

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



Disusun oleh :
Taufiq Ramadhan Ahmad / 18222060
Muhammad Kevinza Faiz / 18222072
Viktor Arsindiantoro S / 18222083
Timotius Vivaldi Gunawan / 18222091

Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung
Jl. Ganesha 10, Bandung 40132

DAFTAR ISI

ABSTRAK	3
I. Deskripsi Persoalan	3
II. Pembahasan	5
A. Objective Function	5
B. Implementasi Algoritma Local Search	19
1. Steepest Ascent Hill-Climbing	19
2. Hill Climbing with Sideways Move	21
3. Stochastic Hill Climbing	24
4. Random Restart Hill-Climbing	27
5. Simulated Annealing	29
6. Genetic Algorithm	33
C. Hasil Eksperimen dan Analisis	42
1. Steepest Ascent Hill-Climbing	43
Percobaan #1	43
Percobaan #2	44
Percobaan #3	46
2. Random Restart Hill-Climbing	49
Percobaan #1	49
Percobaan #2	52
Percobaan #3	56
3. Hill-Climbing with Sideways Move	59
Percobaan #1	59
Percobaan #2	61
Percobaan #3	63
4. Stochastic Hill Climbing	65
Percobaan #1	65
Percobaan #2	67
Percobaan #3	69
5. Simulated Annealing	71
Percobaan #1	72
Percobaan #2	73
Percobaan #3	75
6. Genetic Algorithm	77
Percobaan #1	77
Percobaan #2	80

Percobaan #3	82
Percobaan #4	84
Percobaan #5	86
Percobaan #6	88
Pembahasan	89
III. Kesimpulan dan Saran	95
IV. Pembagian Tugas	96
V. Referensi	98

ABSTRAK

Dalam konsep matematika, *magic cube* adalah kumpulan bilangan bulat yang disusun dalam pola $1 \times n \times n \times n$ tanpa pengulangan dengan n adalah panjang dari sisi kubus tersebut. Setiap bilangan tersusun menjadi *magic cube* sedemikian rupa sehingga jumlah bilangan pada setiap baris, kolom, tiang, diagonal ruang, dan diagonal pada suatu potongan bidang dari kubus adalah sama, yang disebut juga dengan *magic number*. Aplikasi dari *magic cube* tidak hanya terbatas pada konsep matematika murni, tetapi juga dapat diimplementasikan pada pengembangan algoritma optimisasi dan pencarian solusi, seperti *local search algorithm*.

Kata Kunci : *magic cube*, *magic number*, *local search algorithm*.

I. Deskripsi Persoalan

Terdapat sebuah *magic cube* dengan ukuran $5 \times 5 \times 5$ yang memiliki *initial state* adalah susunan angka 1 hingga 5^3 secara acak.

25	16	80	104	90					
115	98	4	1	97					
42	111	85	2	75					
66	72	27	102	48					
67	18	119	106	5					
67	18	119	106	5	5	48	75	13	70
116	17	14	73	95	95	114	23	94	86
40	50	81	65	79	79	19	37	100	10
56	120	55	49	35	35	74	96	11	59
36	110	46	22	101	101	60	84		

Jumlah bilangan pada setiap baris, kolom, tiang, diagonal ruang, dan diagonal pada suatu potongan bidang kubus bernilai sama yang disebut dengan *magic number*. Untuk menentukan *magic number* suatu kubus, dapat dicari dengan menggunakan rumus berikut :

$$\text{Magic Number} = \frac{n(n^3 + 1)}{2},$$

dengan n adalah panjang kubus, pada kasus ini n bernilai 5.

Kemudian, lakukan iterasi menggunakan algoritma *local search* dan langkah yang boleh dilakukan adalah dengan cara menukar posisi untuk 2 angka pada kubus tersebut (angka yang ditukar tidak harus bersebelahan). Akan tetapi, untuk *Genetic Algorithm* boleh dilakukan *swap* lebih dari 2 angka pada satu kesempatan yang sama.

Berikut adalah ketentuan yang harus dilakukan dalam pengeroaan tugas besar ini :

1. Mengimplementasikan 3 algoritma *local search*, yaitu salah satu jenis *hill climbing algorithm*, *simulated annealing*, dan *genetic algorithm*.
2. Jalankan setiap algoritma sebanyak 3 kali dan catat hal-hal berikut :
 - *State* awal dan akhir kubus.
 - Nilai *objective function* yang dicapai pada *state* akhir kubus.
 - Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati.
 - Durasi proses pencarian.
3. Pencatatan khusus untuk algoritma tertentu :
 - a. *Steepest Ascent Hill Climbing* dan *Stochastic Hill Climbing*
 - Banyak iterasi hingga proses pencarian berhenti.
 - b. *Hill Climbing with Sideways Move*
 - Banyak iterasi hingga proses pencarian berhenti.
 - Tambahkan parameter maximum sideways move sebagai parameter untuk banyak sideways move yang dapat dilakukan dalam proses pencarian. Jika sudah menyentuh maximum sideways move maka pencarian dihentikan.
 - c. *Random Restart Hill-Climbing*
 - Banyak *restart*.
 - Banyak iterasi per *restart*
 - Tambahkan parameter maximum restart sebagai parameter untuk banyak restart yang dapat dilakukan dalam proses pencarian. Jika sudah menyentuh maximum restart maka pencarian dihentikan.
 - d. *Simulated Annealing*
 - Plot $e^{\frac{\Delta E}{T}}$ terhadap banyak iterasi yang telah dilewati
 - Frekuensi ‘stuck’ di local optima

e. *Genetic Algorithm*

Algoritma ini dapat dijalankan dengan dua parameter yang dapat diubah, yaitu jumlah populasi dan banyak iterasi. Pertama, jumlah populasi akan dijadikan sebagai kontrol, sementara banyak iterasi akan divariasikan menjadi tiga nilai berbeda. Untuk setiap konfigurasi parameter, program akan dijalankan sebanyak tiga kali. Selanjutnya, banyak iterasi akan dijadikan sebagai kontrol, dan jumlah populasi akan divariasikan menjadi tiga nilai berbeda pula. Sama seperti sebelumnya, setiap konfigurasi parameter ini akan dijalankan sebanyak tiga kali. Berikut adalah hal yang harus dicatat pada *genetic algorithm* :

- *State* awal dan akhir kubus
- Nilai *objective function* akhir yang dicapai
- Plot nilai *objective function* terhadap banyak iterasi yang telah dilewati
- Jumlah populasi
- Banyak iterasi
- Durasi proses pencarian

II. Pembahasan

A. Objective Function

Objective function merupakan sebuah fungsi yang digunakan agar suatu algoritma yang ingin diimplementasikan mendapatkan suatu bentuk objektif. *Objective function* disini dibataskan dengan *constraints* dan variabel yang ingin ditentukan kemudian, fungsi objektif ini dapat berupa fungsi yang minimal atau memaksimalkan suatu objektif.

Pada penyelesaian masalah magic cube kita mengetahui bahwa terdapat sebuah penjumlahan deret aritmatika dalam rentang 1 hingga n^3 . Sebelumnya, diketahui juga bahwa aturan sebuah magic number muncul pada penjumlahan n^2 baris atau n^2 kolom atau n^2 tinggi. Disini kita mampu membuktikan bahwa mutlak n^2 baris jika dikalikan dengan sebuah konstanta magic akan menjadi penjumlahan seluruh angka yang terdapat dalam satu kubus

$$1 + 2 + 3 + \dots + n^3 = n^2 K$$

Menggunakan dekomposisi rumus penjumlahan deret maka menjadi :

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

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

Maka K atau konstanta dari sebuah magic number menjadi :

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

Dengan kasus persoalan nilai $n = 5$, maka magic number akan berjumlah 315. Melalui informasi ini kita mengetahui bahwa objektif utama adalah untuk memastikan seluruh syarat diatas akan memiliki jumlah magic number 315. Jika ditinjau, magic number menjadi objektif yang ingin dicapai untuk setiap syarat tersebut. Dengan menghitung semua kasus, maka terdapat 25 baris, 25 kolom, 25 tiang, 30 diagonal bidang, dan 4 diagonal ruang dengan total sejumlah 109 kasus. Dengan ini, untuk mengukur 109 observasi maka dirumuskan sebuah *objective function* sebagai berikut :

$$\begin{aligned}
 f(n) = & \sum_{i=1}^n \sum_{j=1}^n \left| M - \left(\sum_{k=1}^n \text{Cube}[i][j][k] \right) \right| + \sum_{i=1}^n \sum_{k=1}^n \left| M - \left(\sum_{j=1}^n \text{Cube}[i][j][k] \right) \right| + \\
 & \sum_{i=1}^n \sum_{k=1}^n \left| M - \left(\sum_{j=1}^n \text{Cube}[i][j][k] \right) \right| + \left| M - \left(\sum_{i=1}^n \text{Cube}[i][i][i] \right) \right| + \\
 & \left| M - \left(\sum_{i=1}^n \text{Cube}[i][i][n-1-i] \right) \right| + \left| M - \left(\sum_{i=1}^n \text{Cube}[i][n-1-i][n-1-i] \right) \right| + \\
 & \left| M - \left(\sum_{i=1}^n \text{Cube}[i][n-1-i][i] \right) \right| + \\
 & \sum_{k=1}^n \left(\left| M - \left(\sum_{i=1}^n \text{Cube}[i][i][k] \right) \right| + \left| M - \left(\sum_{i=1}^n \text{Cube}[i][n-1-i][k] \right) \right| \right)
 \end{aligned}$$

$M = \text{magic number constant}$

$n = \text{ukuran kubus}$

Objective function diatas merupakan penjumlahan selisih dari seluruh kasus yang ingin dicapai. Secara terurut diatas merupakan selisih antara magic number dengan baris,

kolom, tiang, diagonal ruang, diagonal ruang arah sebaliknya, dan 30 diagonal bidang. Pemilihan objective function ini dilakukan agar kami dapat melakukan objektif yang diminimalkan. Alasan kami memilih untuk diminimalkan karena dengan domain pencarian yang cukup besar untuk perfect pandiagonal magic cube ini sehingga menghitung selisih atau error menjadi alasan yang lebih feasible untuk dilakukan.

Selain itu, melalui pendekatan fungsi objektif yang menggunakan selisih mutlak dari atribut-atribut pada Magic Cube cukup efektif karena berfokus pada pengurangan nilai error di setiap atribut yang terlibat. Dengan mengurangi error secara bertahap, fungsi ini juga meningkatkan keakuratan dan kebenaran konfigurasi Magic Cube dengan berfokus pada atribut yang telah benar. Fungsi objektif ini secara langsung memetakan seberapa jauh solusi saat ini dari solusi ideal, baik dalam algoritma pencarian berbasis tetangga seperti Hill-Climbing dan Simulated Annealing, maupun dalam evolusi populasi dalam Genetic Algorithm. Hasilnya adalah proses optimasi yang bertujuan untuk meminimalkan selisih antara total penjumlahan elemen dalam dimensi-dimensi kubus dengan nilai target Magic Number. Hal ini secara efisien mengurangi error di setiap iterasi atau generasi. Secara matematis, fungsi objektif ini mengukur error sebagai deviasi dari target magic number. Nilai absolut memastikan bahwa deviasi dihitung secara positif, sementara penjumlahan elemen-elemen dari baris, kolom, dan diagonal memungkinkan evaluasi menyeluruh dari struktur kubus. Dengan demikian, algoritma dapat terus memperbaiki konfigurasi untuk mendekati solusi optimal.

Deskripsi Fungsi dan Kelas :

1) Kelas *Result*

Kelas ini merupakan representasi dari suatu state kubus dengan memiliki beberapa atribut berikut :

- a. *error*: Atribut yang digunakan untuk menyimpan nilai error dari kubus.
- b. *steps*: Atribut yang digunakan untuk menyimpan jumlah step atau perpindahan yang dilakukan dari swap yang sudah dilakukan oleh kubus.
- c. *time_taken*:

- d. *cube* : Atribut yang digunakan untuk menyimpan cube selama proses local search berlangsung
- e. *error_history* : Atribut yang digunakan untuk menyimpan nilai error history yang digunakan untuk saat perbandingan
- f. *iterasi*: Atribut berupa array dinamis untuk menyimpan banyak iterasi
- g. *objfunc*: Atribut berupa array dinamis untuk menyimpan nilai objective function dari setiap iterasi yang dilakukan
- h. *probability* : Atribut berupa array dinamis untuk menyimpan nilai e pada algoritma simulated annealing
- i. *Probabilityindex* : Atribut berupa array dinamis untuk menyimpan indeks iterasi pada nilai e algoritma simulated annealing
- j. *frekuensi_stuck* : Atribut berupa array dinamis untuk menyimpan frekuensi stuck saat menjalankan algoritma simulated annealing

```
struct Result {  
    double error;  
    int steps;  
    double time_taken;  
    std::vector<std::vector<std::vector<int>>> cube;  
    std::vector<double> error_history;  
    std::vector<double> iterasi;  
    std::vector<double> objfunc;  
    std::vector<double> probability;  
    std::vector<double> probabilityindex;  
    int frekuensi_stuck;  
};
```

2) generate_neighbors

Fungsi ini menerima parameter suatu cube dan akan mengembalikan semua neighbor dari cube tersebut dalam bentuk `std::vector<std::vector<std::vector<std::vector<int>>>`.

```
std::vector<std::vector<std::vector<std::vector<int>>>
generate_neighbors(const
std::vector<std::vector<std::vector<int>>>& cube) {
    std::vector<std::vector<std::vector<std::vector<int>>>>
neighbors;
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            for (int k = 0; k < 5; k++) {
                for (int l = i; l < 5; l++) {
                    for (int m = (l == i ? j : 0); m < 5; m++) {
                        for (int n = (l == i && m == j ? k + 1 :
0); n < 5; n++) {
                            if (i == l && j == m && k == n)
                                continue;
                            auto new_cube = cube;
                            std::swap(new_cube[i][j][k],
new_cube[l][m][n]);
                            neighbors.push_back(new_cube);
                        }
                    }
                }
            }
        }
    }
    return neighbors;
}
```

3) initialize_random_cube

Fungsi ini akan mengembalikan suatu cube dengan kondisi angka-angka di dalamnya sudah tersusun secara acak.

```
std::vector<std::vector<std::vector<int>>>
initialize_random_cube() {
    std::vector<int> numbers(N * N * N);
    std::iota(numbers.begin(), numbers.end(), 1);

    std::random_device rd;
    std::default_random_engine rng(rd());

    std::shuffle(numbers.begin(), numbers.end(), rng);
    std::vector<std::vector<std::vector<int>>> cube(N,
    std::vector<std::vector<int>>(N, std::vector<int>(N)));
    int idx = 0;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
                cube[i][j][k] = numbers[idx++];
    return cube;
}
```

4) jumlahSkor

Fungsi ini akan meninjau banyak baris, kolom, tiang dan diagonal bidang yang memiliki nilai sejumlah 315, lalu akan mengeluarkan output senilai tersebut.

```
int jumlahSkor(const std::vector<std::vector<std::vector<int>>>&
cube) {
    int sum = 0;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            int row_sum = 0, col_sum = 0, pillar_sum = 0;
            for (int k = 0; k < N; ++k) {
                row_sum += cube[i][j][k];
                col_sum += cube[i][k][j];
                if (i == j && k == i)
                    pillar_sum += cube[i][j][k];
            }
            sum += max(row_sum, col_sum);
            if (i == j)
                sum += pillar_sum;
        }
    }
    return sum;
}
```

```
        col_sum += cube[i][k][j];
        pillar_sum += cube[k][i][j];
    }
    if (row_sum == target_sum) sum++;
    if (col_sum == target_sum) sum++;
    if (pillar_sum == target_sum) sum++;
}
}

for (int k = 0; k < N; ++k) {
    int diag1 = 0, diag2 = 0;
    for (int i = 0; i < N; ++i) {
        diag1 += cube[i][i][k];
        diag2 += cube[i][N - 1 - i][k];
    }
    if (diag1 == target_sum) sum++;
    if (diag2 == target_sum) sum++;
}

for (int j = 0; j < N; ++j) {
    int diag1 = 0, diag2 = 0;
    for (int i = 0; i < N; ++i) {
        diag1 += cube[i][j][i];
        diag2 += cube[i][j][N - 1 - i];
    }
    if (diag1 == target_sum) sum++;
    if (diag2 == target_sum) sum++;
}

for (int i = 0; i < N; ++i) {
    int diag1 = 0, diag2 = 0;
    for (int j = 0; j < N; ++j) {
```

```
        diag1 += cube[i][j][j];
        diag2 += cube[i][j][N - 1 - j];
    }

    if (diag1 == target_sum) sum++;
    if (diag2 == target_sum) sum++;
}

int diag1 = 0, diag2 = 0, diag3 = 0, diag4 = 0;
for (int i = 0; i < N; ++i) {
    diag1 += cube[i][i][i];
    diag2 += cube[i][i][N - 1 - i];
    diag3 += cube[i][N - 1 - i][i];
    diag4 += cube[i][N - 1 - i][N - 1 - i];
}

if (diag1 == target_sum) sum++;
if (diag2 == target_sum) sum++;
if (diag3 == target_sum) sum++;
if (diag4 == target_sum) sum++;

return sum;
}
```

5) calculate_error

Fungsi ini menerima parameter sebuah *cube*, dan akan menghitung *error* dari tiap *constraint* yang ada, yang kemudian akan dijumlahkan semuanya untuk mendapatkan *error* total dari *state* suatu kubus.

```
double calculate_error(const
std::vector<std::vector<std::vector<int>>>& cube) {
    double M = (N * (pow(N, 3) + 1)) / 2;
    double error_baris = 0, error_kolom = 0, error_tiang = 0;
    double error_diagonal_ruang = 0, error_diagonal_bidang = 0;

    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            double row_sum = 0, col_sum = 0, pillar_sum = 0;
            for (int k = 0; k < N; ++k) {
                row_sum += cube[i][j][k];
                col_sum += cube[i][k][j];
                pillar_sum += cube[k][i][j];
            }
            error_baris += std::abs(M - row_sum);
            error_kolom += std::abs(M - col_sum);
            error_tiang += std::abs(M - pillar_sum);
        }
    }

    for (int k = 0; k < N; ++k) {
        double diag1 = 0, diag2 = 0;
        for (int i = 0; i < N; ++i) {
            diag1 += cube[i][i][k];
            diag2 += cube[i][N - 1 - i][k];
        }
        error_diagonal_bidang += std::abs(M - diag1) +
std::abs(M - diag2);
    }

    for (int j = 0; j < N; ++j) {
        double diag1 = 0, diag2 = 0;
```

```
        for (int i = 0; i < N; ++i) {
            diag1 += cube[i][j][i];
            diag2 += cube[i][j][N - 1 - i];
        }
        error_diagonal_bidang += std::abs(M - diag1) +
        std::abs(M - diag2);
    }

    for (int i = 0; i < N; ++i) {
        double diag1 = 0, diag2 = 0;
        for (int j = 0; j < N; ++j) {
            diag1 += cube[i][j][j];
            diag2 += cube[i][j][N - 1 - j];
        }
        error_diagonal_bidang += std::abs(M - diag1) +
        std::abs(M - diag2);
    }

    double diag1 = 0, diag2 = 0, diag3 = 0, diag4 = 0;
    for (int i = 0; i < N; ++i) {
        diag1 += cube[i][i][i];
        diag2 += cube[i][i][N - 1 - i];
        diag3 += cube[i][N - 1 - i][i];
        diag4 += cube[i][N - 1 - i][N - 1 - i];
    }
    error_diagonal_ruang = std::abs(M - diag1) + std::abs(M -
    diag2) + std::abs(M - diag3) + std::abs(M - diag4);

    return error_baris + error_kolom + error_tiang +
    error_diagonal_ruang + error_diagonal_bidang;
}
```

6) Swap_elements

Fungsi ini akan melakukan proses salin data dari kubus ke variabel baru, lalu akan menukar elemen pada posisi tertentu dan mengembalikan kubus baru dengan elemen yang ditukar.

```
std::vector<std::vector<std::vector<int>>> swap_elements(
    const std::vector<std::vector<std::vector<int>>>& cube, int
    x1, int y1, int z1, int x2, int y2, int z2) {
    auto new_cube = cube;
    std::swap(new_cube[x1][y1][z1], new_cube[x2][y2][z2]);
    return new_cube;
}
```

7) isMagicRow

Fungsi ini akan menerima suatu *cube* dan juga lokasi *row* yang ingin di cek, kemudian akan mengembalikan apakah termasuk *magic row* atau bukan dengan tipe data boolean.

```
bool isMagicRow(const std::vector<std::vector<std::vector<int>>>
&cube, int level, int row) {
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        sum += cube[level][row][i];
    }
    return sum == target_sum;
}
```

8) isMagicColumn

Fungsi ini akan menerima suatu *cube* dan juga lokasi *column* yang ingin di cek, kemudian akan mengembalikan apakah termasuk *magic column* atau bukan dengan tipe data boolean.

```
bool isMagicColumn(const
```

```
std::vector<std::vector<std::vector<int>>> &cube, int level, int
column) {
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        sum += cube[level][column][i];
    }
    return sum == target_sum;
}
```

9) isMagicPillar

Fungsi ini akan menerima suatu *cube* dan juga lokasi *pillar* yang ingin di cek, kemudian akan mengembalikan apakah termasuk *magic pillar* atau bukan dengan tipe data boolean.

```
bool isMagicPillar(const
std::vector<std::vector<std::vector<int>>> &cube, int row, int
column) {
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        sum += cube[i][row][column];
    }
    return sum == target_sum;
}
```

10) isMagicDiagonal

Fungsi ini akan menerima suatu *cube* dan juga lokasi diagonal yang ingin di cek, kemudian akan mengembalikan apakah termasuk *magic diagonal* atau bukan dengan tipe data boolean.

```
bool isMagicDiagonal(const
std::vector<std::vector<std::vector<int>>> &cube, int level) {
    int sum1 = 0, sum2 = 0;
```

```
    for (int i = 0; i < 5; ++i) {  
        sum1 += cube[level][i][i];  
        sum2 += cube[level][i][4 - i];  
    }  
    return (sum1 == target_sum) || (sum2 == target_sum);  
}
```

11) isMagic3DDiagonal

Fungsi ini akan melakukan pengecekan terhadap jumlah nilai pada diagonal utama dalam kubus, dengan dibandingkan dengan jumlah elemen yang sama yaitu pada target sum.

```
bool isMagic3DDiagonal(const  
std::vector<std::vector<std::vector<int>>> &cube) {  
    int sum1 = 0, sum2 = 0, sum3 = 0, sum4= 0;  
    for (int i = 0; i < 5; ++i) {  
        sum1 += cube[i][i][i];  
        sum2 += cube[i][i][4 - i];  
        sum1 += cube[i][4-i][i];  
        sum2 += cube[i][4-i][4 - i];  
    }  
    return (sum1 == target_sum) || (sum2 == target_sum);  
}
```

12) countConstraints

```
int countConstraints(const  
std::vector<std::vector<std::vector<int>>> &cube, int x, int y,  
int z) {  
    int count = 0;  
    if (isMagicRow(cube, x, y)) count++;  
    if (isMagicColumn(cube, x, z)) count++;  
    if (isMagicDiagonal(cube, x)) count++;  
    if (isMagicPillar(cube, y, z)) count++;  
    if (isMagic3DDiagonal(cube)) count++;  
    return count;  
}
```

13) displayGraph

Fungsi ini akan memberikan tampilan dari sebuah grafik yang akan menggambarkan persebaran dari nilai objective function untuk setiap proses iterasi. Fungsi ini menginisialisasi lingkungan Python.

```
namespace plt = matplotlibcpp;  
void displayGraph(Result result) {  
    _putenv_s("PYTHONHOME", "C:/Python312");  
    _putenv_s("PYTHONPATH",  
    "C:/Python312/Lib;C:/Python312/Lib/site-packages");  
    Py_Initialize();  
  
    plt::plot(result.objfunc, result.iterasi);  
    plt::xlabel("Iteration");  
    plt::ylabel("Objective Function (Error)");  
    plt::title("Objective Function over Iterations");  
    plt::show();  
    plt::clf();  
    plt::close();  
    Py_Finalize();
```

}

B. Implementasi Algoritma *Local Search*

1. Steepest Ascent Hill-Climbing

Steepest Ascent Hill Climbing adalah algoritma *local search* yang selalu memilih langkah yang memberikan peningkatan nilai terbesar dari semua *neighbor* yang tersedia. Pada Algoritma ini, digunakan pendekatan matematis *heuristic search*. *Heuristic search* memberikan sebuah optimalisasi pada sebuah fungsi dengan memaksimal atau meminimalkan input dengan memberikan output solusi dengan waktu yang efisien. *Hill climbing* yang sederhana menggunakan metode untuk menganalisis neighboring nodes satu persatu dan melakukan perbandingan setiap state agar selalu optimal dan berhenti melakukan iterasi ketika mencapai sebuah *flat*.

Secara umum, struktur algoritma Steepest Ascent Hill Climbing menggunakan metode pencarian solusi dengan cara mencari tetangga terbaik (yang memiliki nilai error terendah) dari solusi saat ini dan beralih ke solusi tersebut. Proses ini berulang hingga tidak ada solusi yang lebih baik ditemukan, atau solusi optimal tercapai dengan error 0.

Untuk menjalankan struktur algoritma tersebut, Steepest Ascent Hill Climbing akan dilengkapi sebuah *objective function* untuk melakukan pemeriksaan error. Dalam hal ini, pemeriksaan *error* menggunakan *objective function* yang menghitung seberapa baik atau buruk solusi yang ada. *Error* ini dihitung dengan membandingkan hasil solusi saat ini dengan solusi ideal atau target. Pada setiap langkah, setelah melakukan pertukaran atau perubahan elemen dalam ruang pencarian, *objective function* akan menghitung nilai *error* dari solusi baru tersebut.

```
Result
steepest_ascent_hill_climbing(std::vector<std::vector<std::vector<int>>> cube) {
    Result result;
    result(cube = cube;
    result.error = calculate_error(result(cube));
    result.steps = 0;
    bool improvement = true;
    auto start_time = std::chrono::high_resolution_clock::now();
    result.objfunc.push_back(result.error);
    result.iterasi.push_back(result.steps + 1);
    while (improvement) {
        improvement = false;
        auto best_cube = result(cube);
        double best_error = result.error;
        for (int i1 = 0; i1 < N * N * N - 1; ++i1) {
            int x1 = i1 / (N * N), y1 = (i1 / N) % N, z1 = i1 %
N;
            for (int i2 = i1 + 1; i2 < N * N * N; ++i2) {
                int x2 = i2 / (N * N), y2 = (i2 / N) % N, z2 =
i2 % N;
                auto neighbor = swap_elements(result(cube, x1,
y1, z1, x2, y2, z2));
                double neighbor_error =
calculate_error(neighbor);
                if (neighbor_error < best_error) {
                    std::cout << "Error: " << neighbor_error <<
std::endl;
                    std::cout << "Jumlah h: " <<
jumlahSkor(neighbor) << std::endl;
                    best_error = neighbor_error;
                    best_cube = neighbor;
                }
            }
        }
    }
}
```

```
        }
    }

    if (best_error < result.error) {
        result(cube = best_cube;
        result.error = best_error;
        result.steps++;
        improvement = true;
    }

    result.objfunc.push_back(result.error);
    result.iterasi.push_back(result.steps + 1);
    if (result.error == 0) break;
}

auto end_time = std::chrono::high_resolution_clock::now();
result.time_taken = std::chrono::duration<double>(end_time -
start_time).count();

return result;
}
```

2. Hill Climbing with Sideways Move

Algoritma *Hill-Climbing with Sideways Move* merupakan salah satu jenis dari algoritma *hill climbing* dan memiliki cara kerja yang hampir sama dengan *steepest ascent hill-climbing*. Pada *steepest ascent hill-climbing*, algoritma akan selalu dan hanya memilih tetangga yang memiliki nilai *objective function* yang lebih baik. Akan tetapi, pada *stochastic hill-climbing* diperbolehkan untuk melakukan *sideway move* yaitu pindah ke tetangga yang memiliki nilai *objective function* sama dengan nilai *objective function current*. Harapannya dengan mengimplementasikan teknik tersebut, algoritma ini dapat menghindari kasus terjebak di *local optimal*.

