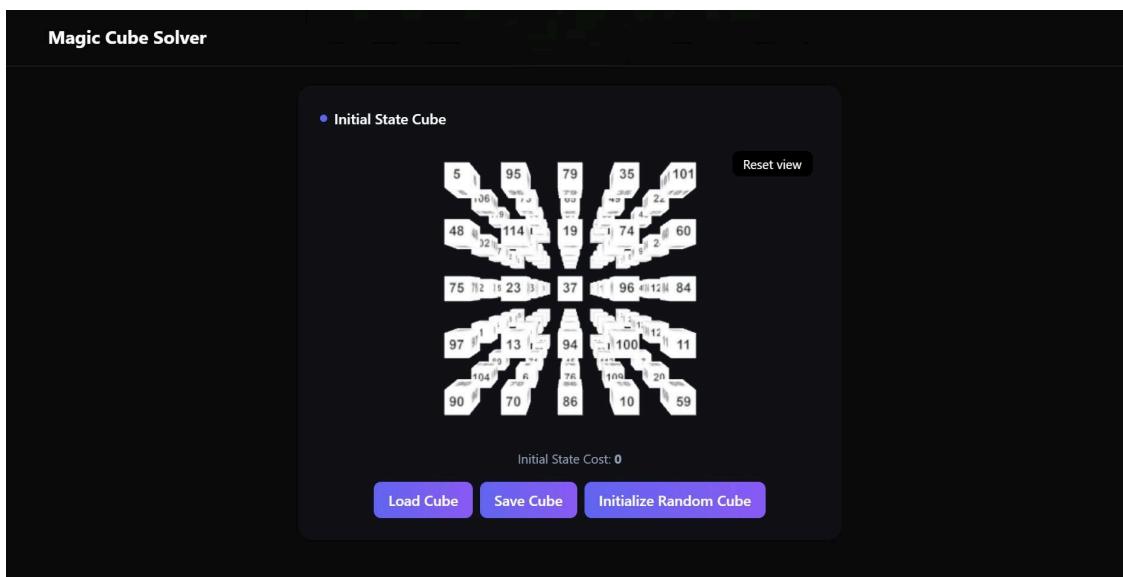
Hasil ini menandakan bahwa proses load berhasil, dan backend mampu mengembalikan konfigurasi cube yang sudah memenuhi kondisi optimal (`cost = 0`) untuk divisualisasikan dalam aplikasi.



Gambar 2.76 Eksperimen Save and Load (1)



Gambar 2.77 Eksperimen Save and Load (2)



Gambar 2.78 Eksperimen Save and Load (3)

BAB III

Kesimpulan dan Saran

A. Kesimpulan

Hasil eksperimen menunjukkan bahwa algoritma *Genetic* unggul dalam mencapai nilai *best cost* yang mendekati *global optima* dibandingkan algoritma lain untuk masalah *Diagonal Magic Cube*. Dengan mekanisme *crossover* dan *mutation*, algoritma ini mampu mengeksplorasi ruang solusi secara lebih luas dan menghindari jebakan *local optima*. Walaupun algoritma *Genetic* memiliki durasi eksekusi yang lebih lama karena ukuran populasi dan iterasi yang besar, hasil yang dicapai memberikan solusi yang lebih optimal dibandingkan algoritma lainnya.

Sebaliknya, algoritma *local search* seperti *Steepest Ascent Hill-climbing*, *Hill-climbing with Sideways Move*, *Stochastic*, dan *Simulated Annealing* menunjukkan hasil yang cukup baik tetapi kurang efektif dalam menghadapi jebakan *local optima*. *Simulated Annealing* menghasilkan solusi yang lebih baik daripada *Stochastic* dan *Random Restart* berkat mekanisme *annealing schedule* yang membantu eksplorasi. Namun, algoritma *Random Restart* dan *Stochastic* menunjukkan ketidakmampuan dalam mendekati *global optima* secara konsisten, meskipun keduanya memiliki waktu eksekusi yang cepat.

B. Saran

Dalam melakukan eksperimen lanjutan agar mendapatkan peningkatan pada hasil yang diperoleh, dapat dilakukan penyesuaian parameter pada algoritma *Genetic* yaitu ukuran populasi dan tingkat muasi untuk menemukan keseimbangan yang lebih optimal antara kualitas solusi dan waktu eksekusi. Tak hanya itu, peningkatan juga dapat dilakukan pada algoritma *Simulated Annealing* dengan mengoptimalkan *cooling schedule* agar dapat membantu algoritma lebih eksploratif dalam pencarian solusi tanpa menurunkan tingkat kecepatan.

Pembagian Tugas

NIM	Nama	Pembagian Tugas
18222026	Tamara Mayranda Lubis	<ul style="list-style-type: none"> - Mengerjakan algoritma bagian <i>sideways move</i> dan <i>simulated annealing</i> - Melakukan <i>testing</i> - Mengerjakan laporan bagian <i>objective function</i>, penjelasan <i>sideways move</i>, penjelasan hasil <i>testing</i>, pembahasan serta kesimpulan dan saran.
18222094	Yovanka Sandrina Maharaja	<ul style="list-style-type: none"> - Mengerjakan algoritma bagian <i>utils</i> dan <i>stochastic</i> - Membuat tampilan <i>frontend website</i> - Melakukan <i>testing</i> - Mengerjakan laporan bagian penjelasan <i>utils</i>, <i>stochastic</i>, dan penjelasan hasil <i>testing</i>
18222130	Bryan P. Hutagalung	<ul style="list-style-type: none"> - Mengerjakan algoritma bagian <i>utils</i> dan <i>stochastic</i> - Membuat program backend, media player, serta save dan load - Mengerjakan laporan bagian genetic, media player, serta save dan load
18222141	Yusril Fazri Mahendra	<ul style="list-style-type: none"> - Mengerjakan algoritma steepest ascent dan random restart - Membuat tampilan <i>frontend website</i> dan visualisasi cube - Mengerjakan laporan bagian random restart, dan genetic algorithm.

Referensi

Russell, S., & Norvig, P. (2020). Artificial intelligence: A modern approach (4th ed.). Pearson.

Institut Teknologi Bandung. (n.d.). Beyond classical search: Local search [PDF]. Retrieved from https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804818592_IF3170_Materi03_Seg01_BeyondClassicalSearch_LocalSearch.pdf

Institut Teknologi Bandung. (n.d.). Beyond classical search: Hill climbing [PDF]. Retrieved from https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804849395_IF3170_Materi3_Seg03_BeyondClassicalSearch_HillClimbing.pdf

Institut Teknologi Bandung. (n.d.). Beyond classical search: Simulated annealing [PDF]. Retrieved from https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804872404_IF3170_Materi03_Seg04_BeyondClassicalSearch_SimulatedAnnealing.pdf

Institut Teknologi Bandung. (n.d.). Beyond classical search: Genetic algorithm [PDF]. Retrieved from https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693969141836_IF3170_Materi03_Seg05_BeyondClassicalSearch.pdf

Wikipedia contributors. (n.d.). Magic cube. In Wikipedia, The Free Encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Magic_cube

GeeksforGeeks contributors. (n.d.). Introduction to hill climbing | Artificial intelligence. GeeksforGeeks. Retrieved from https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/?ref=header_outind

Magischvierkant.com. (n.d.). Magic features. Retrieved from <https://www.magischvierkant.com/three-dimensional-eng/magic-features/>

Trump, U. (n.d.). Magic cubes: Cubes 1. Retrieved from <https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>