Pada kasus *magic cube* 5x5x5 ini, langkah pertama dalam pengimplementasiannya adalah dengan men-*generate* sebuah kubus awal sebagai *initial state*. Selanjutnya dilakukan iterasi dengan mencoba menemukan tetangga terbaik dengan cara mengevaluasi setiap pasangan pertukaran angka pada *magic cube*. Jika ditemukan tetangga dengan nilai error lebih kecil, maka otomatis *state* akan langsung pindah ke tetangga tersebut. Jika tidak ditemukan tetangga dengan nilai error lebih kecil namun terdapat tetangga dengan nilai error yang sama dengan *current*, maka algoritma dapat berpindah ke *state* tersebut yang disebut dengan *sideway move*. Diberikan juga sebuah parameter “*max_sideways_move*” sebagai batas *sideway move* yang dapat dilakukan algoritma sebelum dilakukan terminasi program. Algoritma ini akan berhenti ketika mencapai solusi optimal, yaitu error bernilai 0 atau mencapai batas *sideway move* yang ditentukan. Berikut adalah fungsi yang diimplementasikan dalam algoritma *hill climbing with sideways move*.

```
Result
hill_climbing_with_sideway_moves(std::vector<std::vector<std::vector<int>>> cube, int max_sideways_moves) {
    Result result;
    result(cube = cube;
    result.error = calculate_error(result(cube));
    result.steps = 0;

    bool improvement = true;
    int sideway_moves = 0;
    auto start_time = std::chrono::high_resolution_clock::now();

    while (improvement) {
        improvement = false;
        auto best_cube = result(cube);
        double best_error = result.error;
```

```
        for (int i1 = 0; i1 < N * N * N - 1; ++i1) {
            int x1 = i1 / (N * N), y1 = (i1 / N) % N, z1 = i1 % N;
            for (int i2 = i1 + 1; i2 < N * N * N; ++i2) {
                int x2 = i2 / (N * N), y2 = (i2 / N) % N, z2 = i2 % N;
                auto neighbor = swap_elements(result(cube, x1, y1, z1, x2, y2, z2));
                double neighbor_error =
                    calculate_error(neighbor);

                if (neighbor_error < best_error) {
                    std::cout << "Error: " << neighbor_error << std::endl;
                    std::cout << "Jumlah h: " << jumlahSkor(neighbor) << std::endl;
                    best_error = neighbor_error;
                    best_cube = neighbor;
                    sideway_moves = 0;
                } else if (neighbor_error == best_error &&
                           sideway_moves < max_sideways_moves) {
                    best_cube = neighbor;
                    sideway_moves++;
                }
            }
        }

        if (best_error <= result.error) {
            if (best_error < result.error) {
                sideway_moves = 0;
            } else {
                sideway_moves++;
            }
        }
    }
}
```

```
    }

    result(cube = best_cube;
    result.error = best_error;
    result.steps++;
    improvement = true;

    result.error_history.push_back(result.error);
}

if (result.error == 0 || sideway_moves >=
max_sideways_moves) {
    break;
}
}

auto end_time = std::chrono::high_resolution_clock::now();
result.time_taken = std::chrono::duration<double>(end_time -
start_time).count();
return result;
}
```

3. Stochastic Hill Climbing

Algoritma *Stochastic Hill Climbing* juga merupakan salah satu jenis *hill climbing* yang bekerja berawal dari sebuah *initial state* dan selanjutnya menghasilkan random *successor*. Algoritma ini hanya akan mengevaluasi *random successor* pada setiap iterasi. Apabila *successor* yang dihasilkan memiliki *value* yang lebih baik dari *current state*, maka algoritma ini akan bergerak ke *successor* tersebut. Algoritma ini juga memiliki batasan jumlah iterasi yang ditentukan di awal sehingga algoritma ini dapat berhenti setelah batas tersebut tercapai, terlepas

dari tercapainya *goal* atau tidak. Pada algoritma ini, kami mengimplementasikan sebuah fungsi *random_neighbor* yang akan men-generate neighbor untuk dipilih oleh algoritma. Berikut adalah pengimplementasiannya :

```
std::vector<std::vector<std::vector<int>>> random_neighbor(const
std::vector<std::vector<std::vector<int>>>& cube) {
    int x1, y1, z1, x2, y2, z2;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, N - 1);

    x1 = dis(gen); y1 = dis(gen); z1 = dis(gen);
    x2 = dis(gen); y2 = dis(gen); z2 = dis(gen);

    while (x1 == x2 && y1 == y2 && z1 == z2) {
        x2 = dis(gen); y2 = dis(gen); z2 = dis(gen);
    }

    return swap_elements(cube, x1, y1, z1, x2, y2, z2);
}
```

```
Result
stochastic_hill_climbing(std::vector<std::vector<std::vector<int
>>> cube, int max_iterations) {
    Result result;
    result(cube = cube;
    result.error = calculate_error(result(cube));
    result.steps = 0;

    std::vector<double> error_history;
```

```
        error_history.push_back(result.error);

        auto start_time = std::chrono::high_resolution_clock::now();

        for (result.steps = 0; result.steps < max_iterations;
++result.steps) {
            auto neighbor = random_neighbor(result(cube));
            double neighbor_error = calculate_error(neighbor);

            if (neighbor_error < result.error) {
                result(cube) = neighbor;
                result.error = neighbor_error;
                error_history.push_back(result.error);
                std::cout << "Stochastic Hill Climbing" <<
std::endl;
                std::cout << "Iterasi ke-" << result.steps+1 <<
std::endl;
                std::cout << "Jumlah h: " << jumlahSkor(result(cube))
<< std::endl;
                std::cout << "Jumlah Error: " << result.error <<
std::endl;

                if (result.error == 0) {
                    break;
                }
            }
        }

        auto end_time = std::chrono::high_resolution_clock::now();
        result.time_taken = std::chrono::duration<double>(end_time -
start_time).count();
        result.error_history = error_history;
```

```
    std::cout << "Banyak iterasi : " << result.steps <<
std::endl;
    return result;
}
```

4. Random Restart Hill-Climbing

Random restart Hill Climbing merupakan salah satu varian dari algoritma jenis hill climbing yang berawal dari sebuah *initial state* yang kemudian akan bergerak menuju *goal* yang telah ditetapkan sebelumnya dengan melakukan perbaikan lokal. Apabila pencarian mencapai *local maximum* tanpa menemukan *goal*, maka algoritma akan secara otomatis melakukan *restart* dengan mengulang *random state* baru dan memulai pencarian lagi. Algoritma ini akan berhenti jika sudah mendapatkan *goal* atau sesuai dengan jumlah maksimal *restart* yang sudah ditentukan.

```
Result
random_restart_hill_climbing(std::vector<std::vector<std::vector<int>>> cube, int max_restarts) {
    Result best_result;
    best_result.error = std::numeric_limits<double>::infinity();
    std::vector<Result> result_history;
    auto start_time = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < max_restarts; i++) {
        Result current_result;

        if (i == 0) {
            current_result =
steepest_ascent_hill_climbing(cube);
```

```
        } else {
            std::vector<std::vector<std::vector<int>>> new_state
= initialize_random_cube();
            current_result =
steepest_ascent_hill_climbing(new_state);
        }
        if (current_result.error < best_result.error) {
            best_result = current_result;
        }

best_result.error_history.push_back(current_result.error);
        result_history.push_back(current_result);
        if (best_result.error == 0) {
            break;
        }
    }
    std::cout << "=====RESTART"
LOG=====
for(int j = 0; j < max_restarts; j++ ){
    std::cout << "RESTART KE- " << j + 1 << std::endl;
    std::cout << "STEPS TAKEN : " <<
result_history[j].steps << std::endl;
    std::cout << "CURRENT ERROR : " <<
result_history[j].error << std::endl;
    std::cout <<
"====="
= " << std::endl;
}
```

```
        auto end_time = std::chrono::high_resolution_clock::now();
        best_result.time_taken =
    std::chrono::duration<double>(end_time - start_time).count();
        return best_result;
}
```

5. Simulated Annealing

Simulated Annealing merupakan algoritma optimasi yang menggabungkan eksplorasi ruang pencarian solusi dengan mekanisme probabilistik yang memungkinkan penerimaan solusi yang lebih buruk pada tahap awal pencarian untuk menghindari keterjebakan pada local minima. Namun, seiring berjalannya waktu (atau iterasi), algoritma semakin membatasi penerimaan solusi yang lebih buruk, memfokuskan pencarian pada perbaikan secara bertahap menuju solusi yang lebih baik.

Secara Umum, Struktur algoritma simulated annealing sangat mirip dengan hill climbing, tetapi perbedaannya ketimbang selalu memilih langkah terbaik, algoritma ini memilih langkah secara acak. Jika langkah tersebut memperbaiki solusi, ia selalu diterima. Namun, jika langkah tersebut memperburuk keadaan, langkah itu tetap bisa diterima dengan probabilitas tertentu yang lebih kecil dari 1. Probabilitas ini berkurang secara eksponensial dengan "tingkat buruknya".

Untuk kasus Magic Cube pada tugas besar ini, *Simulated Annealing* yang kami buat sesuai dengan konsep utama, berawal dengan meng-generate inisialisasi *magic cube* secara random dengan suhu tinggi yang kami berikan bernilai 1000 dengan memiliki cooling rate bernilai 0.99999. Selain itu, *Simulated Annealing* memiliki function yang mendukung yaitu function scheduling untuk menghitung suhu (temperature) pada setiap iterasi selama proses pendinginan (cooling). Selanjutnya, pada setiap iterasi yang dilakukan apabila neighbours

lebih baik (error lebih kecil), maka solusi neighbours diambil sebagai solusi baru. Jika tidak, solusi lebih buruk juga dapat diterima, tetapi dengan probabilitas yang

semakin menurun seiring berjalannya waktu yaitu $e^{\frac{\Delta E}{T}}$ lebih besar dari random nilai probabilitas dari 0 sampai 1.

```
#include "sa.hpp"

double scheduling(double t0, double cooling_rate, int iteration)
{
    return t0 * pow(cooling_rate, iteration);
}

Result
simulated_annealing(std::vector<std::vector<std::vector<int>>>
cube) {
    Result result;
    result(cube = cube;
    result.error = calculate_error(result(cube));
    result.steps = 0;
    result.frekuensi_stuck = 0;

    double cooling_rate = 0.99999;
    double temp = 1000.0;
    int max_iterate = 100000;
    auto start_time = std::chrono::high_resolution_clock::now();
    std::vector<std::vector<std::vector<int>>> best_cube =
result(cube;
    double best_error = result.error;

    result.objfunc.push_back(result.error);
    result.iterasi.push_back(result.steps + 1);
    result.probability.push_back(1);
```

```
result.probabilityindex.push_back(result.steps+1);
for (int i = 1; i < max_iterate; ++i) {
    temp = scheduling(temp, cooling_rate, i);

    int x1 = rand() % N;
    int y1 = rand() % N;
    int z1 = rand() % N;
    int x2 = rand() % N;
    int y2 = rand() % N;
    int z2 = rand() % N;
    while (x1==x2 && y1==y2 && z1==z2)
    {
        x2 = rand() % N;
        y2 = rand() % N;
        z2 = rand() % N;
    }
    auto neighbor = swap_elements(result(cube, x1, y1, z1,
x2, y2, z2);
    double neighbor_error = calculate_error(neighbor);
    double selisih = neighbor_error - result.error;
    double prob = std::exp(-selisih / temp);
    if (selisih < 0) {
        result(cube = neighbor;
        result.error = neighbor_error;
        result.steps += 1;
        result.probability.push_back(1);
        result.probabilityindex.push_back(i);
    } else {
        if (static_cast<double>(rand()) / RAND_MAX < prob) {
            result(cube = neighbor;
            result.error = neighbor_error;
```

```
        result.steps += 1;
        result.probability.push_back(prob);
        result.probabilityindex.push_back(i);
        result.frekuensi_stuck += 1;
    }
}

std::cout << "Jumlah h: " << jumlahSkor(result(cube)) <<
std::endl;
std::cout << "Jumlah Error: " <<
calculate_error(result(cube)) << std::endl;

if (result.error == 0) {
    break;
}
result.objfunc.push_back(result.error);
result.iterasi.push_back(i);

}
auto end_time = std::chrono::high_resolution_clock::now();
result.time_taken = std::chrono::duration<double>(end_time -
start_time).count();

return result;
}
```

Deskripsi Fungsi :

1) Function Scheduling

Fungsi scheduling dalam algoritma Simulated Annealing digunakan untuk menghitung suhu pada setiap iterasi dengan rumus

$$T(i) = T_0 \times (\text{cooling rate})^i$$

Dengan T_0 adalah suhu awal, `cooling_rate` mengontrol laju pendinginan, dan i adalah iterasi saat itu. Suhu yang tinggi di awal memungkinkan algoritma menjelajahi ruang solusi secara luas, sementara suhu yang menurun secara bertahap selama iterasi mengurangi kemungkinan menerima solusi yang lebih buruk. Fungsi ini membantu algoritma untuk menyeimbangkan eksplorasi dan eksplotasinya, dengan fokus beralih ke solusi yang lebih optimal seiring berjalananya waktu.

```
double scheduling(double t0, double cooling_rate, int iteration) {
    return t0 * pow(cooling_rate, iteration);
}
```

6. Genetic Algorithm

Genetic Algorithm merupakan algoritma yang terinspirasi dari prinsip seleksi alam dan juga evolusi genetik dengan melakukan beberapa tahapan. Dimulai dari meng-*generate* populasi individu acak dimana setiap individu dinilai berdasarkan *fitness function* untuk mengevaluasi kualitas solusinya. Individu dengan *fitness* terbaik dipilih untuk bereproduksi melalui proses *crossover*, di mana bagian-bagian gen dari dua orang tua digabungkan untuk membentuk individu baru. Beberapa individu juga mengalami mutasi untuk menjaga keragaman genetik. Proses seleksi, *crossover*, dan mutasi berulang membentuk generasi baru, yang diharapkan lebih baik dari generasi sebelumnya. Algoritma ini terus berlanjut hingga mencapai kondisi berhenti, seperti tercapainya jumlah generasi maksimum atau solusi optimal ditemukan.

Untuk kasus Magic Cube pada tugas besar ini, Genetic Algorithm yang kami buat sesuai dengan konsep utama, namun dengan beberapa penyesuaian. Genetic Algorithm untuk Magic Cube kami berawal dengan meng-*generate* populasi individu dengan melakukan randomize kromosom terlebih dahulu.

Kromosom merupakan representasi dari angka-angka yang ada di Cube nantinya. Setelah itu, tiap angka yang ada di kromosom kemudian akan di mapping ke cube untuk dihitung nilai fitnessnya dengan memanfaatkan fungsi magicSum yang digunakan di algoritma lain. Berikut adalah persamaan untuk menghitung nilai fitness dari suatu Cube :

$$\text{Fitness Function} = \frac{1}{1 + (\text{Calculate Error})} ,$$

Nilai fitness dihitung dengan pembalikan *Calculate Error* agar semakin kecil selisih dari target, semakin tinggi nilai *fitness* yang dihasilkan. Pembalikan ini memberi keunggulan pada individu yang lebih dekat ke solusi. Penambahan 1 pada penyebut berfungsi untuk menghindari pembagian 0 jika *Calculate Error* bernilai 0, yaitu saat individu telah mencapai solusi optimal.

```
Result geneticAlgorithm(int N, int populationSize, int
maxGenerations, double crossoverRate, double mutationRate) {
    Result result;
    int target_sum = target_sum;
    auto population = initializePopulation(populationSize, N,
target_sum);

    Individual bestIndividual = population[0];
    auto startTime = std::chrono::high_resolution_clock::now();

    for (int generation = 0; generation < maxGenerations;
++generation) {
        std::vector<Individual> newPopulation;
        newPopulation.push_back(bestIndividual);

        while (newPopulation.size() < populationSize) {
            auto parent1 = selectParent(population);
            auto parent2 = selectParent(population);
```

```
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_real_distribution<> distribution(0.0,
1.0);

        auto children = distribution(gen) < crossoverRate ?
performCrossover(parent1, parent2) : std::make_pair(parent1,
parent2);
        performMutation(children.first, mutationRate);
        performMutation(children.second, mutationRate);
        children.first.calculateFitness(N, target_sum);
        children.second.calculateFitness(N, target_sum);
        newPopulation.push_back(children.first);
        if (newPopulation.size() < populationSize)
            newPopulation.push_back(children.second);
    }
    population = newPopulation;

    for (const auto& individual : population) {
        if (individual.fitness > bestIndividual.fitness)
            bestIndividual = individual;
    }
    result.error = calculate_error(bestIndividual.cube);
    result.error_history.push_back(result.error);
    result.steps = generation + 1;

    int currentScore = jumlahSkor(bestIndividual.cube);
    std::cout << "Generation: " << generation + 1
        << ", Score: " << currentScore
        << ", Error: " << result.error
        << std::endl;
```

```
        if (currentScore == N * N * 3 + N * 2 + 4)
            break;
    }

    auto endTime = std::chrono::high_resolution_clock::now();
    result.time_taken = std::chrono::duration<double>(endTime -
startTime).count();
    result(cube = bestIndividual(cube);

    return result;
}
```

Deskripsi Fungsi dan Kelas :

1) Kelas Individu

Kelas ini mewakili satu kemungkinan solusi untuk susunan kubus.

Setiap objek Individu memiliki dua atribut utama, yaitu :

a. Kromosom (std::vector<int>)

Array satu dimensi yang berisi angka-angka unik untuk mewakili gen dalam susunan kubus. Misalnya, untuk kubus 5x5x5, kromosom ini berisi angka dari 1 hingga 125.

b. Fitness (double)

Nilai yang menunjukkan seberapa baik susunan ini mendekati magic cube yang sempurna. *Fitness* ini dihitung berdasarkan seberapa dekat jumlah baris, kolom, dan diagonal dalam kubus mendekati nilai target.

c. Cube (std::vector<std::vector<std::vector<int>>>)

Menyimpan representasi fisik dari kubus yang sudah di mapping dari kromosom.

```
Individual::Individual(const std::vector<int>&
chromosome, int N, int target_sum) :
chromosome(chromosome), fitness(0.0) {
    cube =
    std::vector<std::vector<std::vector<int>>>(N,
    std::vector<std::vector<int>>(N, std::vector<int>(N)));
    int index = 0;
    for (int layer = 0; layer < N; ++layer)
        for (int row = 0; row < N; ++row)
            for (int col = 0; col < N; ++col)
                cube[layer][row][col] =
chromosome[index++];
    calculateFitness(N, target_sum);
}
```

2) Fungsi calculateFitness

Fungsi ini akan mengkalkulasikan fitness dari suatu kromosom dengan menggunakan fitness function diatas.

```
void Individual::calculateFitness(int N, int target_sum) {
    fitness = 1.0 / (1.0 + calculate_error(cube));
}
```

3) Fungsi initializePopulation

Fungsi ini membentuk populasi awal yang terdiri dari individu-individu dengan kromosom acak. Fungsi ini menghasilkan std::vector<Individu>, yaitu kumpulan objek Individu yang masing-masing memiliki kromosom acak untuk memulai pencarian. Kromosom diacak dengan menggunakan fungsi std::shuffle untuk membuat kombinasi acak dari angka-angka.

```
std::vector<Individual> initializePopulation(int
populationSize, int N, int target_sum) {
    std::vector<Individual> population;
    std::vector<int> baseChromosome(N * N * N);
    for (int i = 0; i < N * N * N; ++i)
        baseChromosome[i] = i + 1;

    std::random_device rd;
    std::mt19937 g(rd());
    for (int i = 0; i < populationSize; ++i) {
        std::vector<int> chromosome = baseChromosome;
        std::shuffle(chromosome.begin(), chromosome.end(), g);
        population.emplace_back(chromosome, N,
target_sum);
    }
    return population;
}
```

4) Fungsi selectParent

Fungsi ini memilih satu individu dari populasi untuk bereproduksi berdasarkan nilai *fitness*. Fungsi ini mengembalikan satu objek Individu yang memiliki peluang lebih besar untuk terpilih jika *fitness*-nya tinggi. Seleksi ini membantu algoritma mempertahankan solusi terbaik dalam proses pencarian.

```
Individual selectParent(const std::vector<Individual>&
population) {
    double totalFitness = 0.0;
    for (const auto& individual : population)
        totalFitness += individual.fitness;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double>
distribution(0.0, totalFitness);
    double randomValue = distribution(gen);
    double cumulativeFitness = 0.0;
    for (const auto& individual : population) {
        cumulativeFitness += individual.fitness;
        if (cumulativeFitness >= randomValue)
            return individual;
    }
    return population.back();
}
```

5) Fungsi performCrossover

Fungsi ini menggabungkan gen dari dua individu (tipe Individu) dan menghasilkan dua anak baru. Hasil fungsi ini adalah std::pair<Individu, Individu>, yaitu dua objek Individu yang mewakili dua anak baru hasil penggabungan gen orang tua. *Crossover* memastikan anak-anak memiliki kombinasi gen yang berbeda tanpa angka berulang.

```
std::pair<Individual, Individual> performCrossover(const Individual& parent1, const Individual& parent2) {
    int size = parent1.chromosome.size();
    std::vector<int> childChromosome1(size),
    childChromosome2(size);
    std::vector<int> indices(size, -1);
    std::vector<bool> visited(size, false);
    int cycle = 1;
    for (int i = 0; i < size; ++i) {
        if (!visited[i]) {
            int currentIndex = i;
            do {
                indices[currentIndex] = cycle;
                visited[currentIndex] = true;
                int value =
parent2.chromosome[currentIndex];
                currentIndex =
std::distance(parent1.chromosome.begin(),
std::find(parent1.chromosome.begin(),
parent1.chromosome.end(), value));
            } while (currentIndex != i);
            cycle++;
        }
    }
    for (int i = 0; i < size; ++i) {
        if (indices[i] % 2 == 1) {
            childChromosome1[i] = parent1.chromosome[i];
            childChromosome2[i] = parent2.chromosome[i];
        } else {
            childChromosome1[i] = parent2.chromosome[i];
            childChromosome2[i] = parent1.chromosome[i];
        }
    }
}
```

```
    }

    return {Individual(childChromosome1,
parent1(cube.size(), target_sum),
        Individual(childChromosome2,
parent1(cube.size(), target_sum)};
```

```
}
```

6) Fungsi performMutation

Fungsi ini mengubah secara acak sebagian kecil dari kromosom individu dengan cara menukar posisi dua gen dalam kromosom. Fungsi ini menerima parameter Individu yang akan dimutasi dan probabilitas mutasi (double). Mutasi membantu menjaga keragaman genetik dalam populasi, mencegah algoritma terjebak pada solusi lokal.

```
void performMutation(Individual& individual, double
mutationRate) {

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> distribution(0.0,
1.0);

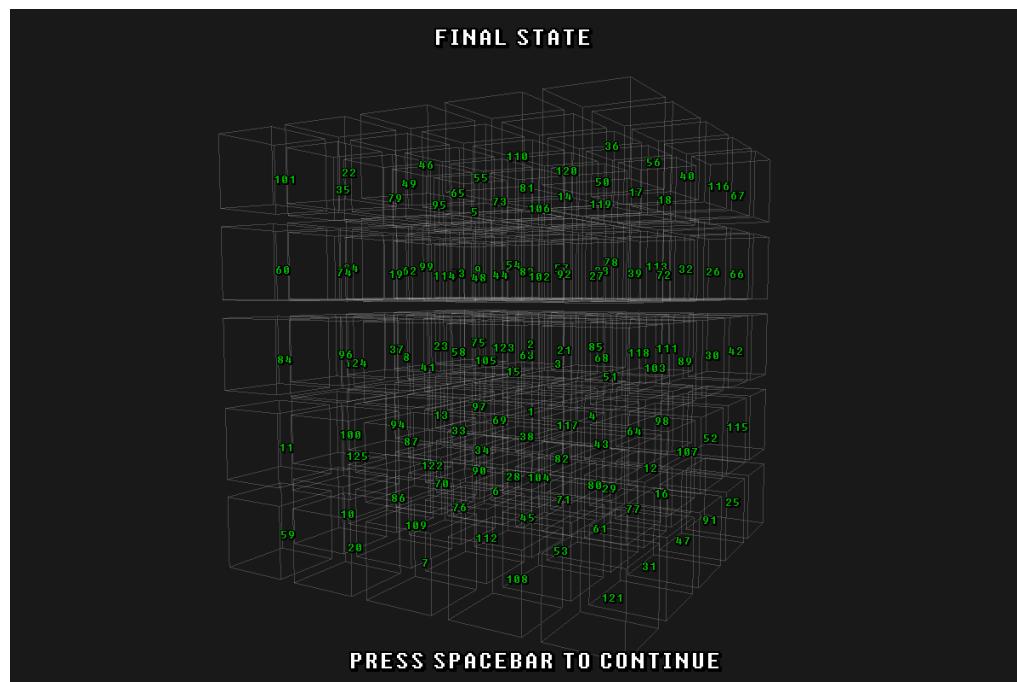
    if (distribution(gen) < mutationRate) {
        std::uniform_int_distribution<> indexDist(0,
individual.chromosome.size() - 1);
        int index1 = indexDist(gen), index2 =
indexDist(gen);
        std::swap(individual.chromosome[index1],
individual.chromosome[index2]);
    }
}
```

C. Hasil Eksperimen dan Analisis

Visualisasi kubus menggunakan OPENGL dengan GLAD library, untuk visualisasi keberhasilan sebuah objective function kami menggunakan representasi setiap angka yang akan terkena bagian dari *magic number sum* (Misalnya : angka yang ditengah kubus akan memiliki 13 *constraints* dari *row*, *column*, *pillar*, dan seterusnya). Kubus dapat dilakukan rotasi bagaikan kamera 360 derajat sehingga dapat melihat berbagai kasus *magic sum*. Pada hasil eksperimen dibawah ini, kami menampilkan dua view kubus yang berbeda untuk setiap inisialisasi dan final state dari kubus. Selain itu juga ditampilkan dengan representasi warna dan ketebalannya dengan deskripsi sebagai berikut :

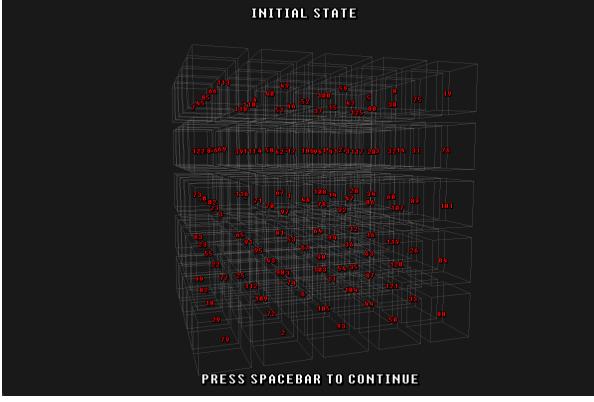
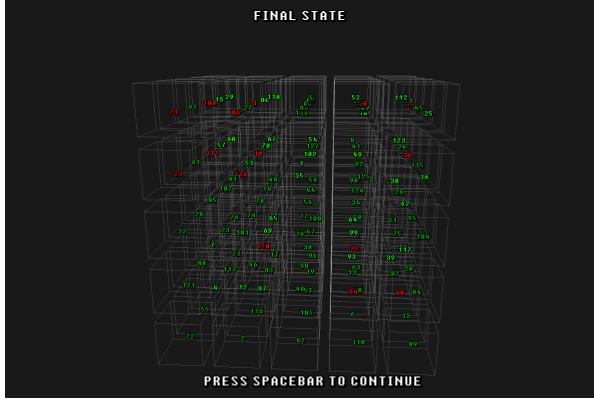
1. Merah = Tidak bagian dari magic number *constraints*
2. Hijau terang = memenuhi 1 *constraints*
3. Hijau muda = memenuhi 2 - 4 *constraints*
4. Hijau = memenuhi 5 - 7 *constraints*
5. Hijau tua = memenuhi 8 - 10 *constraints*
6. Hijau gelap = memenuhi 11 - 13 *constraints*

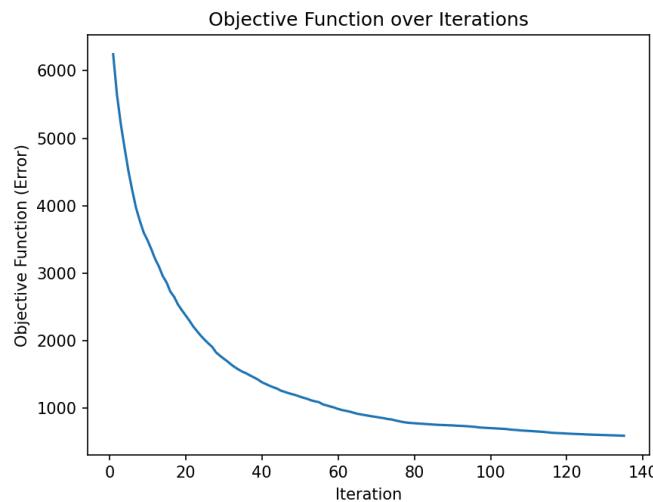
Sebagai *benchmark* visualisasi paling optimal, kami menampilkan magic cube yang ditemukan oleh Walter Trump and Christian Boyer :



Kubus didominasi oleh warna hijau tua menyesuaikan dengan constraint yang didapatkan pada setiap letak nomor kubus.

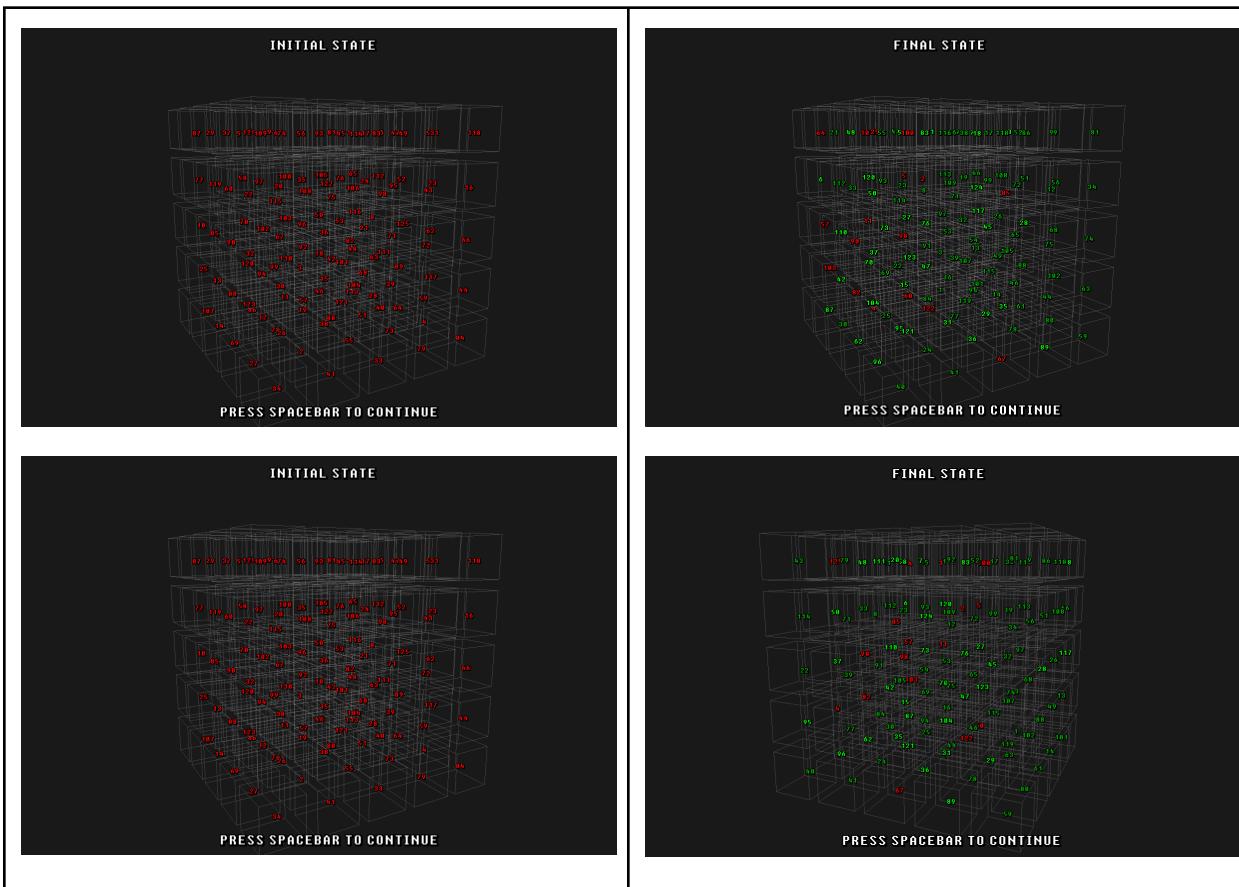
1. Steepest Ascent Hill-Climbing

Percobaan #1	
Initial State	Final State
<p>INITIAL STATE</p> 	<p>FINAL STATE</p> 
Report	
<pre>=====LOCAL SEARCH REPORT===== SEARCH ALGORITHM : STEEPEST ASCENT HILL CLIMBING FINAL ERROR : 594 TIME ESTIMATED : 9.76189 SECONDS STEPS TAKEN : 134 =====</pre>	
Plot Objective Function	



Pada percobaan pertama algoritma *Steepest Ascent Hill Climbing*, diawali dengan sebuah *initial state* yang belum memenuhi *constraint* apapun. Selanjutnya diterapkan *Steepest Ascent Hill Climbing* untuk mencari solusi dari *initial state* tersebut. Didapatkan *state akhir* dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa error, yaitu 594 . Dibutuhkan waktu pencarian selama 9,76189 detik dan 134 kali iterasi sampai memenuhi kondisi *local optimal*. Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai error sesuai dengan konsep *Steepest Ascent Hill Climbing*.

Percobaan #2	
Initial State	Final State



Report

=====LOCAL SEARCH REPORT=====

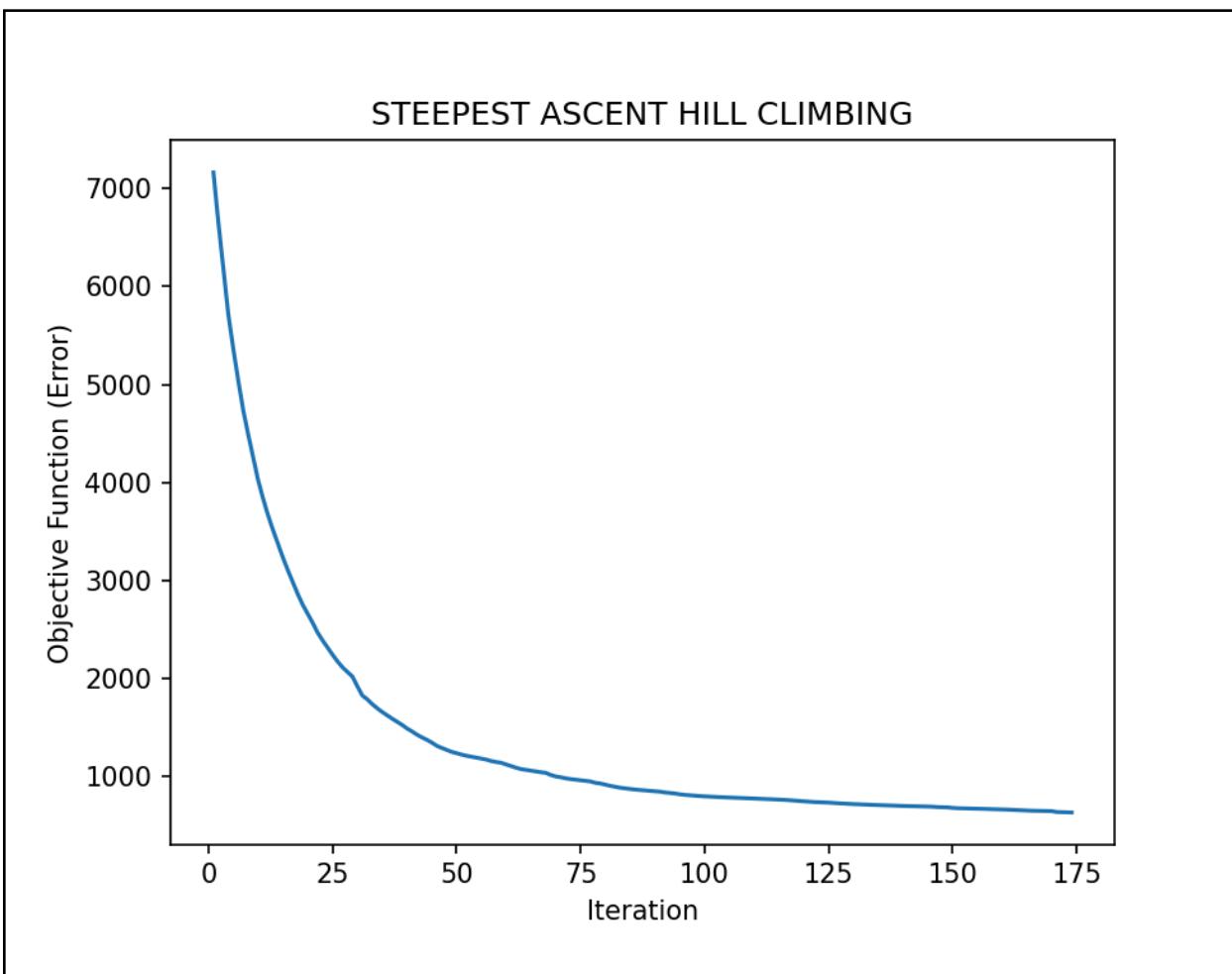
SEARCH ALGORITHM : STEEPEST ASCENT HILL CLIMBING

FINAL ERROR : 636

TIME ESTIMATED : 12.25 SECONDS

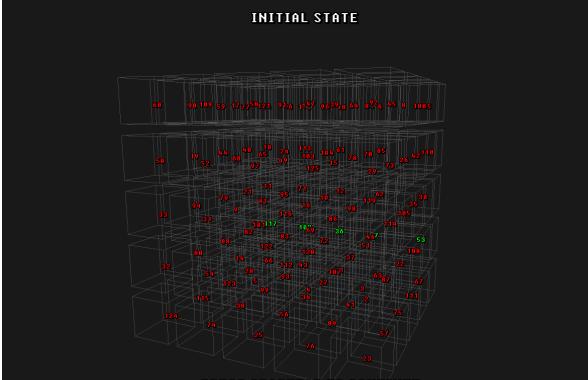
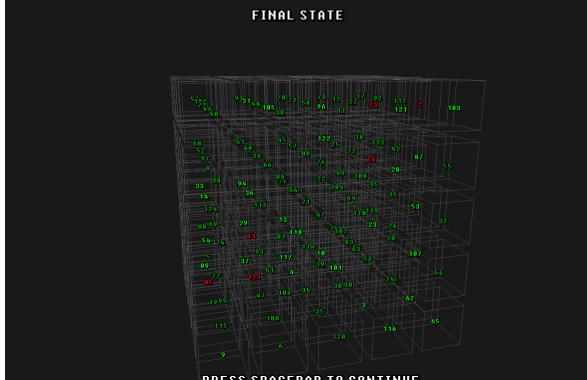
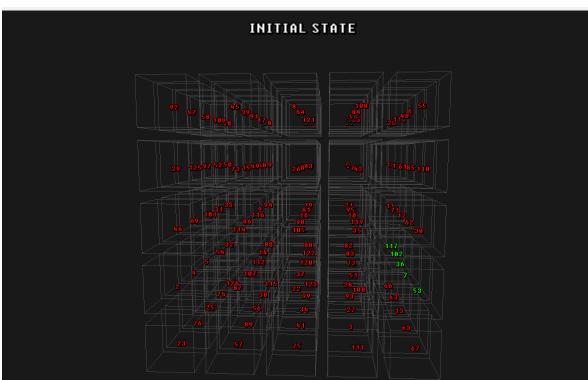
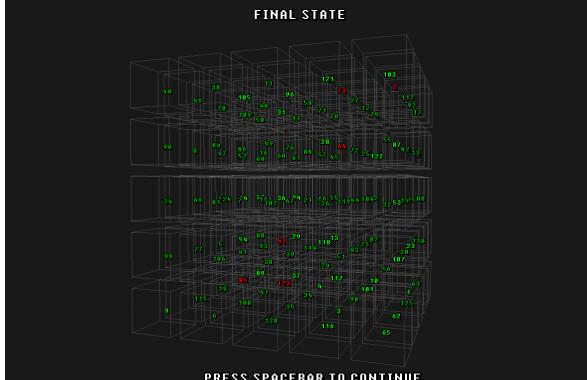
STEPS TAKEN : 173

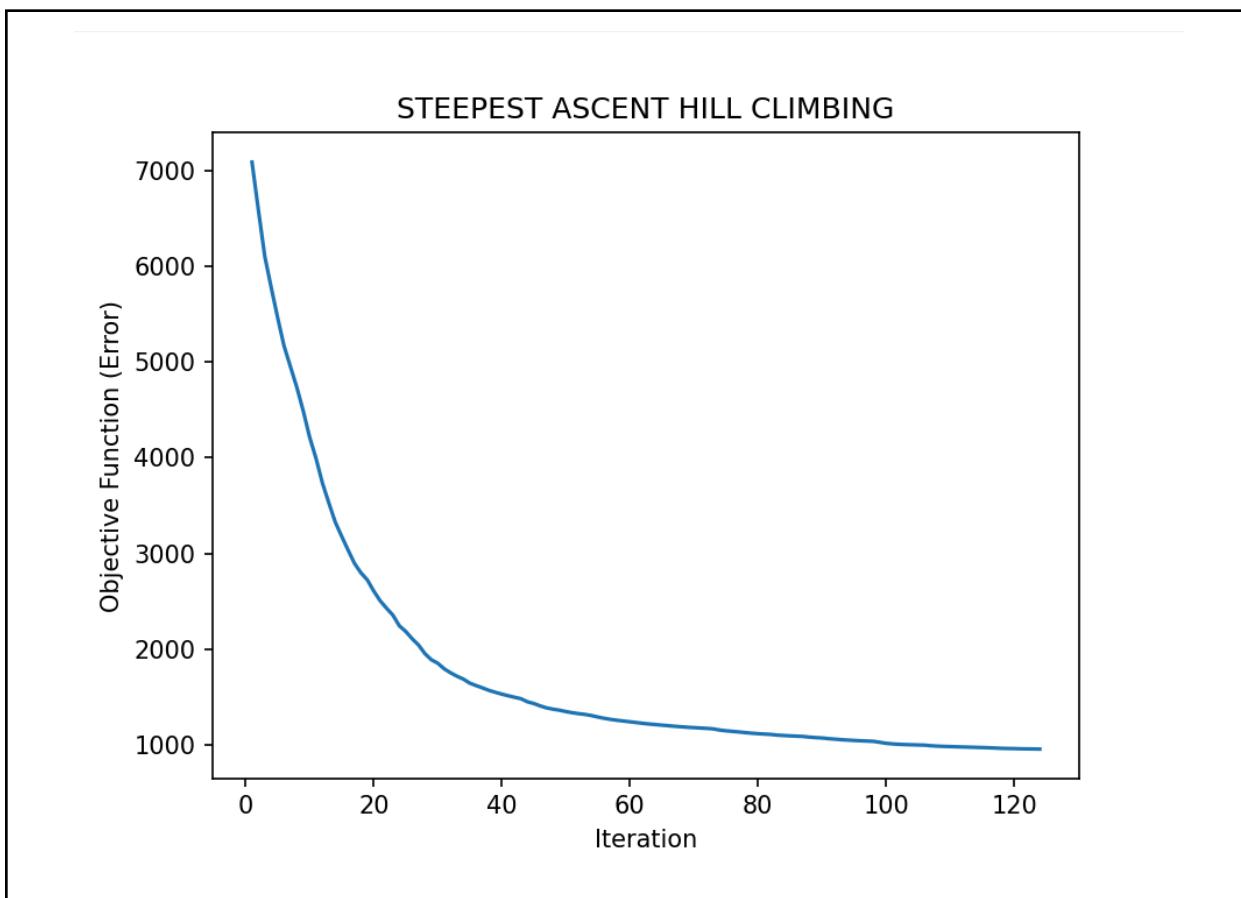
Plot Objective Function



Pada percobaan kedua algoritma *Steepest Ascent Hill Climbing*, diawali dengan sebuah *initial state* yang belum memenuhi *constraint* apapun. Selanjutnya diterapkan *Steepest Ascent Hill Climbing* untuk mencari solusi dari *initial state* tersebut. Didapatkan *state akhir* dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa error, yaitu 636. Dibutuhkan waktu pencarian selama 12,25 detik dan 173 kali iterasi sampai memenuhi kondisi *local optimal*. Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai error sesuai dengan konsep *Steepest Ascent Hill Climbing*.

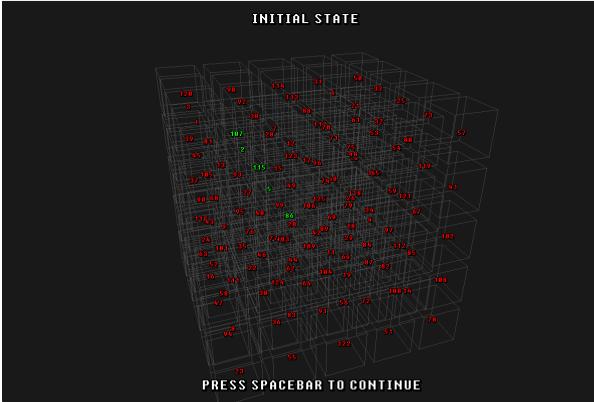
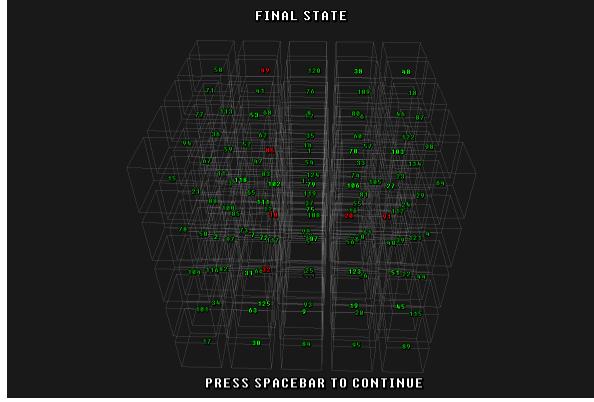
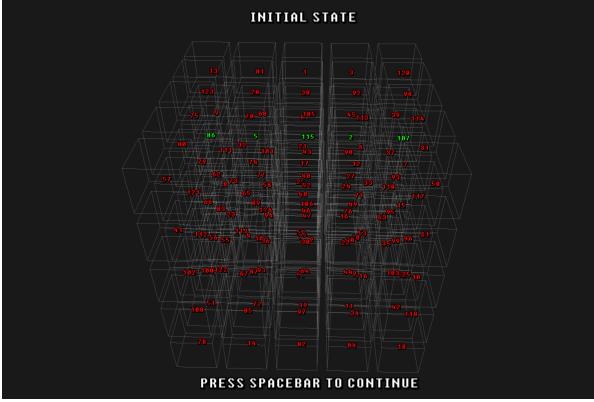
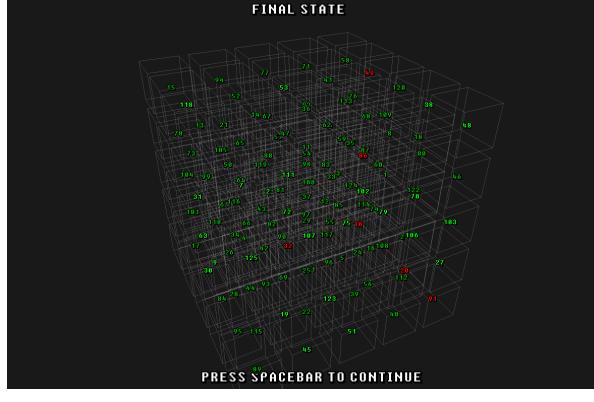
Percobaan #3

Initial State	Final State
<p>INITIAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>	<p>FINAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>
<p>INITIAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>	<p>FINAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>
Report	
<pre>=====LOCAL SEARCH REPORT=====</pre> <p>SEARCH ALGORITHM : STEEPEST ASCENT HILL CLIMBING FINAL ERROR : 956 TIME ESTIMATED : 9.94865 SECONDS STEPS TAKEN : 123</p> <pre>=====</pre>	
Plot Objective Function	



Pada percobaan ketiga algoritma *Steepest Ascent Hill Climbing*, diawali dengan sebuah *initial state* yang sudah memenuhi sebuah *constraint*. Selanjutnya diterapkan *Steepest Ascent Hill Climbing* untuk mencari solusi dari *initial state* tersebut. Didapatkan *state akhir* dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa error, yaitu 956 . Dibutuhkan waktu pencarian selama 9,94865 detik dan 123 kali iterasi sampai memenuhi *local optimal*. Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai error sesuai dengan konsep *Steepest Ascent Hill Climbing*.

2. Random Restart Hill-Climbing

Percobaan #1	Max restarts : 5
Initial State	Final State
	
	
Report	<pre>=====LOCAL SEARCH REPORT===== SEARCH ALGORITHM : RANDOM RESTART HILL CLIMBING FINAL ERROR : 469 TIME ESTIMATED : 49.6974 SECONDS STEPS TAKEN : 141</pre>

Restart Log

```
=====RESTART LOG=====

RESTART KE- 1
STEPS TAKEN    : 133
CURRENT ERROR : 776
=====

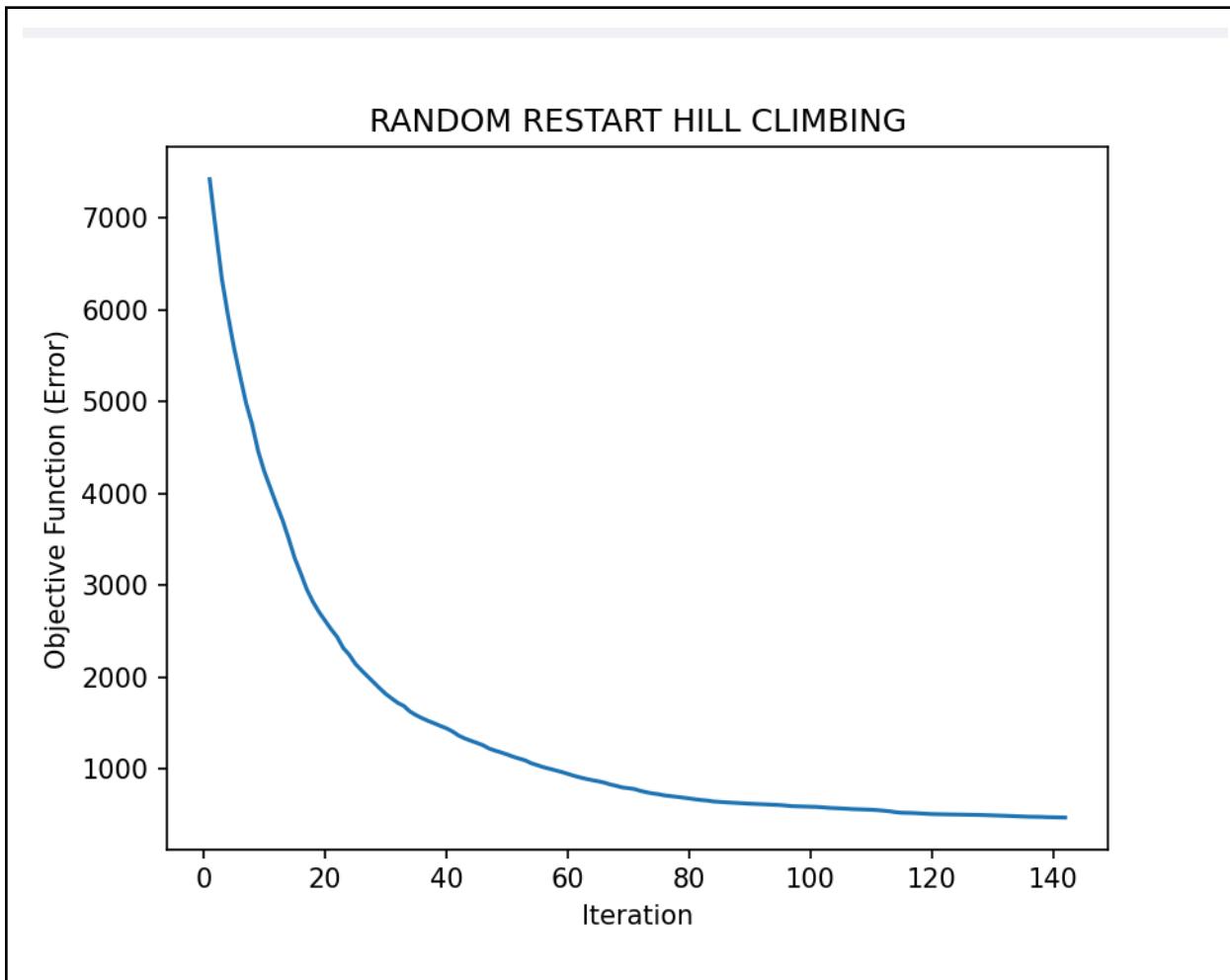
RESTART KE- 2
STEPS TAKEN    : 111
CURRENT ERROR : 699
=====

RESTART KE- 3
STEPS TAKEN    : 141
CURRENT ERROR : 469
=====

RESTART KE- 4
STEPS TAKEN    : 122
CURRENT ERROR : 508
=====

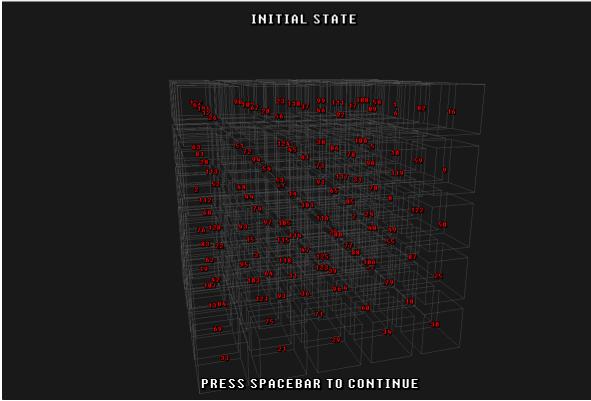
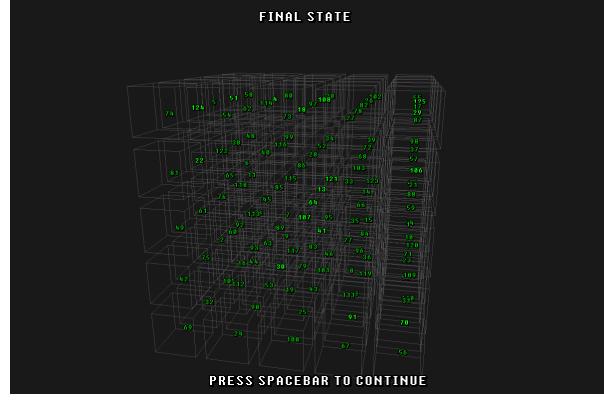
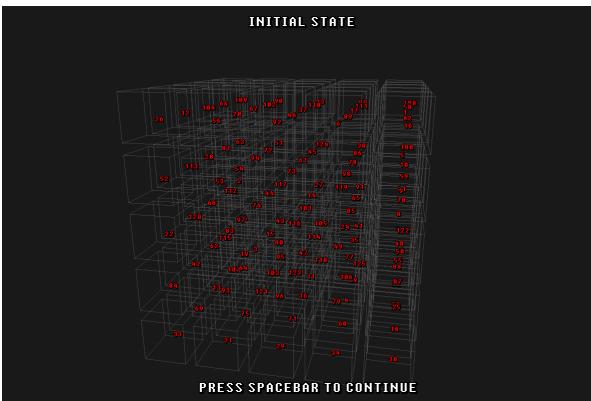
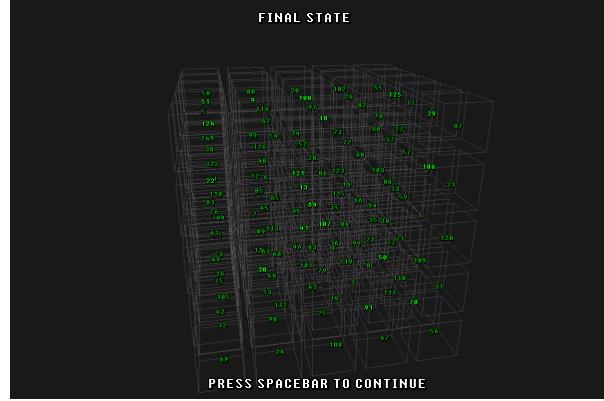
RESTART KE- 5
STEPS TAKEN    : 106
CURRENT ERROR : 892
=====
```

Plot Objective Function



Pada percobaan menggunakan algoritma *Random Restart Hill*, diawali dengan sebuah *initial state* yang sudah memenuhi sebuah *constraint*. Selanjutnya algoritma dimulai untuk mencari solusi dari *initial state* tersebut dengan *set max restart* sebanyak 5 kali. Hasil dari *Random Restart* pertama didapatkan nilai *error* sebesar 469 setelah melewati 141 steps. Setelah itu algoritma lanjut dengan melakukan *restart* hingga mendapatkan *state akhir* dari tiap *restart* seperti pada gambar di atas, dengan *restart* terakhir yang memiliki nilai *error* 892 setelah melewati 106 steps. Dibutuhkan waktu pencarian selama 49,6974 detik untuk pencarian pertama. Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai *error* sesuai dengan konsep *Hill Climbing* pada umumnya, namun jika

dilihat dari *restart log*, dapat dilihat juga nilai dari tiap *restart* dapat naik turun karena benar-benar tidak bergantung dengan satu sama lain.

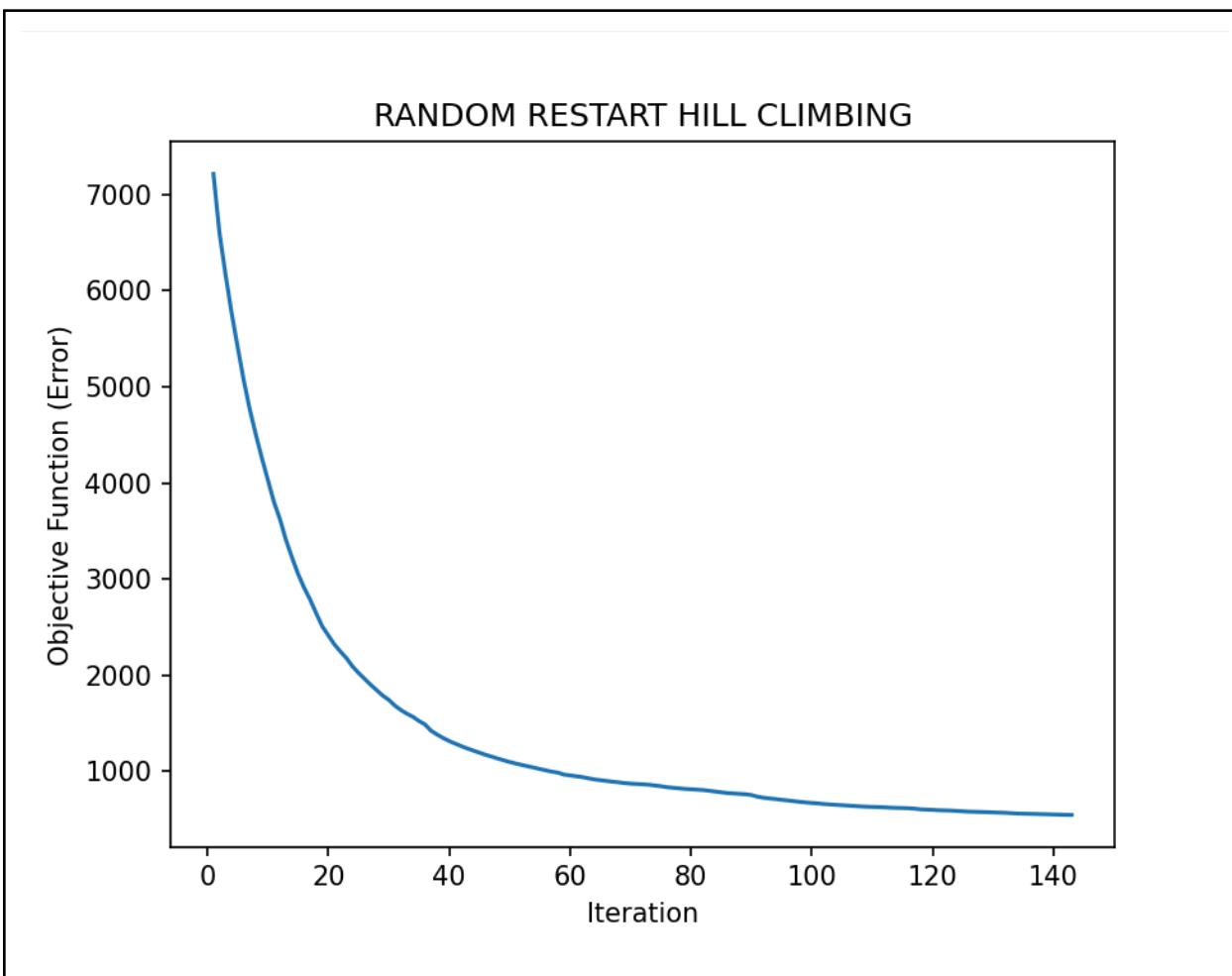
Percobaan #2	Max restarts : 10
Initial State	Final State
	
	
Report	

```
=====LOCAL SEARCH REPORT=====
SEARCH ALGORITHM : RANDOM RESTART HILL CLIMBING
FINAL ERROR      : 542
TIME ESTIMATED   : 94.0269 SECONDS
STEPS TAKEN      : 142
=====
```

Restart Log

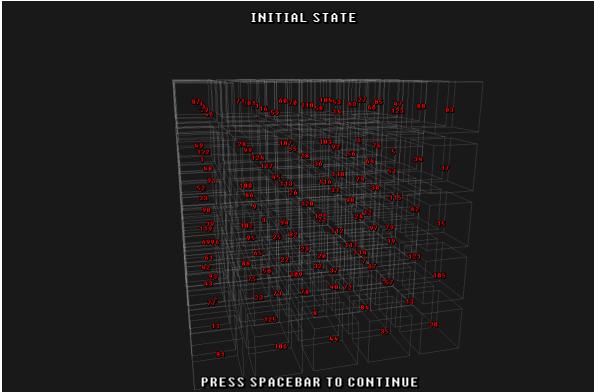
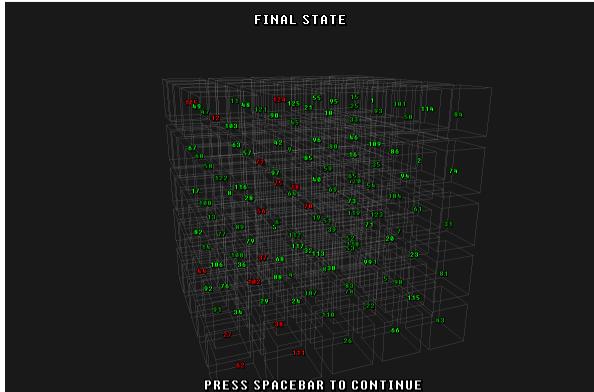
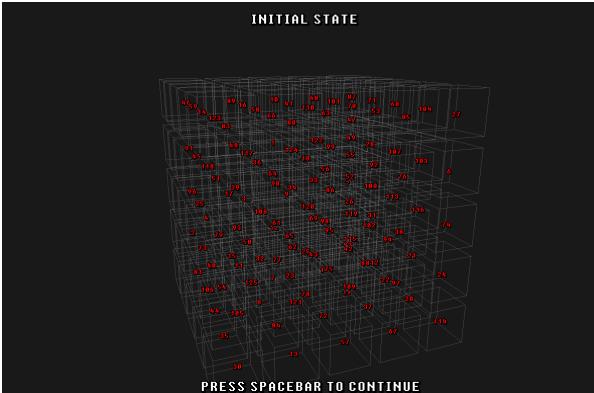
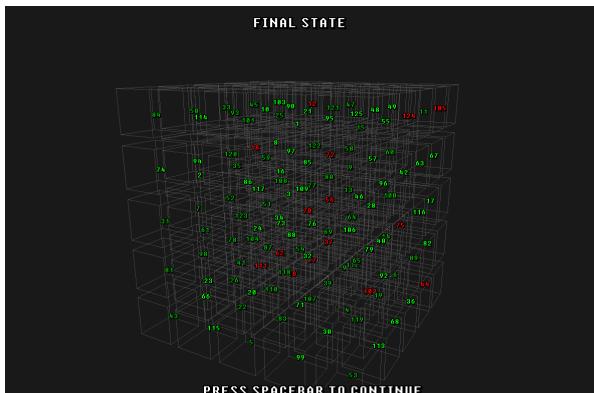
```
=====RESTART LOG=====
RESTART KE- 1
STEPS TAKEN    : 124
CURRENT ERROR : 769
=====
RESTART KE- 2
STEPS TAKEN    : 102
CURRENT ERROR : 1167
=====
RESTART KE- 3
STEPS TAKEN    : 122
CURRENT ERROR : 814
=====
RESTART KE- 4
STEPS TAKEN    : 156
CURRENT ERROR : 879
=====
RESTART KE- 5
STEPS TAKEN    : 92
CURRENT ERROR : 904
=====
RESTART KE- 6
STEPS TAKEN    : 99
CURRENT ERROR : 665
=====
RESTART KE- 7
STEPS TAKEN    : 115
CURRENT ERROR : 592
=====
RESTART KE- 8
STEPS TAKEN    : 142
CURRENT ERROR : 542
```

Plot Objective Function



Pada percobaan menggunakan algoritma *Random Restart Hill*, diawali dengan sebuah *initial state* yang sudah memenuhi sebuah *constraint*. Selanjutnya algoritma dimulai untuk mencari solusi dari *initial state* tersebut dengan *set max restart* sebanyak 10 kali. Hasil dari *Random Restart* pertama didapatkan nilai *error* sebesar 542 setelah melewati 142 *steps*. Setelah itu algoritma lanjut dengan melakukan *restart* hingga mendapatkan *state akhir* dari tiap *restart* seperti pada gambar di atas, dengan *restart* terakhir yang memiliki nilai *error* 542 setelah melewati 142 *steps*. Dibutuhkan waktu pencarian selama 94,0269 detik untuk pencarian pertama. Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai *error* sesuai dengan konsep *Hill Climbing* pada umumnya, namun jika

dilihat dari *restart log*, dapat dilihat juga nilai dari tiap *restart* dapat naik turun karena benar-benar tidak bergantung dengan satu sama lain.

Percobaan #3	Max restarts : 20
Initial State	Final State
	
	
Report	<pre>=====LOCAL SEARCH REPORT===== SEARCH ALGORITHM : RANDOM RESTART HILL CLIMBING FINAL ERROR : 428 TIME ESTIMATED : 181.488 SECONDS STEPS TAKEN : 121 =====</pre>

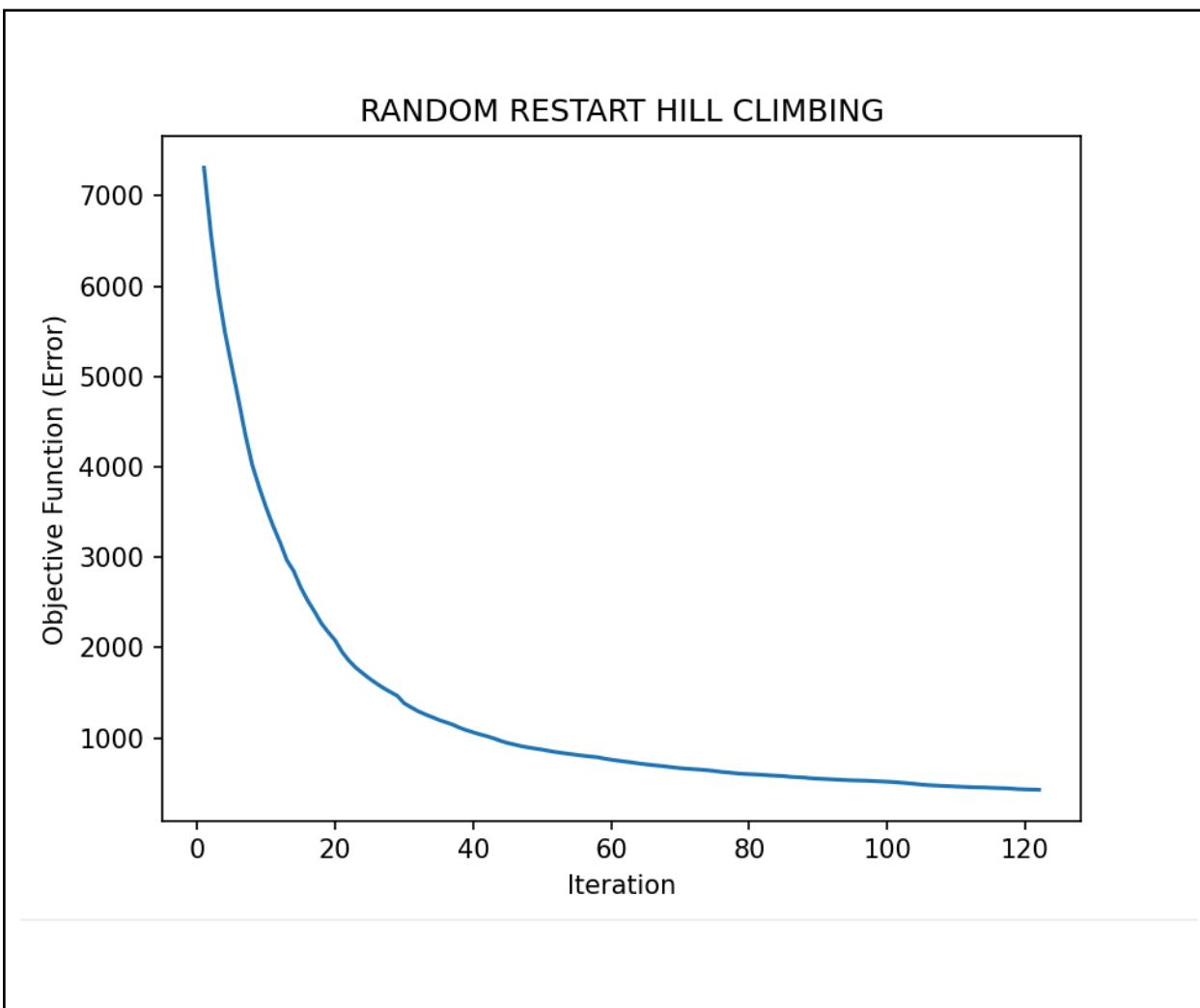
Restart Log

```
=====RESTART LOG=====
RESTART KE- 1
STEPS TAKEN : 148
CURRENT ERROR : 634
=====
RESTART KE- 2
STEPS TAKEN : 106
CURRENT ERROR : 1078
=====
RESTART KE- 3
STEPS TAKEN : 121
CURRENT ERROR : 428
=====
RESTART KE- 4
STEPS TAKEN : 158
CURRENT ERROR : 646
=====
RESTART KE- 5
STEPS TAKEN : 112
CURRENT ERROR : 587
=====
RESTART KE- 6
STEPS TAKEN : 196
CURRENT ERROR : 518
=====
RESTART KE- 7
STEPS TAKEN : 110
CURRENT ERROR : 953
=====
RESTART KE- 8
STEPS TAKEN : 143
CURRENT ERROR : 545
```

```
=====
RESTART KE- 9
STEPS TAKEN : 136
CURRENT ERROR : 678
=====
RESTART KE- 10
STEPS TAKEN : 129
CURRENT ERROR : 633
=====
RESTART KE- 11
STEPS TAKEN : 123
CURRENT ERROR : 486
=====
RESTART KE- 12
STEPS TAKEN : 114
CURRENT ERROR : 621
=====
RESTART KE- 13
STEPS TAKEN : 114
CURRENT ERROR : 1846
=====
RESTART KE- 14
STEPS TAKEN : 127
CURRENT ERROR : 649
=====
RESTART KE- 15
STEPS TAKEN : 123
CURRENT ERROR : 881
=====
RESTART KE- 16
STEPS TAKEN : 183
CURRENT ERROR : 621
```

```
=====
RESTART KE- 17
STEPS TAKEN : 143
CURRENT ERROR : 856
=====
RESTART KE- 18
STEPS TAKEN : 126
CURRENT ERROR : 859
=====
RESTART KE- 19
STEPS TAKEN : 99
CURRENT ERROR : 822
=====
RESTART KE- 20
STEPS TAKEN : 123
CURRENT ERROR : 1001
```

Plot Objective Function

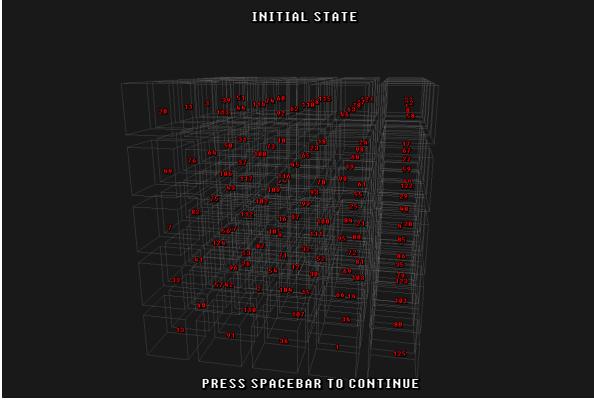
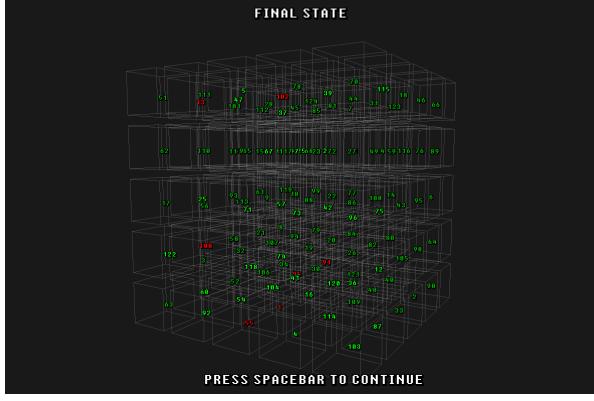
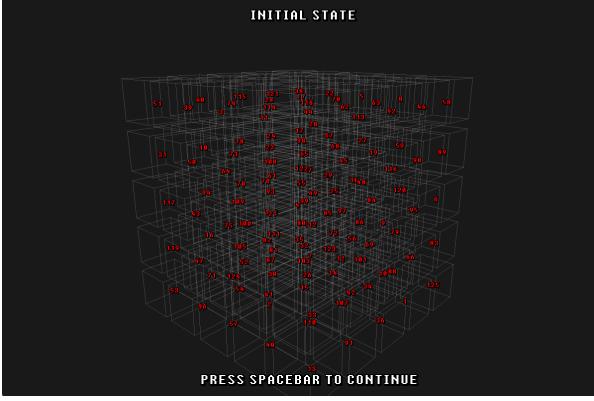
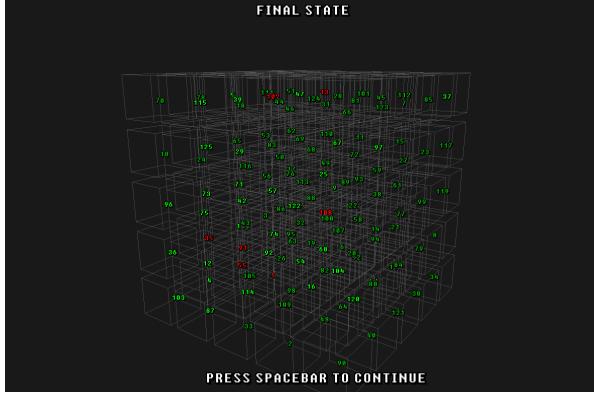


Pada percobaan menggunakan algoritma *Random Restart Hill*, diawali dengan sebuah *initial state* yang sudah memenuhi sebuah *constraint*. Selanjutnya algoritma dimulai untuk mencari solusi dari *initial state* tersebut dengan *set max restart* sebanyak 15 kali. Hasil dari *Random Restart* pertama didapatkan nilai *error* sebesar 428 setelah melewati 121 *steps*. Setelah itu algoritma lanjut dengan melakukan *restart* hingga mendapatkan *state akhir* dari tiap *restart* seperti pada gambar di atas, dengan *restart* terakhir yang memiliki nilai *error* 1001 setelah melewati 123 *steps*. Dibutuhkan waktu pencarian selama 141,488 detik untuk pencarian pertama. Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai *error* sesuai dengan konsep *Hill Climbing* pada umumnya, namun jika

dilihat dari *restart log*, dapat dilihat juga nilai dari tiap *restart* dapat naik turun karena benar-benar tidak bergantung dengan satu sama lain.

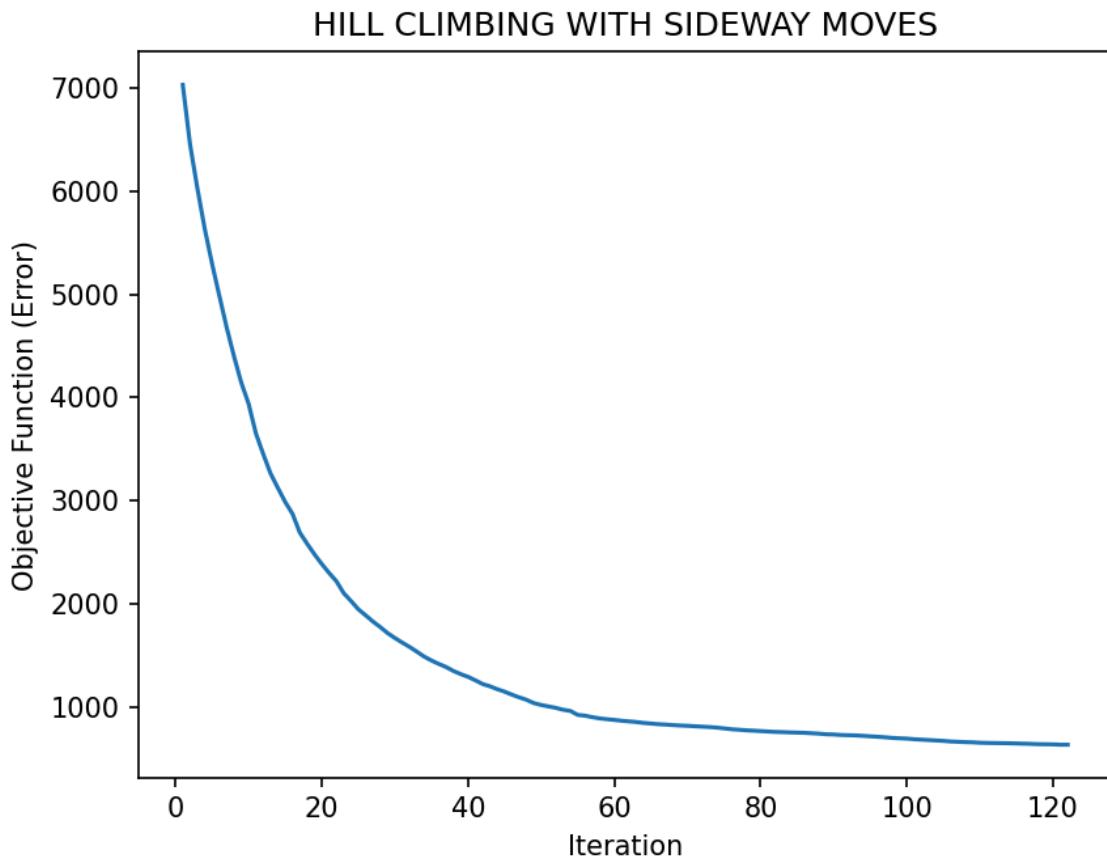
Kesimpulan dari hasil percobaan menggunakan algoritma *Random Restart Hill* adalah sebenarnya berapapun banyak *restart* yang ditentukan, tidak mempengaruhi hasil secara langsung. Dikarenakan yang diambil dari algoritma ini adalah hasil terakhir, bukan hasil terbaik sehingga jumlah *restart* tidak mempengaruhi hasil yang didapatkan semakin bagus atau tidak.

3. Hill-Climbing with Sideways Move

Percobaan #1	Max sideways move : 5
Initial State	Final State
	
	
Report	

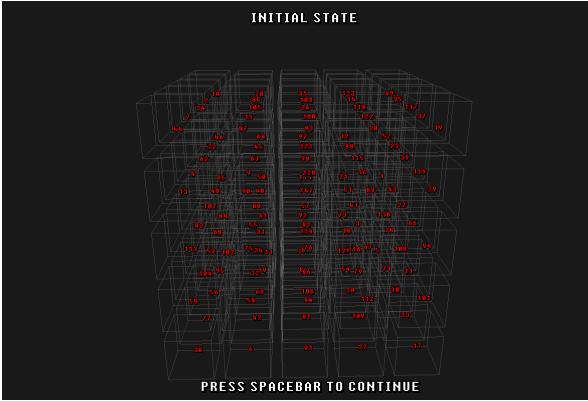
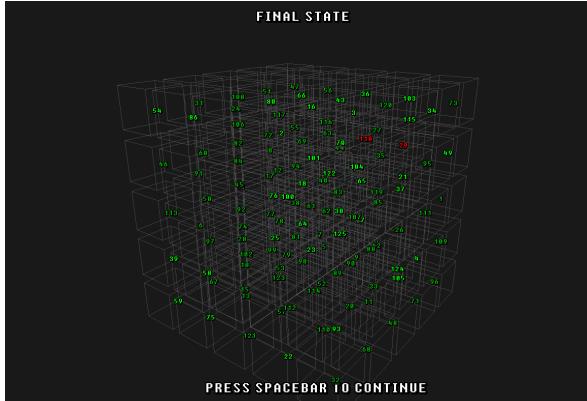
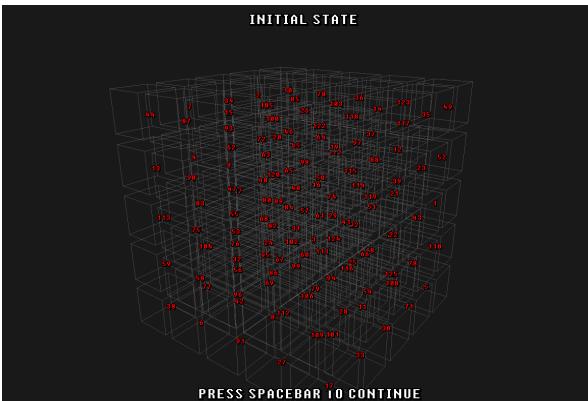
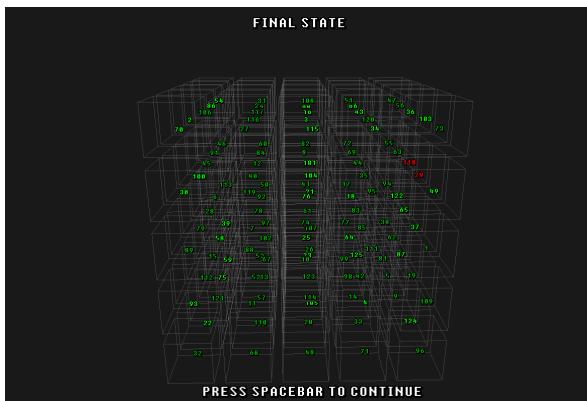
```
=====LOCAL SEARCH REPORT=====
SEARCH ALGORITHM : HILL CLIMBING WITH SIDEWAY MOVES
FINAL ERROR      : 634
TIME ESTIMATED   : 10.0326 SECONDS
STEPS TAKEN      : 121
=====
```

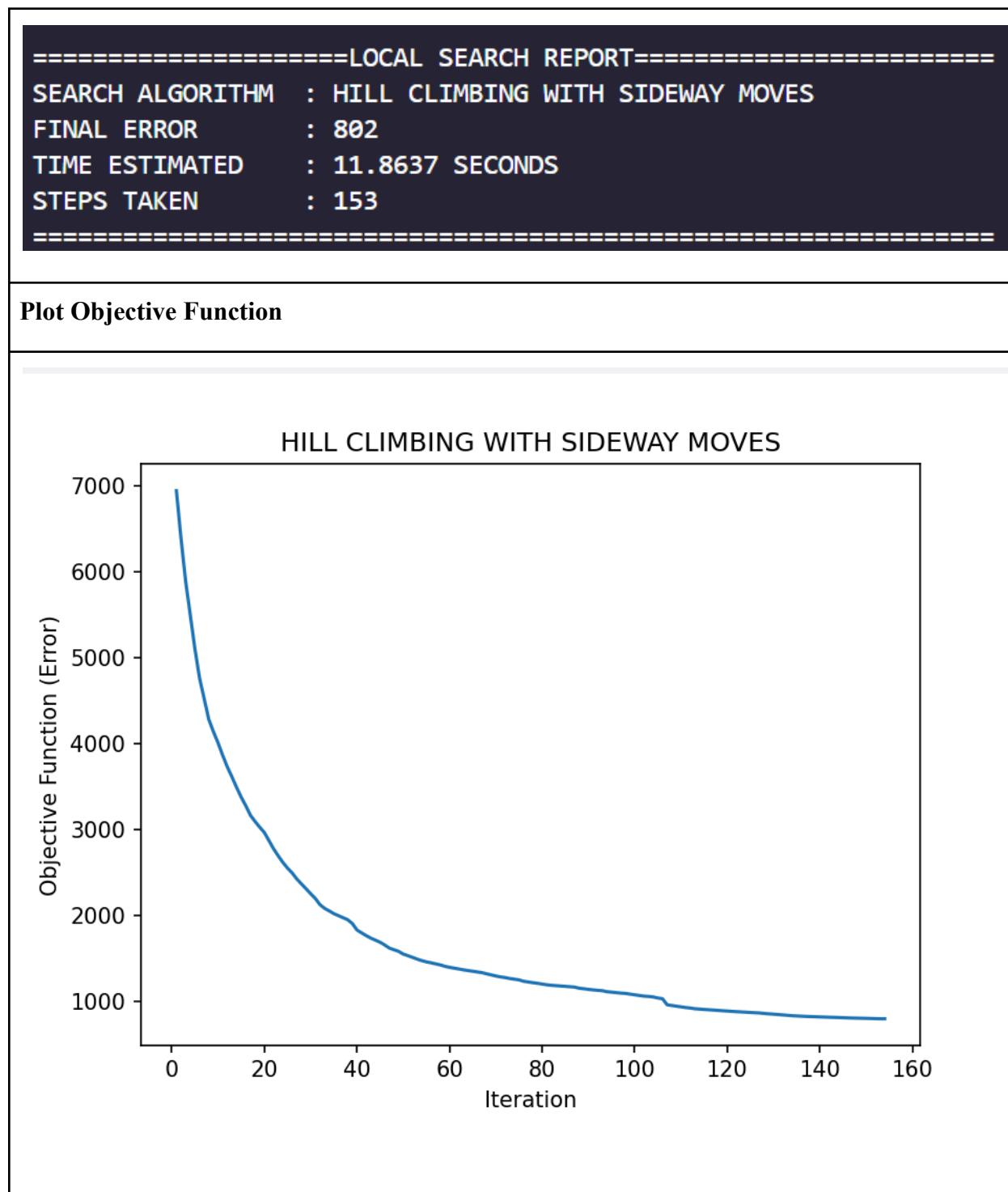
Plot Objective Function



Pada percobaan pertama algoritma *Hill Climbing with Sideways Move*, diawali dengan sebuah *initial state* yang belum memenuhi *constraint* apapun. Selanjutnya diterapkan *Hill Climbing with Sideways Move* untuk mencari solusi dari *initial state* tersebut dengan maksimal

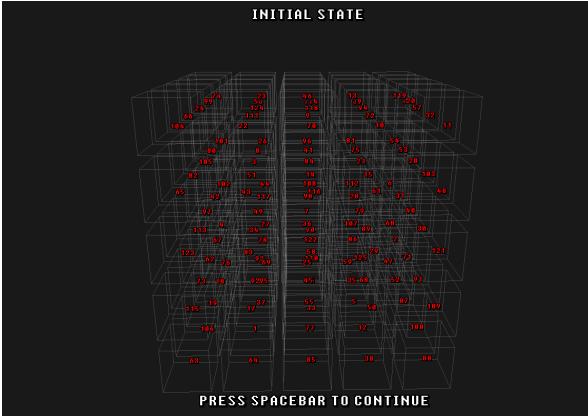
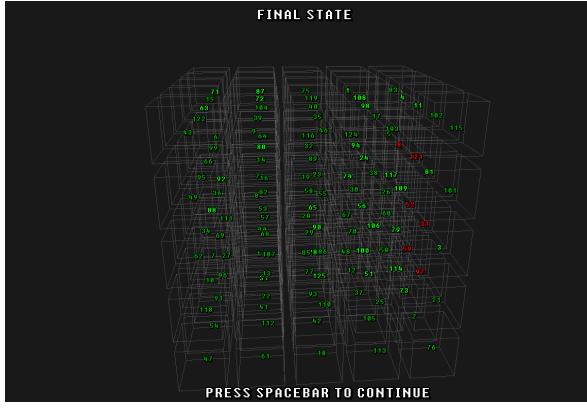
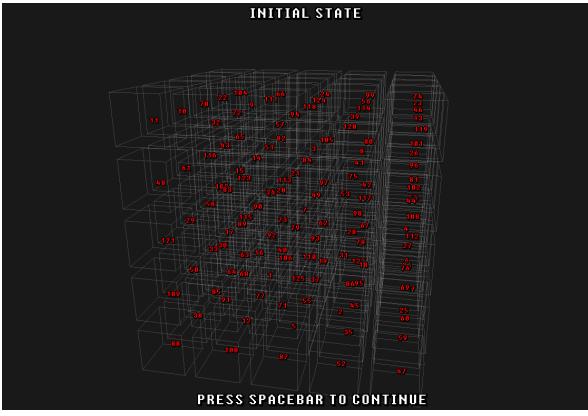
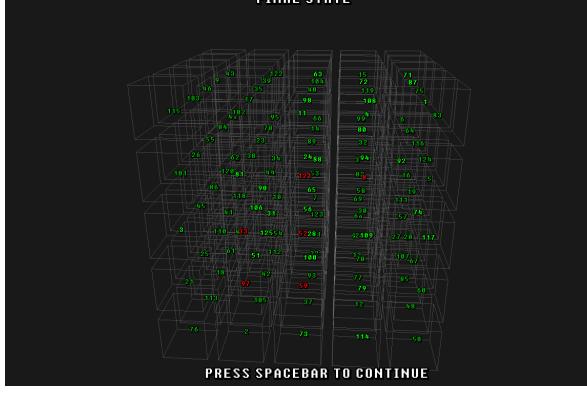
sideways move yang dapat dilakukan pada percobaan pertama adalah sebanyak 5 kali. Didapatkan *state akhir* dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa error, yaitu 634 .Dibutuhkan waktu pencarian selama 10,0326 detik dan 121 kali iterasi sampai memenuhi kondisi *final state*. Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai error sesuai dengan konsep *hill climbing with sideways move*.

Percobaan #2	Max sideways move : 10
Initial State	Final State
	
	
Report	



Pada percobaan kedua algoritma *Hill Climbing with Sideways Move*, diawali dengan sebuah *initial state* yang belum memenuhi *constraint* apapun. Selanjutnya diterapkan *Hill*

Climbing with Sideways Move untuk mencari solusi dari *initial state* tersebut dengan maksimal *sideways move* yang dapat dilakukan pada percobaan pertama adalah sebanyak 10 kali. Didapatkan *state akhir* dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa error, yaitu 802. Dibutuhkan waktu pencarian selama 11,8637 detik dan 153 kali iterasi sampai memenuhi kondisi *final state*. Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai error sesuai dengan konsep *hill climbing with sideways move*.

Percobaan #3	Max sideways move : 5
Initial State	Final State
	
	

Report

```
=====LOCAL SEARCH REPORT=====
```

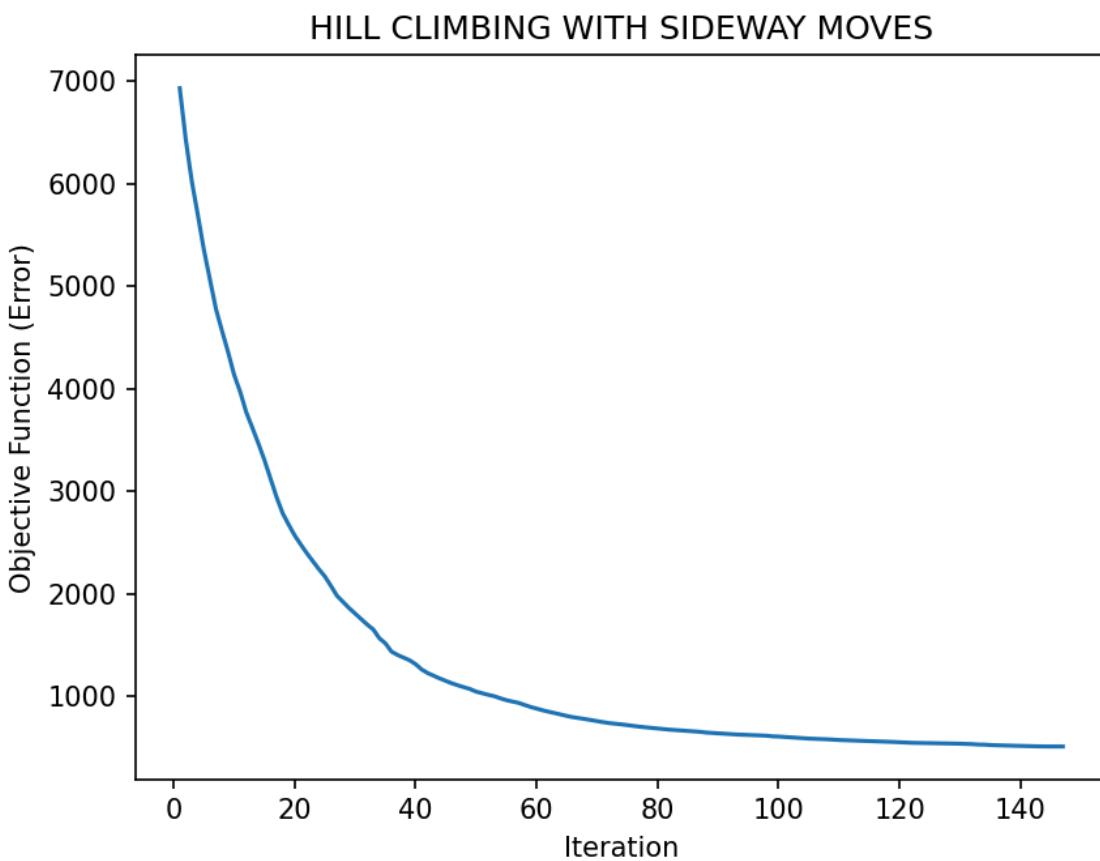
```
SEARCH ALGORITHM : HILL CLIMBING WITH SIDEWAY MOVES
```

```
FINAL ERROR : 511
```

```
TIME ESTIMATED : 12.1045 SECONDS
```

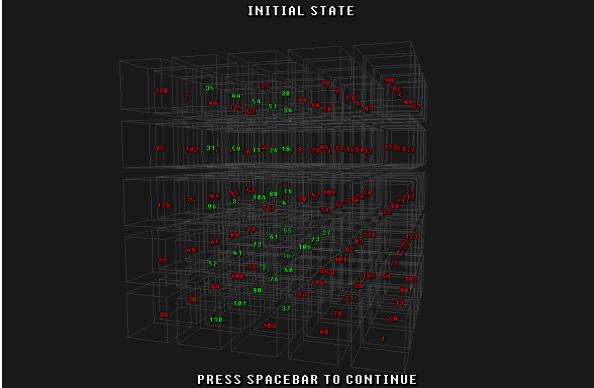
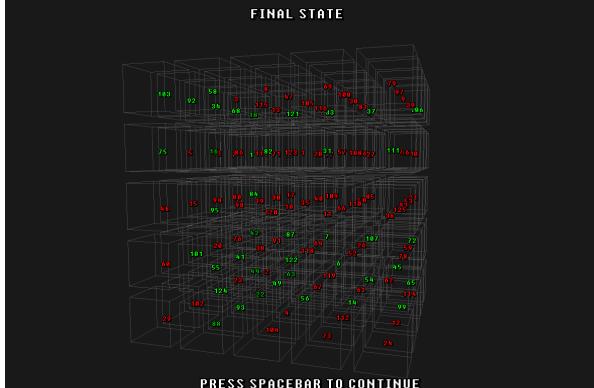
```
STEPS TAKEN : 146
```

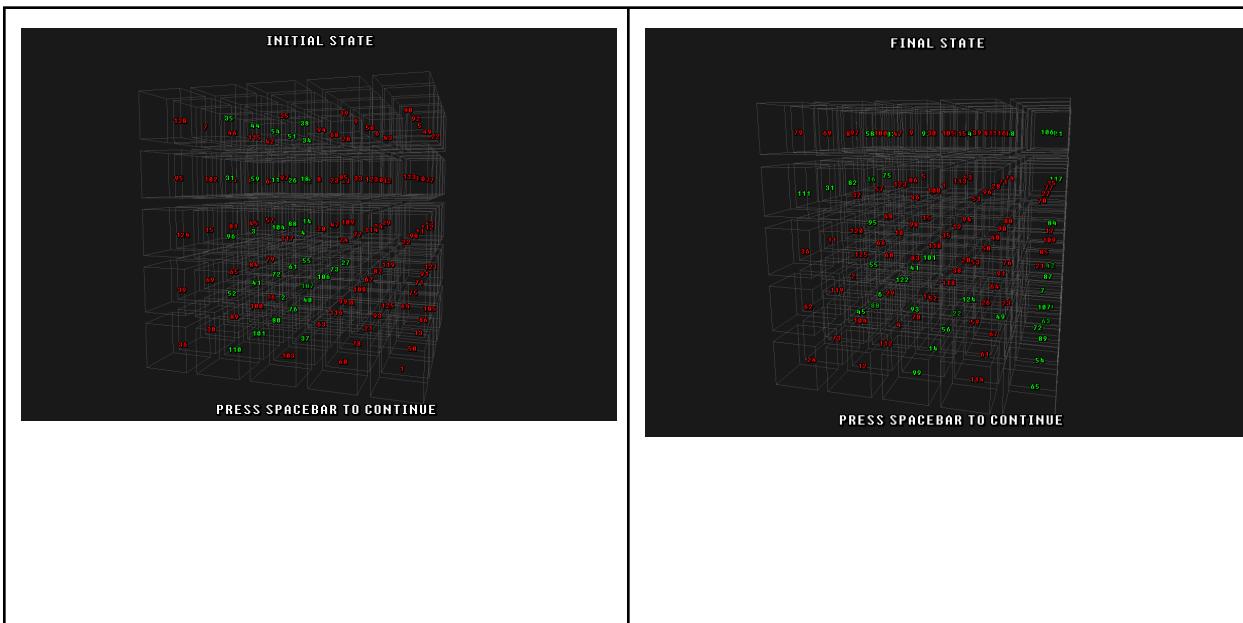
Plot Objective Function



Pada percobaan ketiga algoritma *Hill Climbing with Sideways Move*, diawali dengan sebuah *initial state* yang belum memenuhi *constraint* apapun. Selanjutnya diterapkan *Hill Climbing with Sideways Move* untuk mencari solusi dari *initial state* tersebut dengan maksimal *sideways move* yang dapat dilakukan pada percobaan pertama adalah sebanyak 15 kali. Didapatkan *state akhir* dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa error, yaitu 511. Dibutuhkan waktu pencarian selama 12,1045 detik dan 146 kali iterasi sampai memenuhi kondisi *final state*. Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai error sesuai dengan konsep *hill climbing with sideways move*.

4. Stochastic Hill Climbing

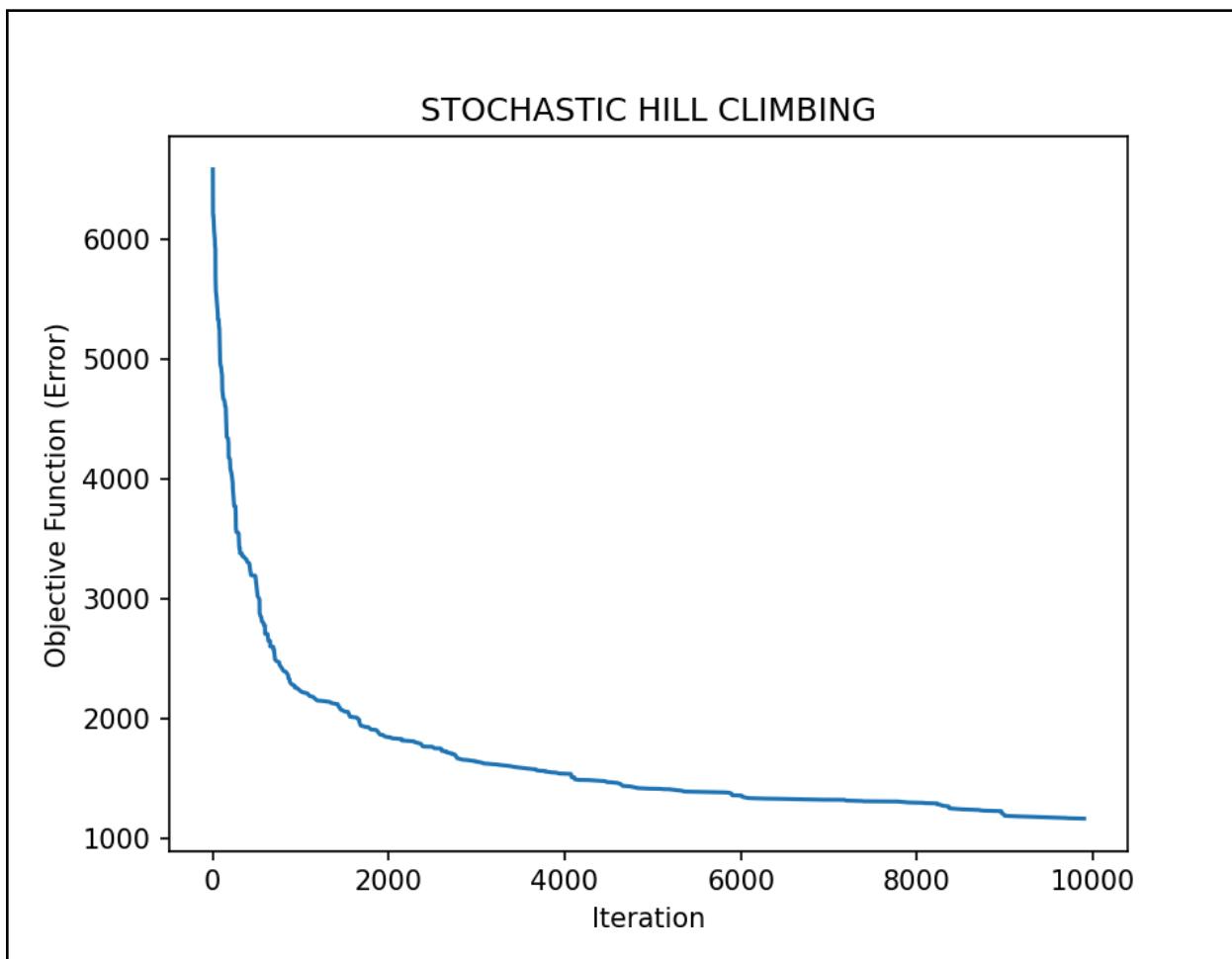
Percobaan #1	Max iteration : 10000
Initial State	Final State
	



Report

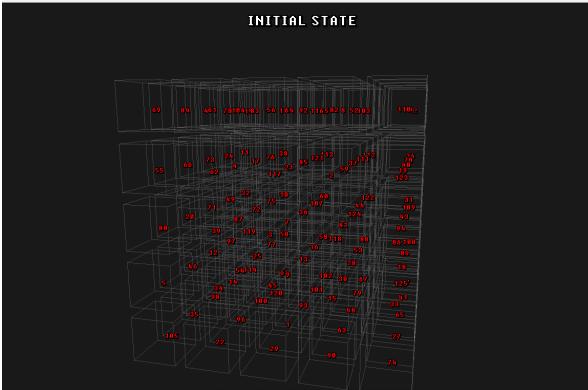
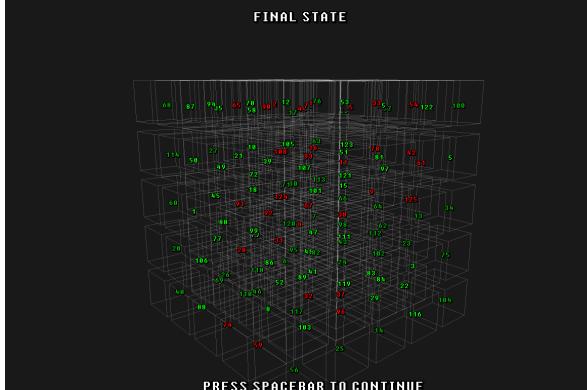
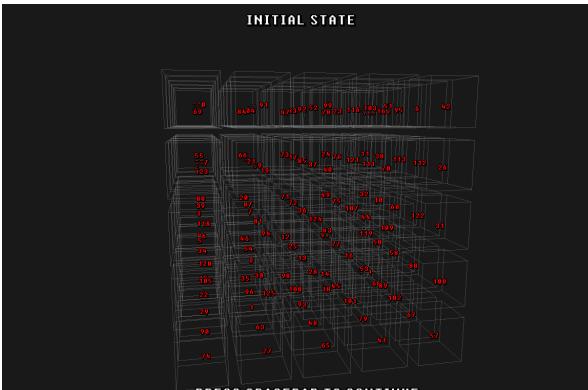
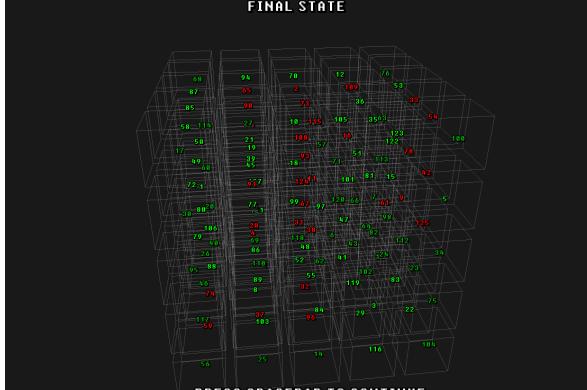
```
=====LOCAL SEARCH REPORT=====
SEARCH ALGORITHM : STOCHASTIC HILL CLIMBING
FINAL ERROR      : 1166
TIME ESTIMATED   : 0.696087 SECONDS
STEPS TAKEN      : 10000
=====
```

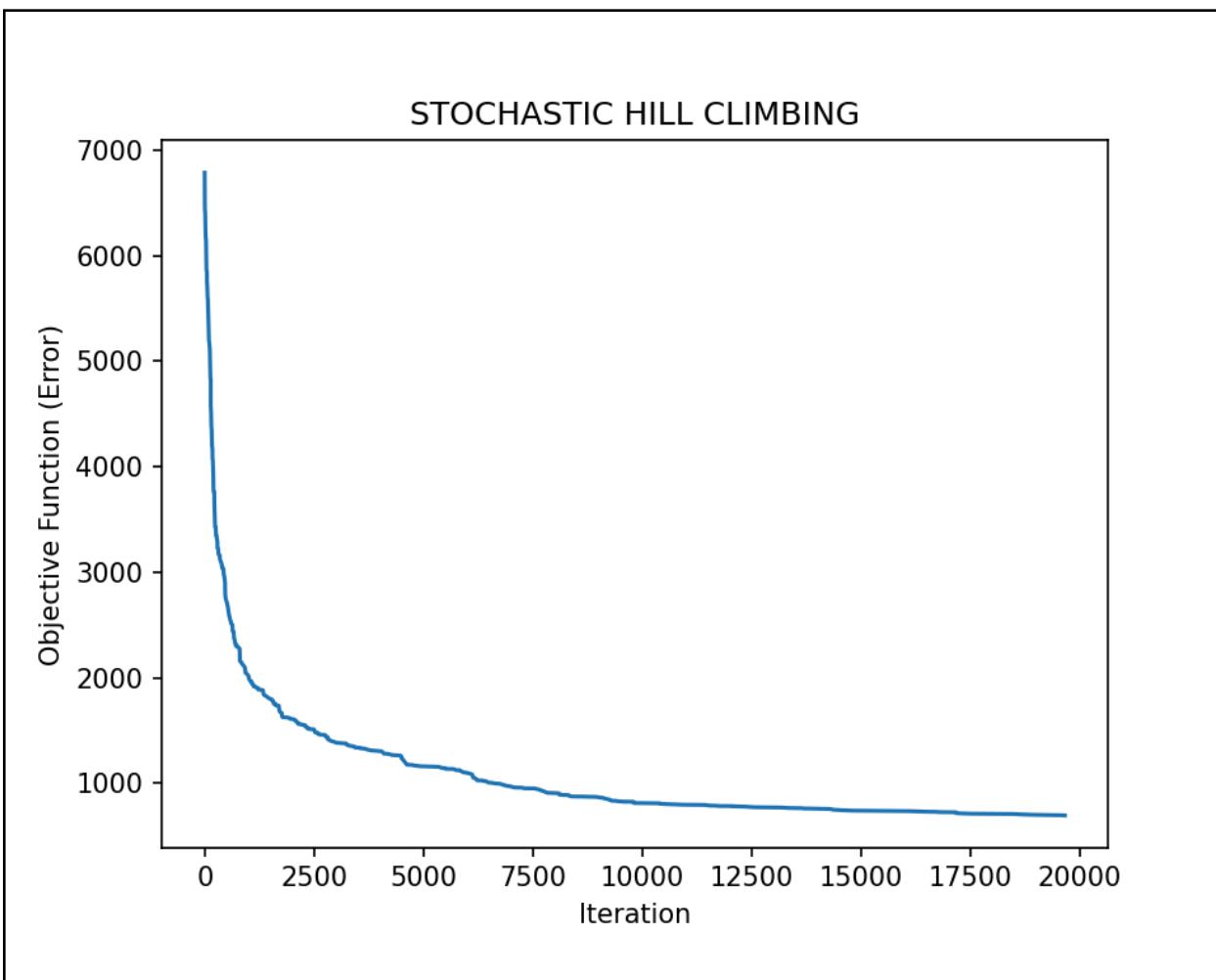
Plot Objective Function



Pada percobaan pertama algoritma *Stochastic Hill Climbing*, diawali dengan sebuah *initial state* yang sudah memenuhi beberapa *constraint*. Selanjutnya diterapkan *Stochastic Hill Climbing* untuk mencari solusi dari *initial state* tersebut dengan *set max* iterasi sebanyak 10000. Didapatkan *state* akhir dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa *error*, yaitu 1166 . Dibutuhkan waktu pencarian selama 0,69687 detik sampai memenuhi *error* 1166 . Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai error sesuai dengan konsep *Stochastic Hill Climbing*.

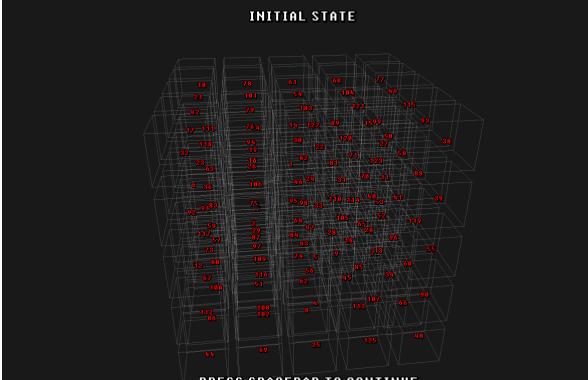
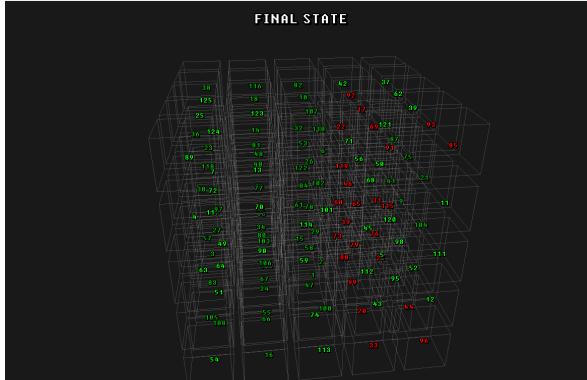
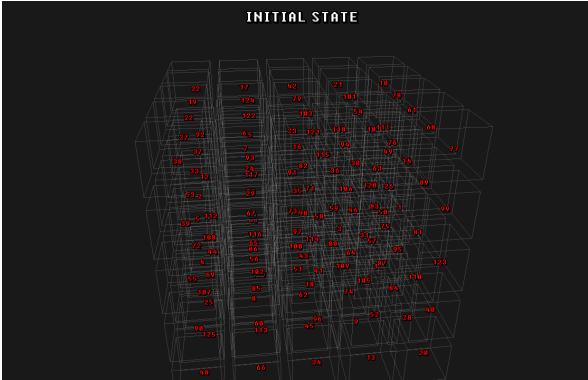
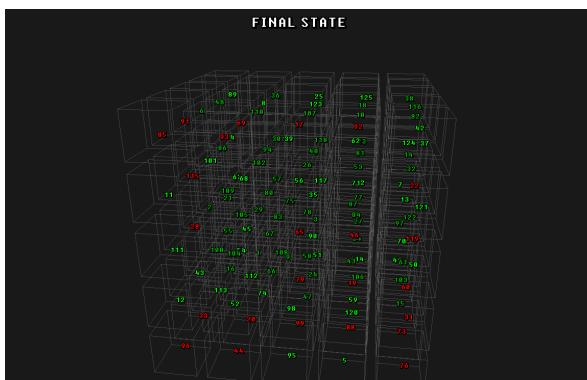
Percobaan #2	Max iteration : 20000
---------------------	------------------------------

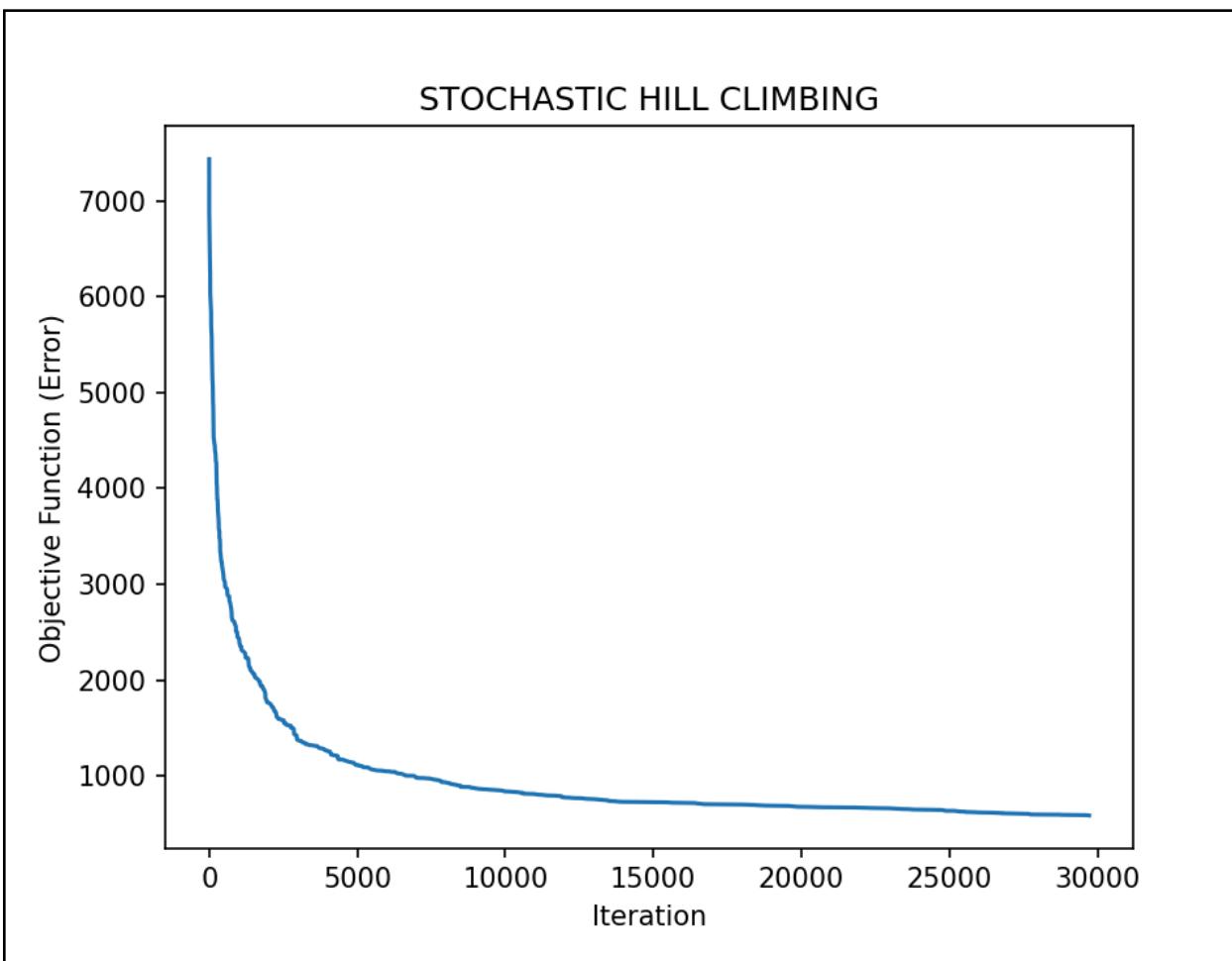
Initial State	Final State
<p>INITIAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>	<p>FINAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>
<p>INITIAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>	<p>FINAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>
Report	
<pre>=====LOCAL SEARCH REPORT===== SEARCH ALGORITHM : STOCHASTIC HILL CLIMBING FINAL ERROR : 694 TIME ESTIMATED : 1.10844 SECONDS STEPS TAKEN : 20000</pre>	
Plot Objective Function	



Pada percobaan pertama algoritma *Stochastic Hill Climbing*, diawali dengan sebuah *initial state* yang sudah memenuhi beberapa *constraint*. Selanjutnya diterapkan *Stochastic Hill Climbing* untuk mencari solusi dari *initial state* tersebut dengan *set max* iterasi sebanyak 30000. Didapatkan *state akhir* dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa *error*, yaitu 694 . Dibutuhkan waktu pencarian selama 1,10844 detik sampai memenuhi *error* 694 . Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai error sesuai dengan konsep *Stochastic Hill Climbing*.

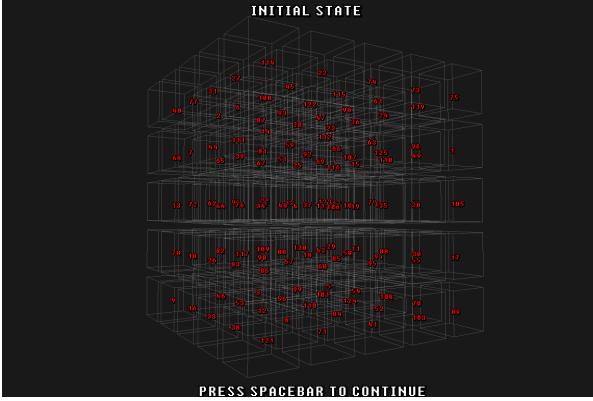
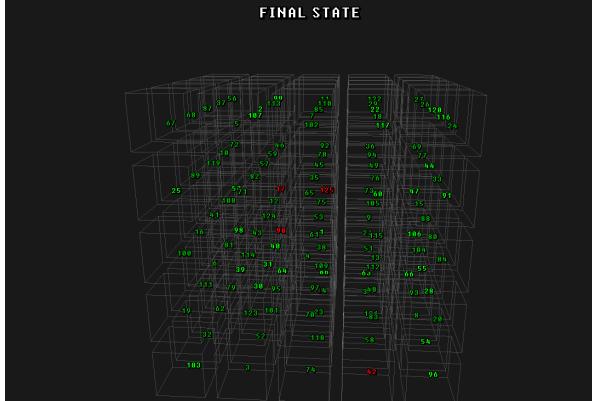
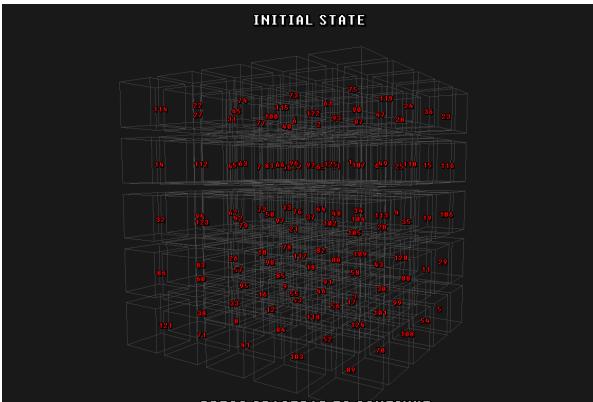
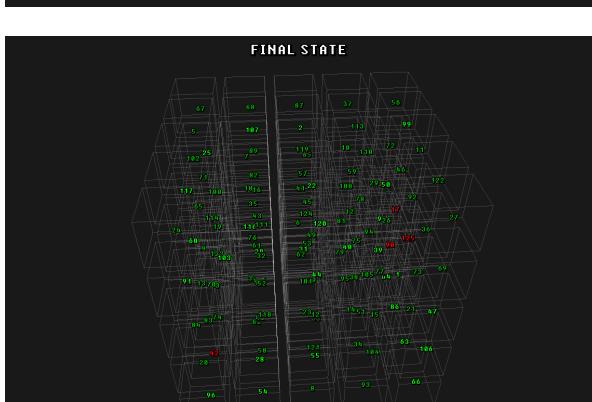
Percobaan #3	Max iteration : 30000
--------------	-----------------------

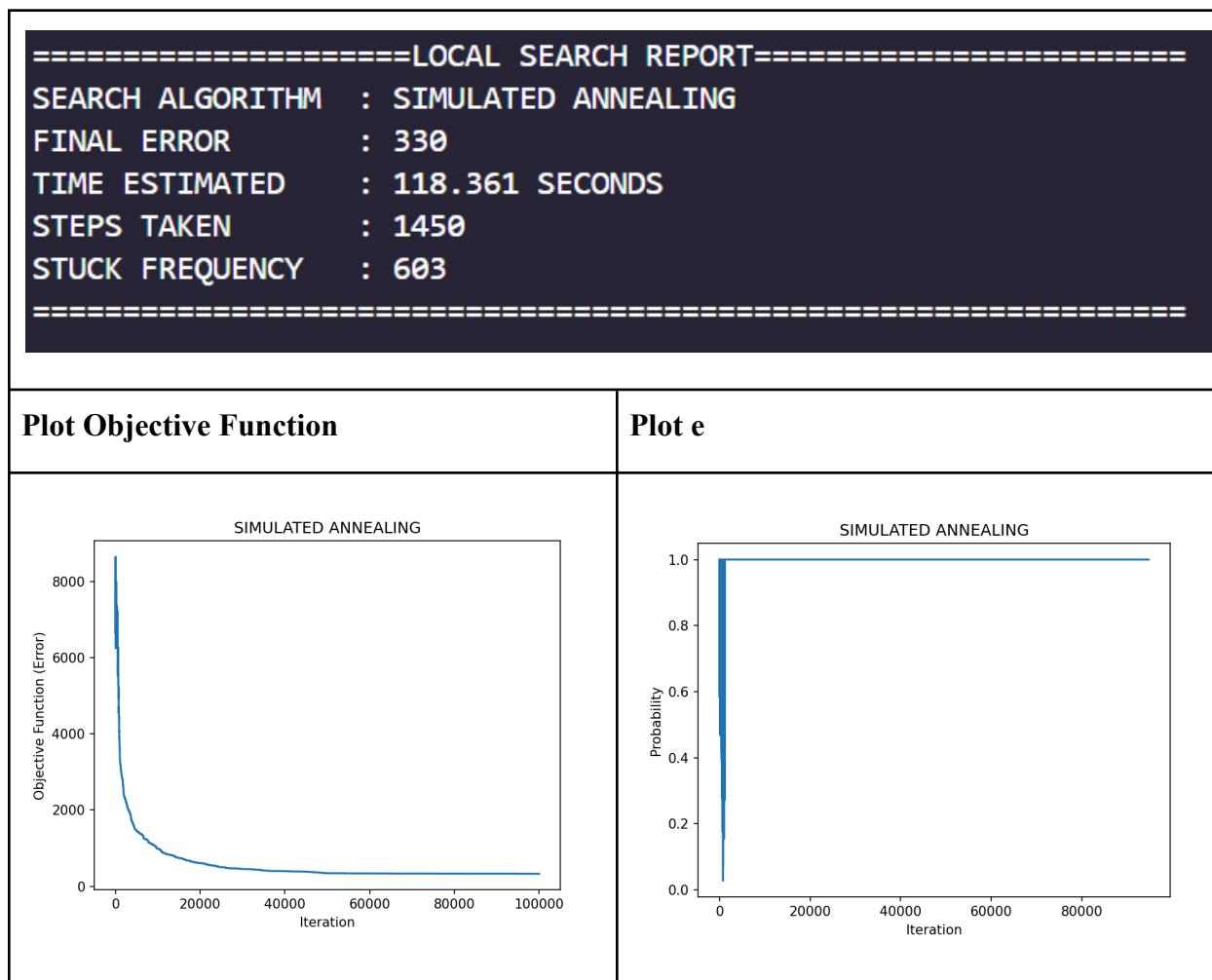
Initial State	Final State
<p>INITIAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>	<p>FINAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>
<p>INITIAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>	<p>FINAL STATE</p>  <p>PRESS SPACEBAR TO CONTINUE</p>
Report	
<pre>=====LOCAL SEARCH REPORT=====</pre> <p>SEARCH ALGORITHM : STOCHASTIC HILL CLIMBING FINAL ERROR : 587 TIME ESTIMATED : 1.63225 SECONDS STEPS TAKEN : 30000</p> <pre>=====</pre>	
Plot Objective Function	



Pada percobaan pertama algoritma *Stochastic Hill Climbing*, diawali dengan sebuah *initial state* yang sudah memenuhi beberapa *constraint*. Selanjutnya diterapkan *Stochastic Hill Climbing* untuk mencari solusi dari *initial state* tersebut dengan *set max* iterasi sebanyak 30000. Didapatkan *state akhir* dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa *error*, yaitu 584 . Dibutuhkan waktu pencarian selama 1,63225 detik sampai memenuhi *error* 584 . Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi selalu terjadi penurunan nilai error sesuai dengan konsep *Stochastic Hill Climbing*.

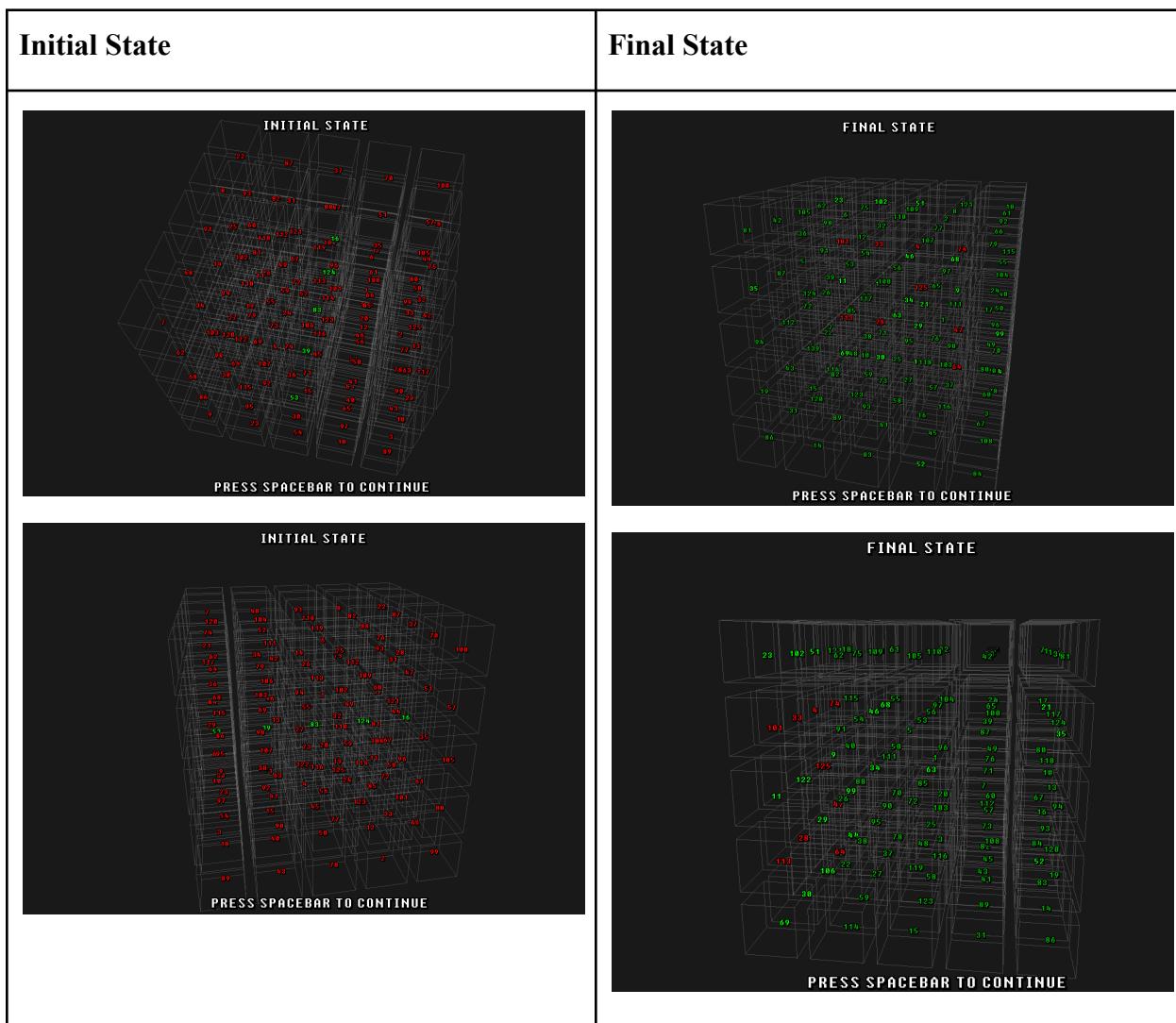
5. Simulated Annealing

Percobaan #1	100000 iterations
Initial State	Final State
	
	
Report	



Pada percobaan pertama algoritma *Simulated Annealing*, diawali dengan sebuah *initial state* tanpa ada yang memenuhi *constraint*. Selanjutnya diterapkan *Simulated Annealing* untuk mencari solusi dari *initial state* tersebut dengan *set max* iterasi sebanyak 10000. Didapatkan *state* akhir dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa *error*, yaitu 330. Dibutuhkan waktu pencarian selama 118,361 detik sampai memenuhi *error* 330 . Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi tidak selalu naik nilai *objective function*nya sesuai dengan konsep *Simulated Annealing*. Dilakukan juga pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai probabilitas.

Percobaan #2

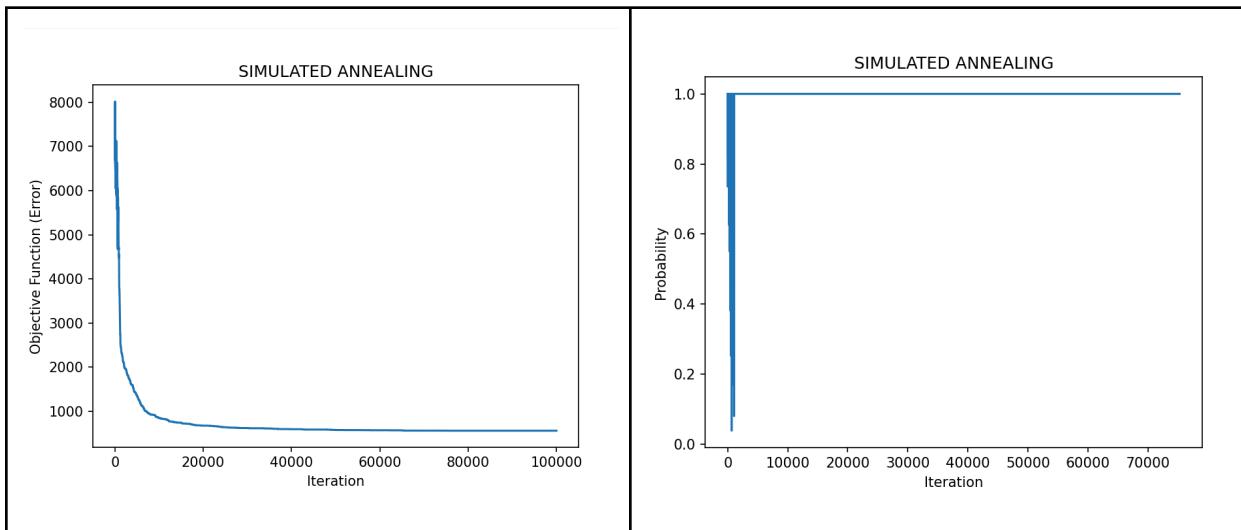


Report

```
=====LOCAL SEARCH REPORT=====
SEARCH ALGORITHM : SIMULATED ANNEALING
FINAL ERROR     : 564
TIME ESTIMATED   : 113.067 SECONDS
STEPS TAKEN     : 1307
STUCK FREQUENCY  : 584
```

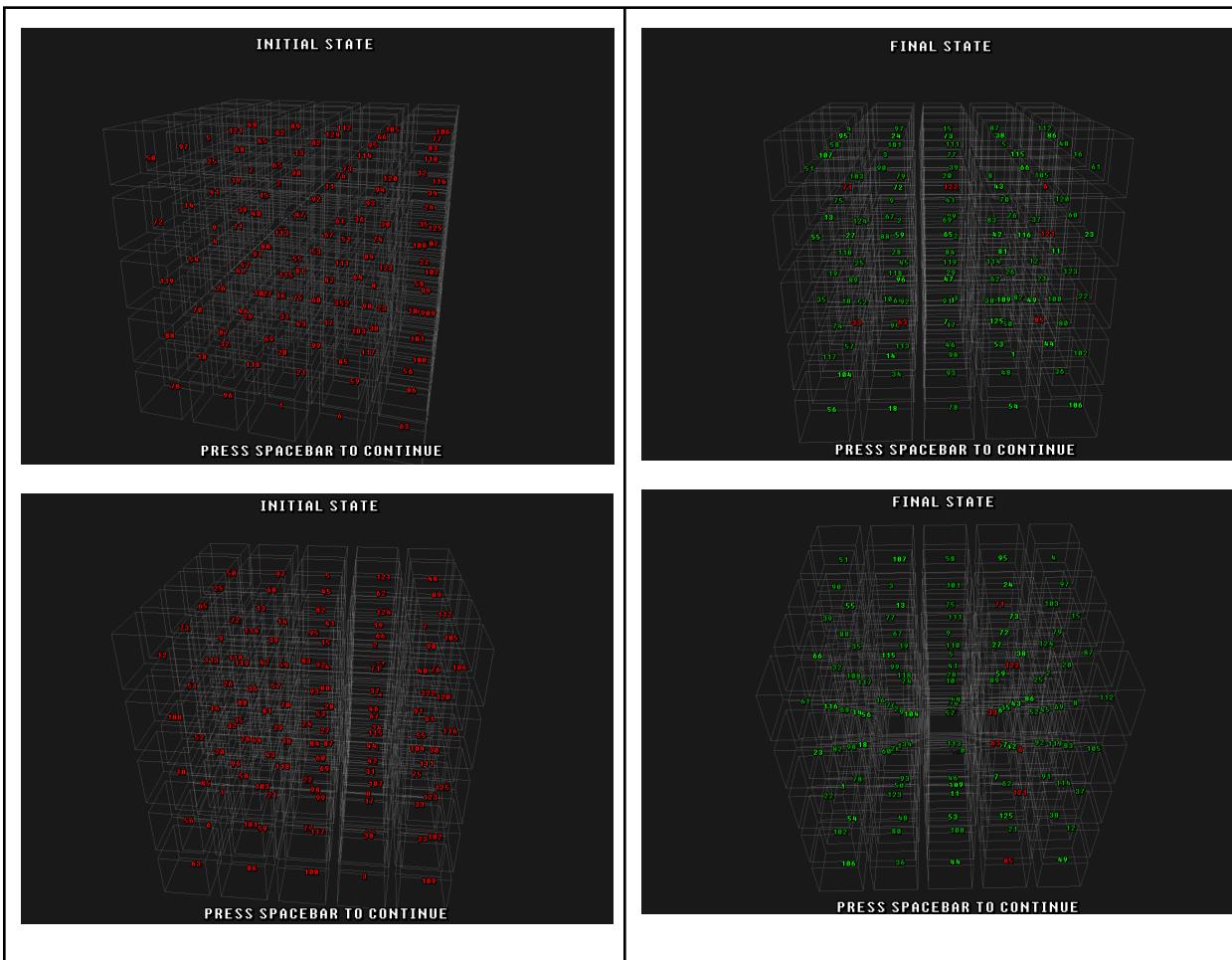
Plot Objective Function

Plot e



Pada percobaan kedua algoritma *Simulated Annealing*, diawali dengan sebuah *initial state* tanpa ada yang memenuhi *constraint*. Selanjutnya diterapkan *Simulated Annealing* untuk mencari solusi dari *initial state* tersebut dengan *set max* iterasi sebanyak 10000. Didapatkan *state* akhir dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa *error*, yaitu 564. Dibutuhkan waktu pencarian selama 113,067 detik sampai memenuhi *error* 564. Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi tidak selalu naik nilai *objective function*nya sesuai dengan konsep *Simulated Annealing*. Dilakukan juga pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai probabilitas.

Percobaan #3	
Initial State	Final State



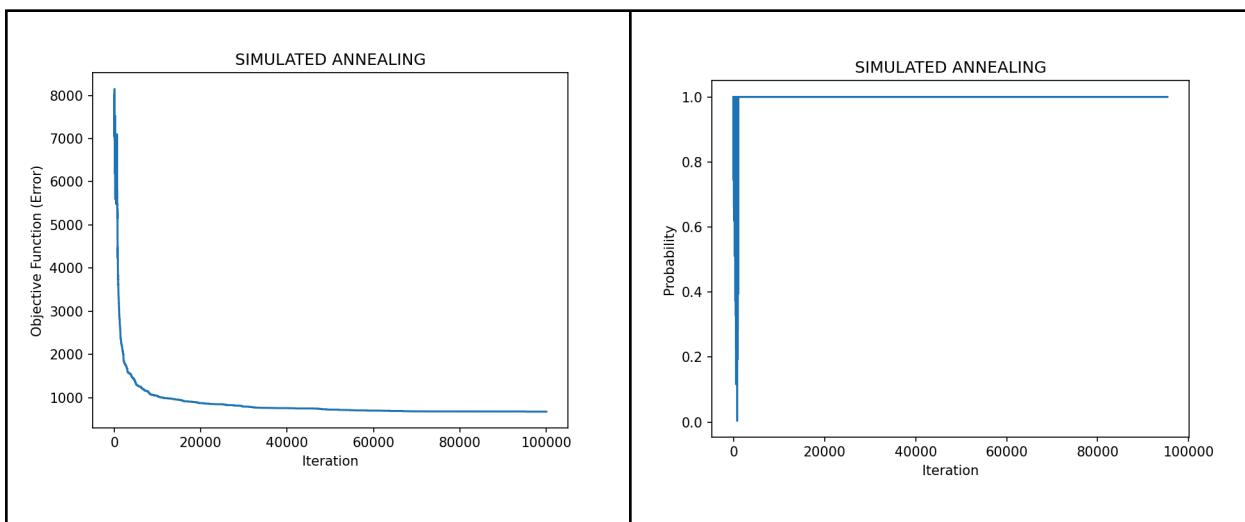
Report

===== LOCAL SEARCH REPORT =====

SEARCH ALGORITHM : SIMULATED ANNEALING
FINAL ERROR : 675
TIME ESTIMATED : 25.2536 SECONDS
STEPS TAKEN : 1407
STUCK FREQUENCY : 613

Plot Objective Function

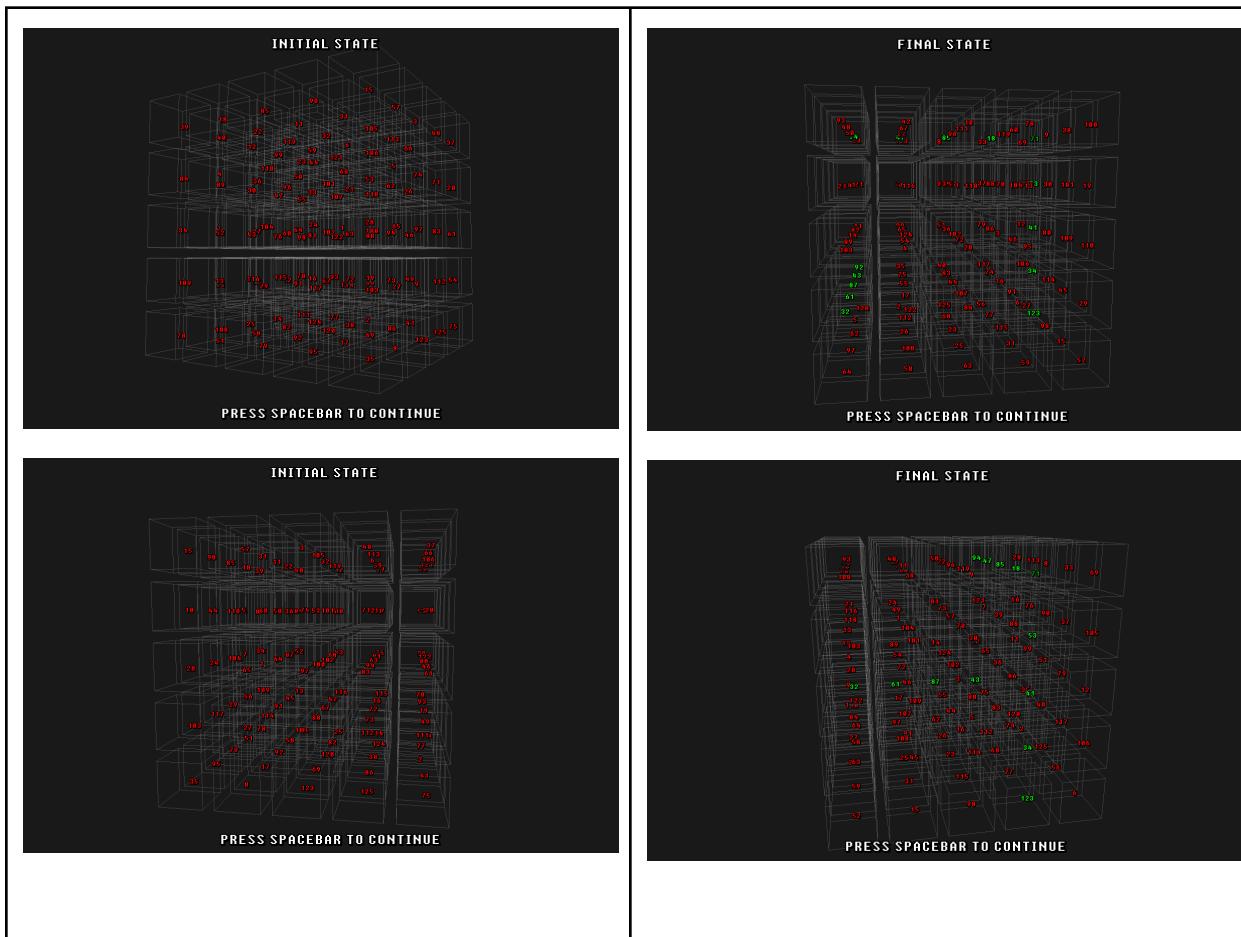
Plot e



Pada percobaan ketiga algoritma *Simulated Annealing*, diawali dengan sebuah *initial state* tanpa ada yang memenuhi *constraint*. Selanjutnya diterapkan *Simulated Annealing* untuk mencari solusi dari *initial state* tersebut dengan *set max* iterasi sebanyak 10000. Didapatkan *state* akhir dari *magic cube* seperti pada gambar di atas dengan nilai *objective function* berupa *error*, yaitu 675. Dibutuhkan waktu pencarian selama 25.2536 detik sampai memenuhi *error* 675. Setelah itu, dilakukan pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai *objective function* pada tiap iterasi yang dilakukan. Terlihat bahwa setiap iterasi tidak selalu naik nilai *objective function*nya sesuai dengan konsep *Simulated Annealing*. Dilakukan juga pemetaan grafik dengan sumbu x adalah iterasi ke-i dan sumbu y adalah nilai probabilitas.

6. Genetic Algorithm

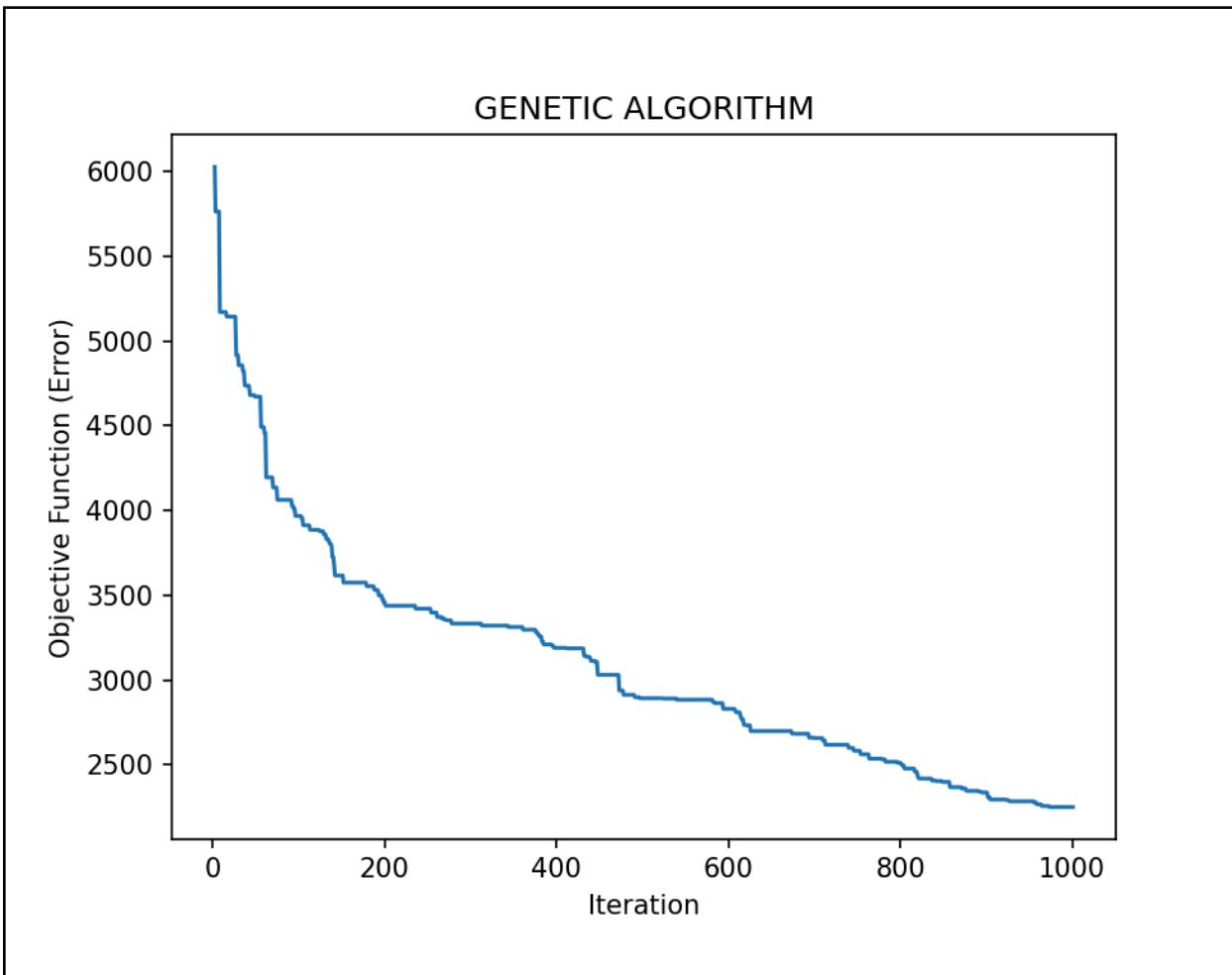
Percobaan #1	Jumlah Populasi : 100
Kontrol Populasi	Banyak Iterasi : 1000
Initial State	Final State



Report

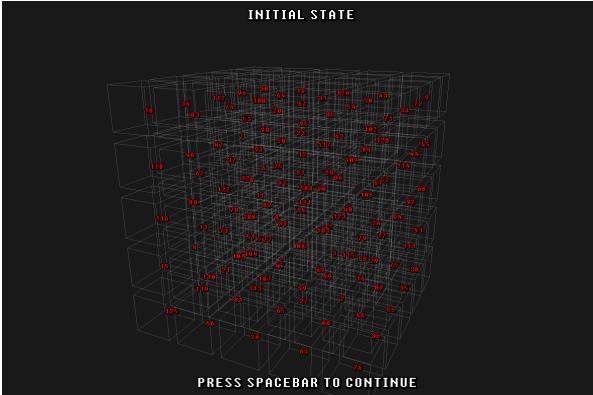
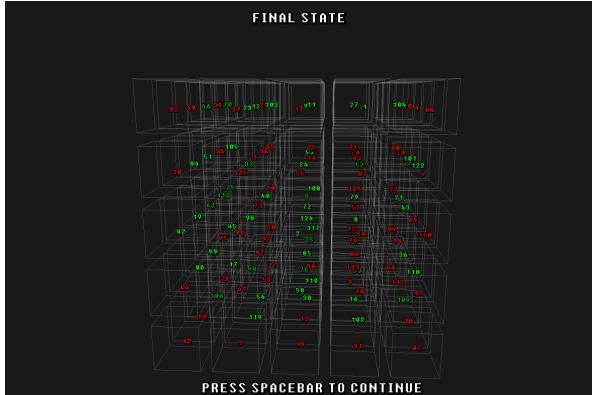
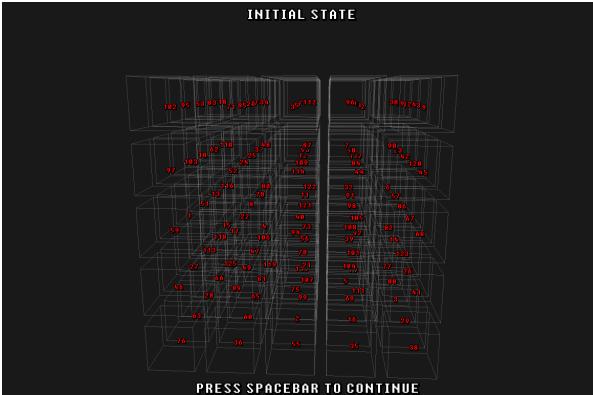
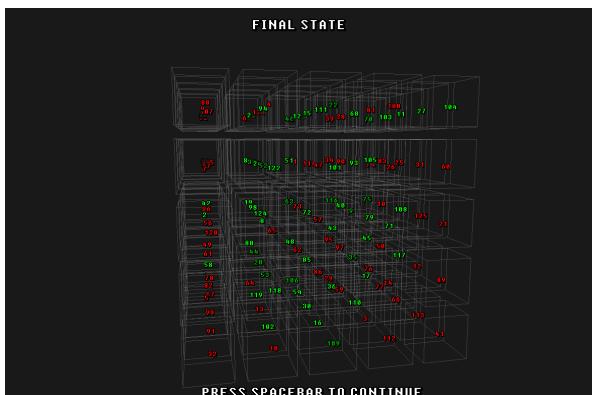
```
=====LOCAL SEARCH REPORT=====
SEARCH ALGORITHM : GENETIC ALGORITHM
FINAL ERROR      : 2251
TIME ESTIMATED   : 9.23021 SECONDS
STEPS TAKEN      : 1000
=====
```

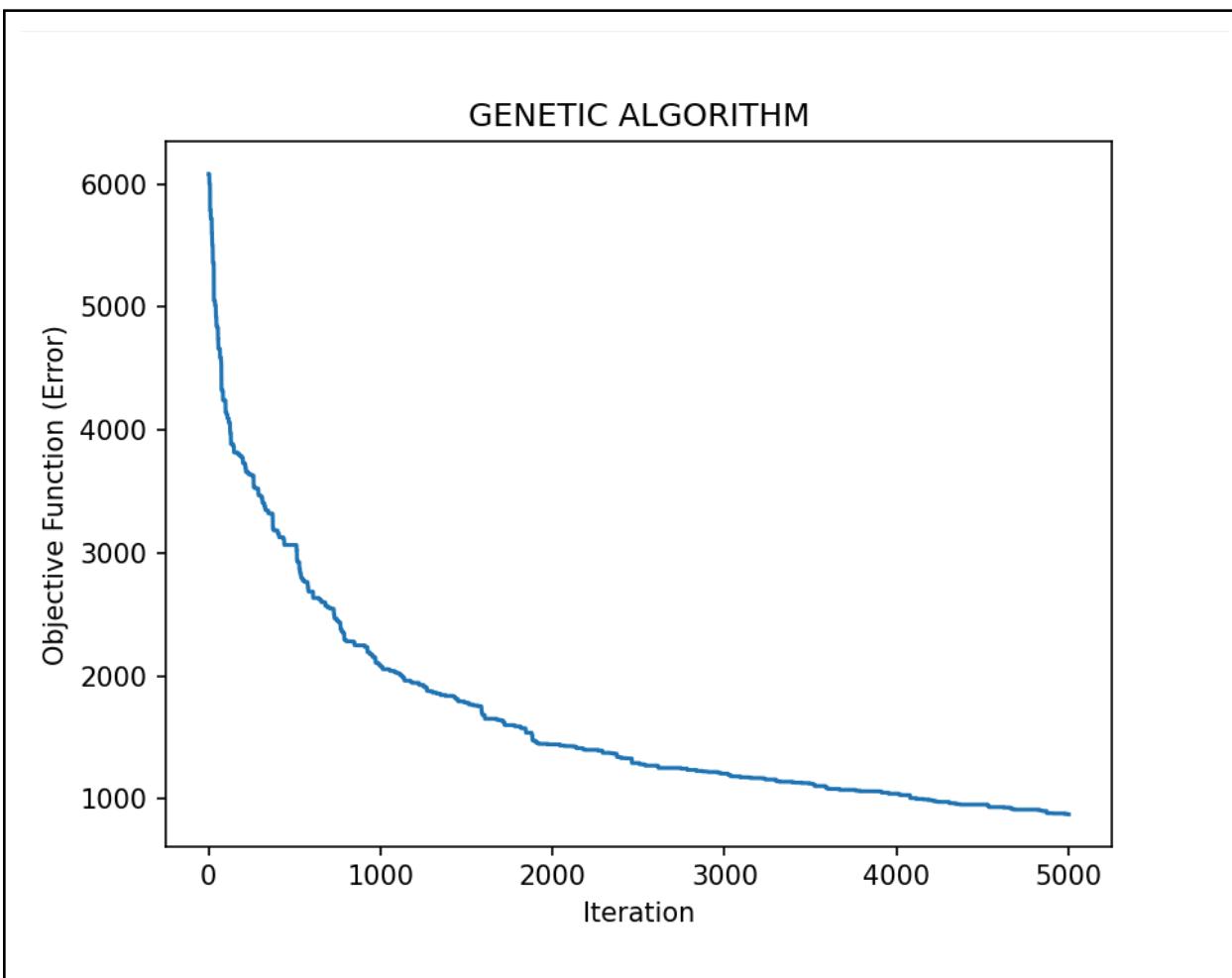
Plot Objective Function



Pada percobaan pertama algoritma *Genetic Algorithm*, diawali dengan inisialisasi populasi awal sebanyak 100 individu. Selanjutnya, algoritma melakukan proses seleksi, *crossover* dengan tingkat *crossover* sebesar 0.8, dan mutasi dengan *mutation rate* sebesar 0.05 untuk menghasilkan generasi-generasi baru yang lebih baik dalam mencari solusi optimal. Algoritma ini dijalankan dengan batas iterasi maksimum sebanyak 1000.

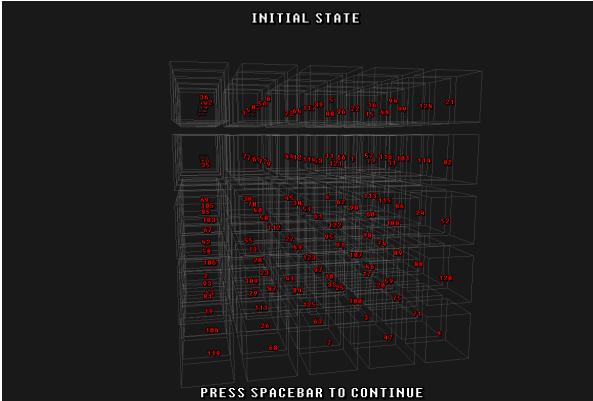
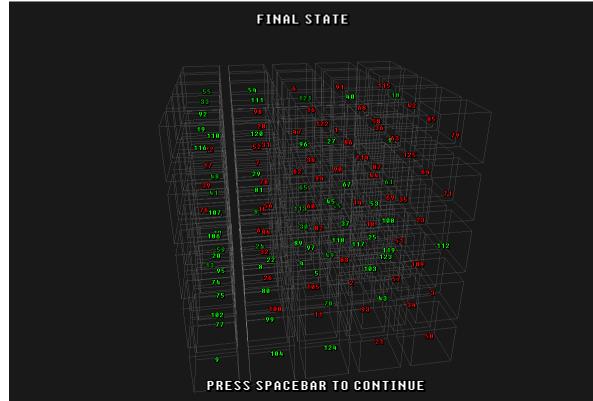
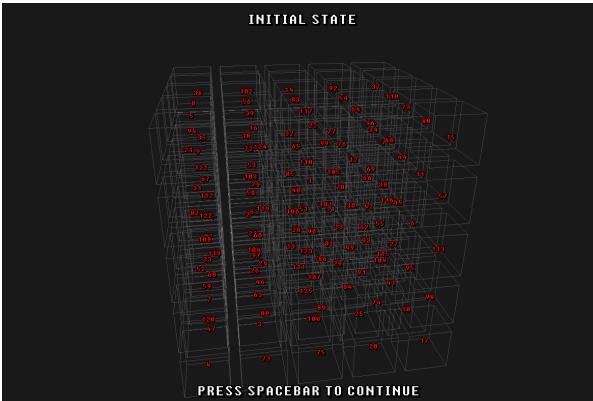
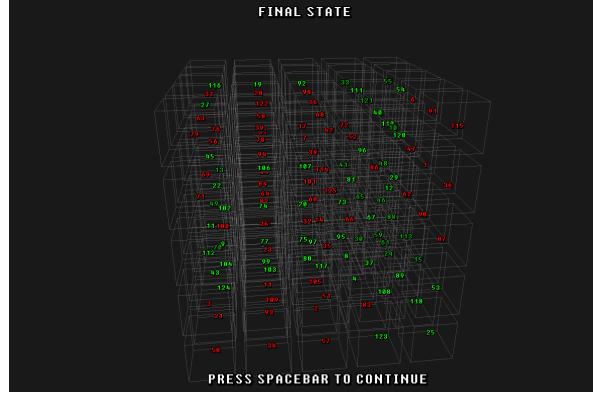
Didapatkan solusi akhir dengan nilai *objective function* berupa *error* sebesar 2251 setelah 1000 iterasi. Pencarian solusi membutuhkan waktu estimasi sebesar 9.23021 detik hingga mencapai *error* tersebut. Hasil ini kemudian dapat divisualisasikan dalam bentuk grafik dengan sumbu x sebagai iterasi ke-i dan sumbu y sebagai nilai objective function di tiap iterasi, yang akan menunjukkan penurunan error sesuai proses evolusi dari Genetic Algorithm.

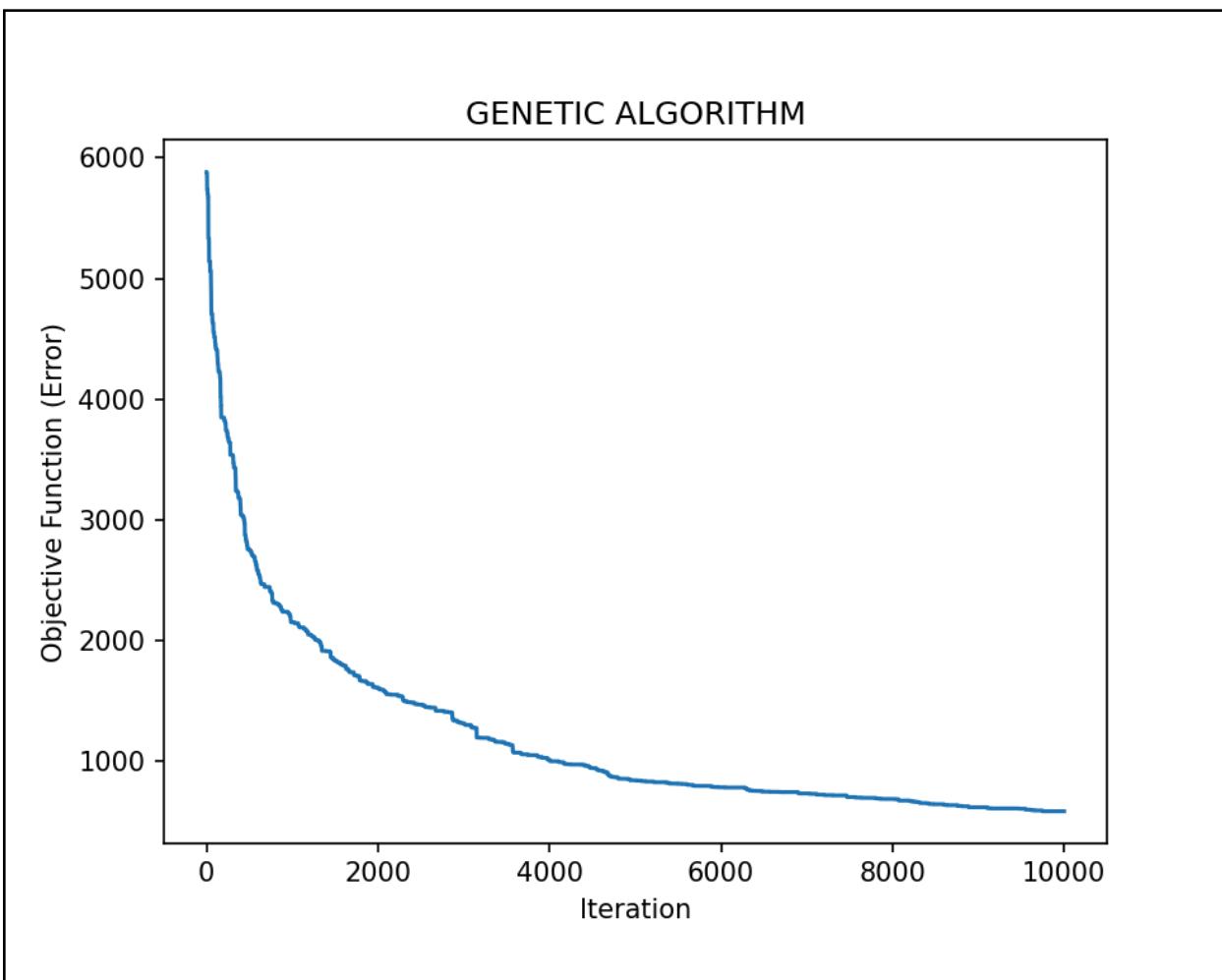
Percobaan #2	Jumlah Populasi : 100
Kontrol Populasi	Banyak Iterasi : 5000
Initial State	Final State
 <p>INITIAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>	 <p>FINAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>
 <p>INITIAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>	 <p>FINAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>
Report	<pre>=====LOCAL SEARCH REPORT===== SEARCH ALGORITHM : GENETIC ALGORITHM FINAL ERROR : 874 TIME ESTIMATED : 45.2126 SECONDS STEPS TAKEN : 5000 =====</pre>
Plot Objective Function	



Pada percobaan kedua algoritma *Genetic Algorithm*, diawali dengan inisialisasi populasi awal sebanyak 100 individu. Selanjutnya, algoritma melakukan proses seleksi, *crossover* dengan tingkat *crossover* sebesar 0.8, dan mutasi dengan *mutation rate* sebesar 0.05 untuk menghasilkan generasi-generasi baru yang lebih baik dalam mencari solusi optimal. Algoritma ini dijalankan dengan batas iterasi maksimum sebanyak 5000.

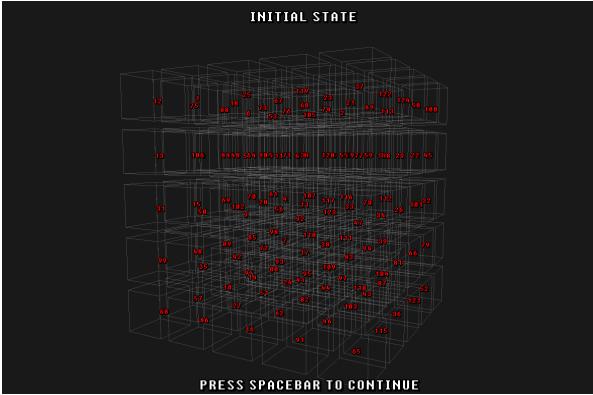
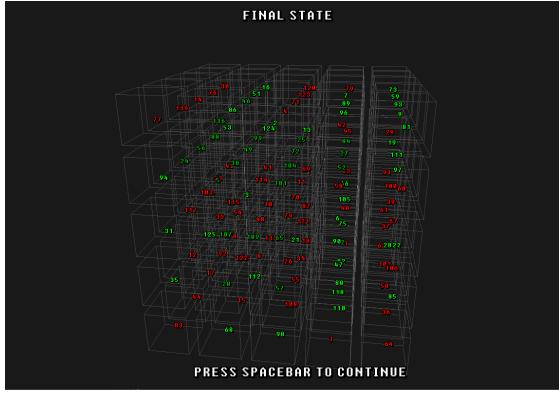
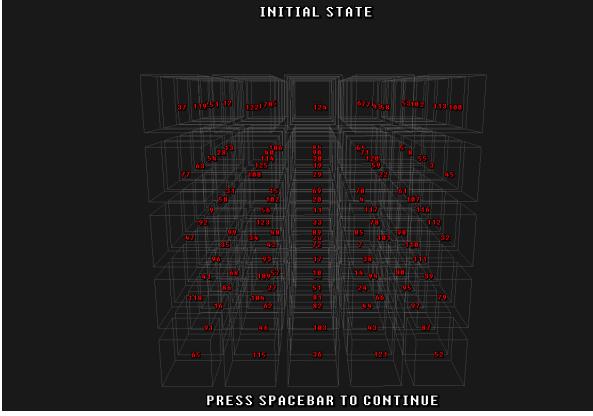
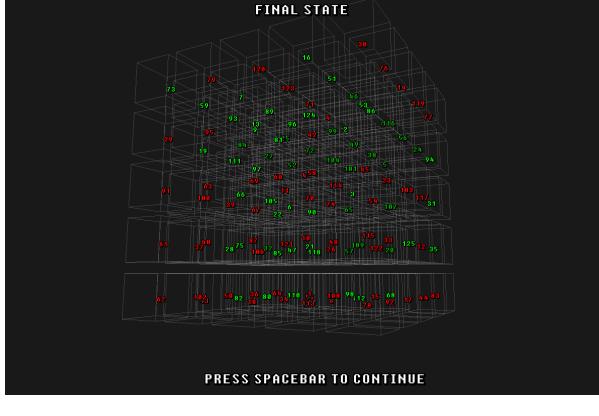
Didapatkan solusi akhir dengan nilai *objective function* berupa *error* sebesar 874 setelah 1000 iterasi. Pencarian solusi membutuhkan waktu estimasi sebesar 45.2126 detik hingga mencapai *error* tersebut. Hasil ini kemudian dapat divisualisasikan dalam bentuk grafik dengan sumbu x sebagai iterasi ke-*i* dan sumbu y sebagai nilai objective function di tiap iterasi, yang akan menunjukkan penurunan error sesuai proses evolusi dari Genetic Algorithm.

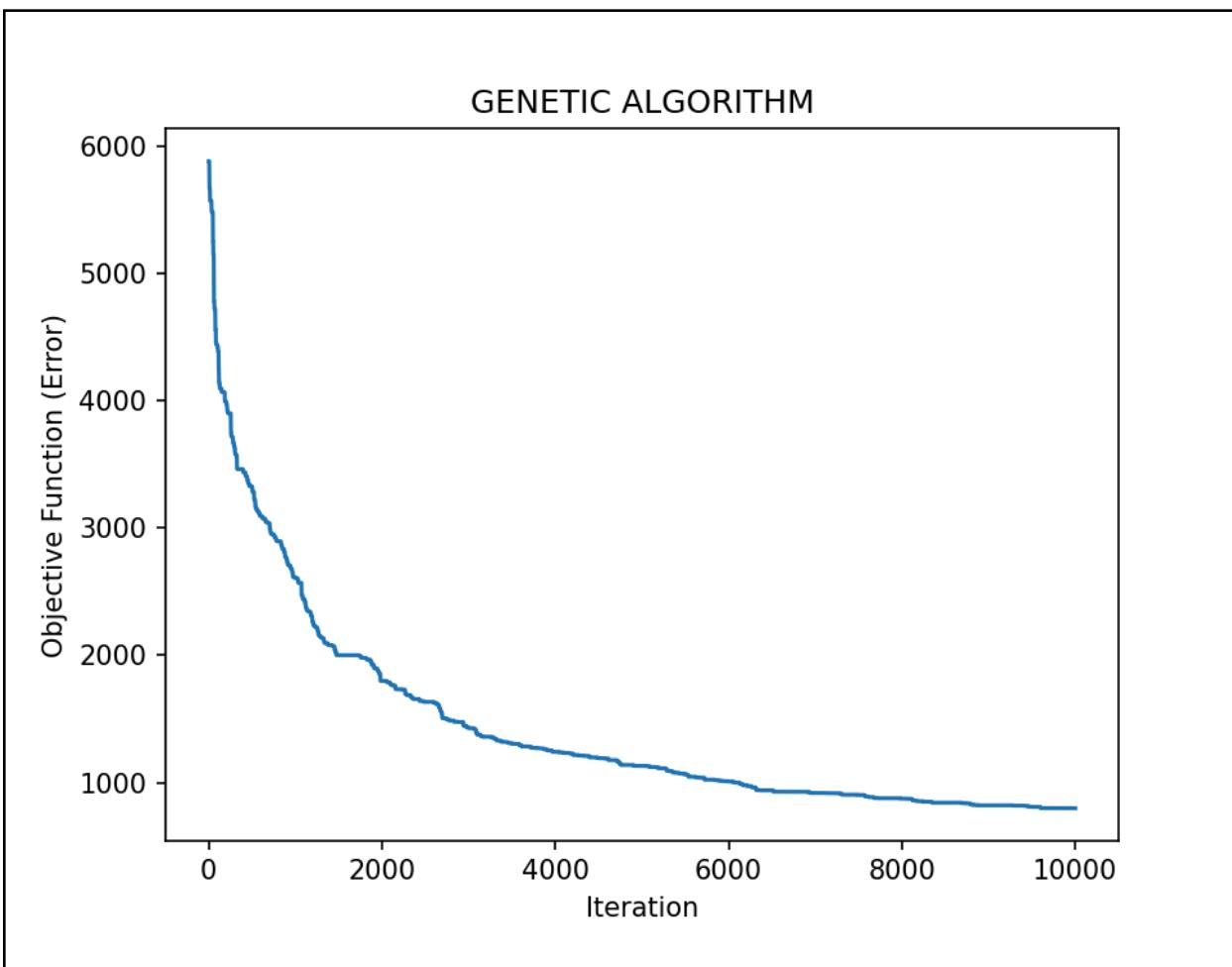
Percobaan #3	Jumlah Populasi : 100
Kontrol Populasi	Banyak Iterasi : 10000
Initial State	Final State
 <p>INITIAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>	 <p>FINAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>
 <p>INITIAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>	 <p>FINAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>
Report	
<pre>=====LOCAL SEARCH REPORT===== SEARCH ALGORITHM : GENETIC ALGORITHM FINAL ERROR : 589 TIME ESTIMATED : 88.9773 SECONDS STEPS TAKEN : 10000 =====</pre>	
Plot Objective Function	



Pada percobaan ketiga algoritma *Genetic Algorithm*, diawali dengan inisialisasi populasi awal sebanyak 100 individu. Selanjutnya, algoritma melakukan proses seleksi, *crossover* dengan tingkat *crossover* sebesar 0.8, dan mutasi dengan *mutation rate* sebesar 0.05 untuk menghasilkan generasi-generasi baru yang lebih baik dalam mencari solusi optimal. Algoritma ini dijalankan dengan batas iterasi maksimum sebanyak 10000.

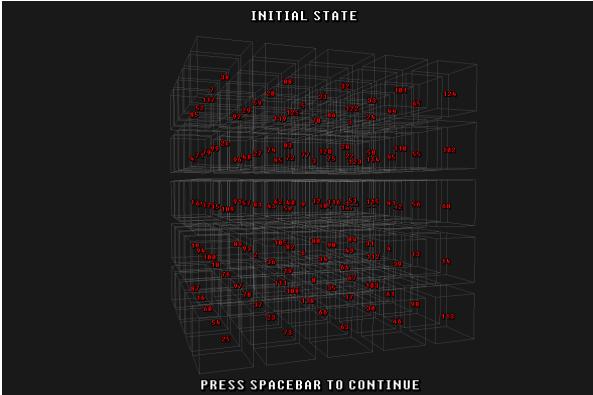
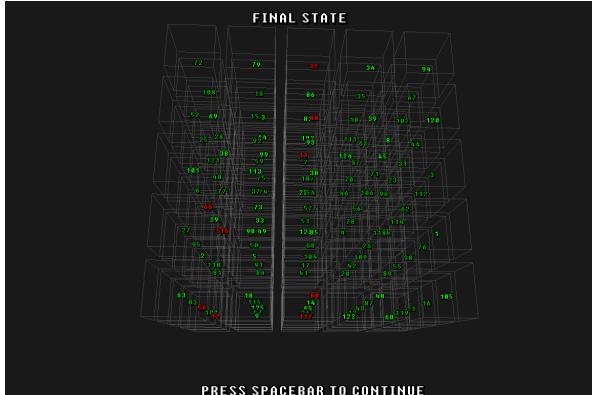
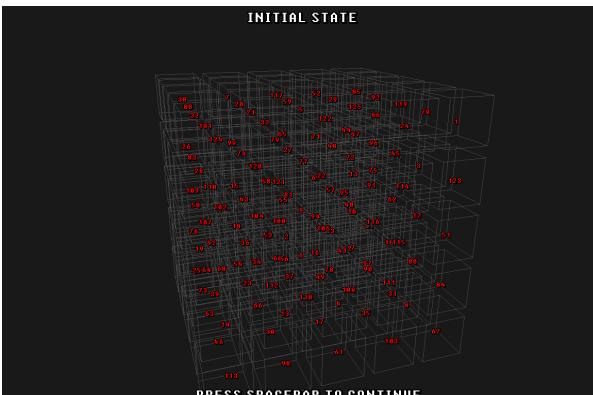
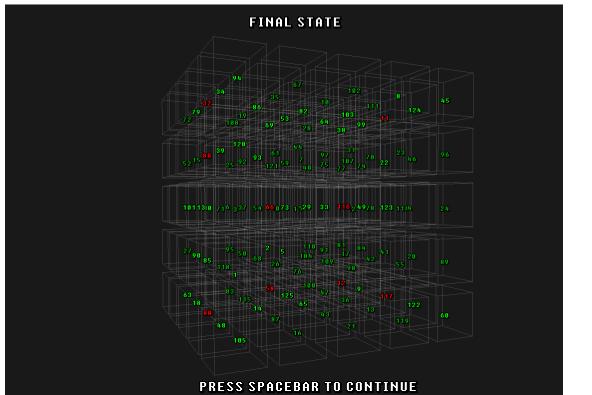
Didapatkan solusi akhir dengan nilai *objective function* berupa *error* sebesar 589 setelah 10000 iterasi. Pencarian solusi membutuhkan waktu estimasi sebesar 88.973 detik hingga mencapai *error* tersebut. Hasil ini kemudian dapat divisualisasikan dalam bentuk grafik dengan sumbu x sebagai iterasi ke-i dan sumbu y sebagai nilai objective function di tiap iterasi, yang akan menunjukkan penurunan error sesuai proses evolusi dari Genetic Algorithm.

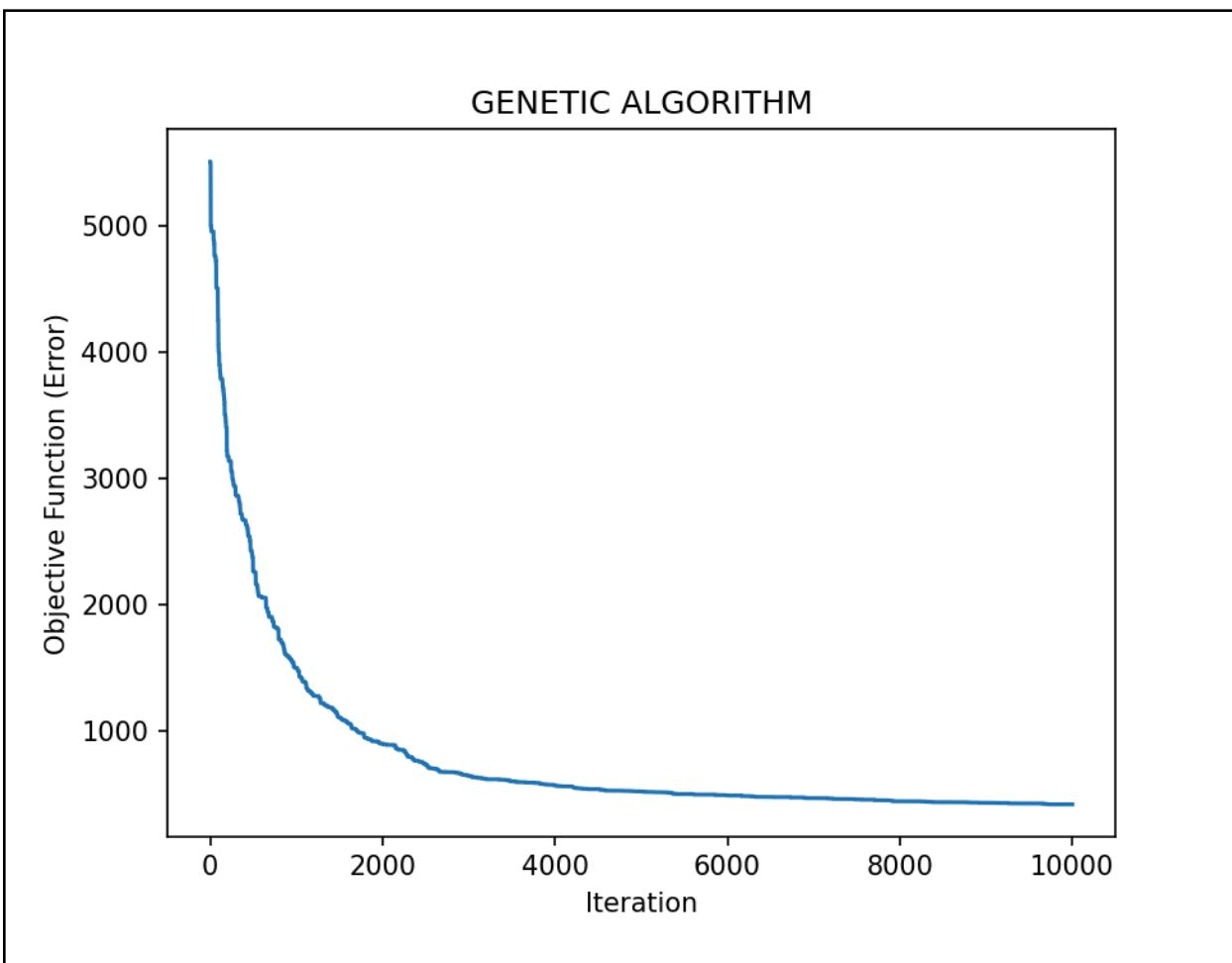
Percobaan #4	Jumlah Populasi : 100
Kontrol Iterasi	Banyak Iterasi : 10000
Initial State	Final State
 <p>INITIAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>	 <p>FINAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>
 <p>INITIAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>	 <p>FINAL STATE</p> <p>PRESS SPACEBAR TO CONTINUE</p>
Report	<pre>=====LOCAL SEARCH REPORT===== SEARCH ALGORITHM : GENETIC ALGORITHM FINAL ERROR : 798 TIME ESTIMATED : 87.5307 SECONDS STEPS TAKEN : 10000 =====</pre>
Plot Objective Function	



Pada percobaan keempat algoritma *Genetic Algorithm*, diawali dengan inisialisasi populasi awal sebanyak 100 individu. Selanjutnya, algoritma melakukan proses seleksi, *crossover* dengan tingkat *crossover* sebesar 0.8, dan mutasi dengan *mutation rate* sebesar 0.05 untuk menghasilkan generasi-generasi baru yang lebih baik dalam mencari solusi optimal. Algoritma ini dijalankan dengan batas iterasi maksimum sebanyak 10000.

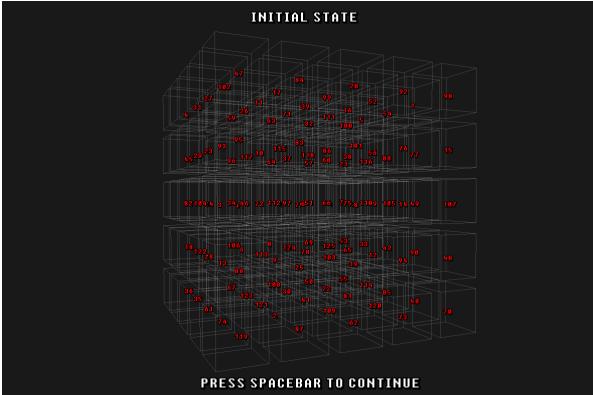
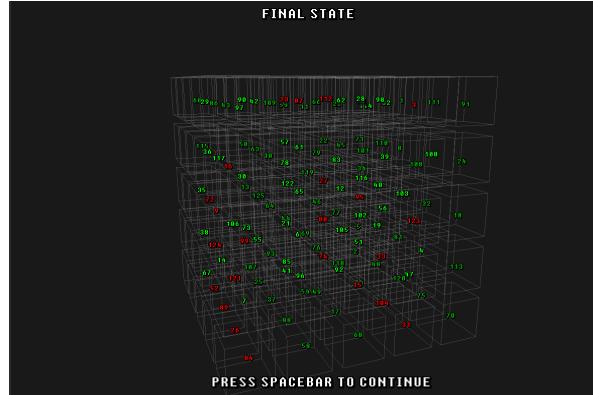
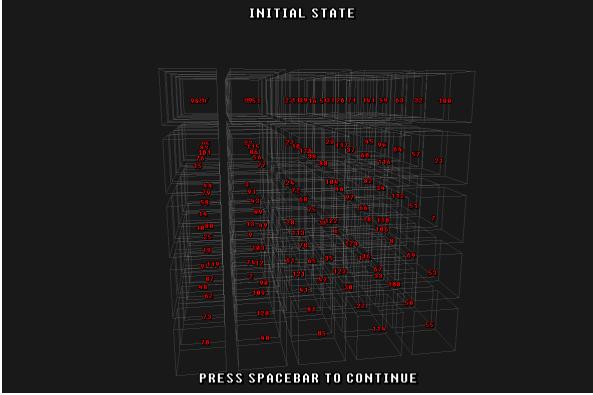
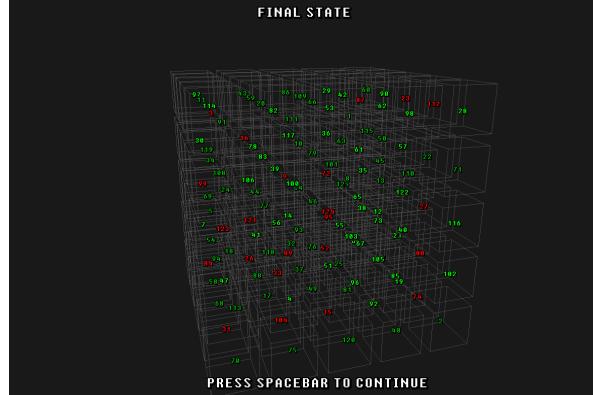
Didapatkan solusi akhir dengan nilai *objective function* berupa *error* sebesar 798 setelah 10000 iterasi. Pencarian solusi membutuhkan waktu estimasi sebesar 87.5307 detik hingga mencapai *error* tersebut. Hasil ini kemudian dapat divisualisasikan dalam bentuk grafik dengan sumbu x sebagai iterasi ke-i dan sumbu y sebagai nilai objective function di tiap iterasi, yang akan menunjukkan penurunan error sesuai proses evolusi dari Genetic Algorithm.

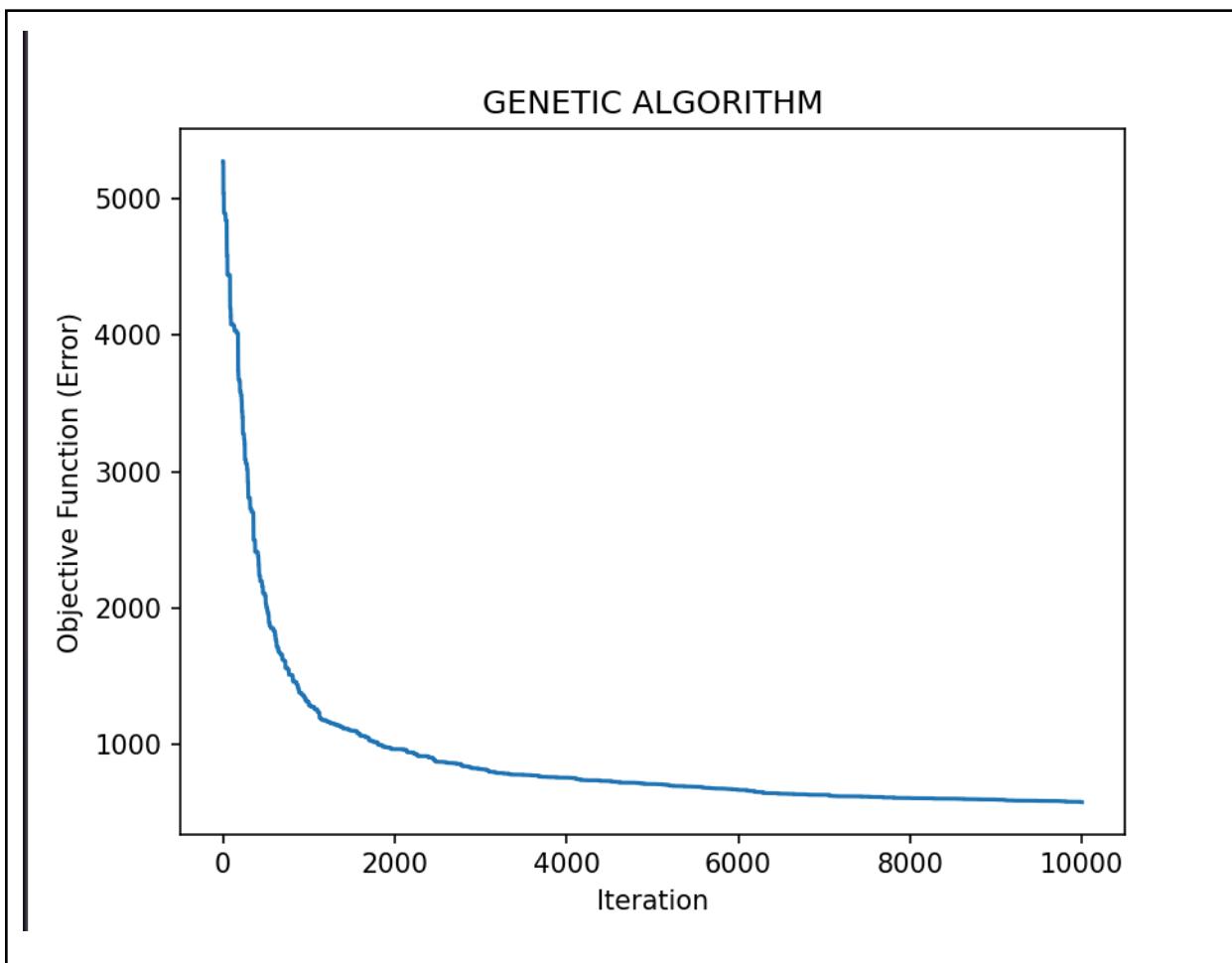
Percobaan #5	Jumlah Populasi : 500
Kontrol Iterasi	Banyak Iterasi : 10000
Initial State	Final State
 <p>INITIAL STATE PRESS SPACEBAR TO CONTINUE</p>	 <p>FINAL STATE PRESS SPACEBAR TO CONTINUE</p>
 <p>INITIAL STATE PRESS SPACEBAR TO CONTINUE</p>	 <p>FINAL STATE PRESS SPACEBAR TO CONTINUE</p>
Report	<pre>=====LOCAL SEARCH REPORT===== SEARCH ALGORITHM : GENETIC ALGORITHM FINAL ERROR : 419 TIME ESTIMATED : 806.61 SECONDS STEPS TAKEN : 10000 =====</pre>
Plot Objective Function	



Pada percobaan kelima algoritma *Genetic Algorithm*, diawali dengan inisialisasi populasi awal sebanyak 500 individu. Selanjutnya, algoritma melakukan proses seleksi, *crossover* dengan tingkat *crossover* sebesar 0.8, dan mutasi dengan *mutation rate* sebesar 0.05 untuk menghasilkan generasi-generasi baru yang lebih baik dalam mencari solusi optimal. Algoritma ini dijalankan dengan batas iterasi maksimum sebanyak 10000.

Didapatkan solusi akhir dengan nilai *objective function* berupa *error* sebesar 419 setelah 10000 iterasi. Pencarian solusi membutuhkan waktu estimasi sebesar 806.61 detik hingga mencapai *error* tersebut. Hasil ini kemudian dapat divisualisasikan dalam bentuk grafik dengan sumbu x sebagai iterasi ke-i dan sumbu y sebagai nilai objective function di tiap iterasi, yang akan menunjukkan penurunan *error* sesuai proses evolusi dari Genetic Algorithm.

Percobaan #6	Jumlah Populasi : 1000
Kontrol Iterasi	Banyak Iterasi : 10000
Initial State	Final State
	
	
Report	<pre>=====LOCAL SEARCH REPORT===== SEARCH ALGORITHM : GENETIC ALGORITHM FINAL ERROR : 575 TIME ESTIMATED : 1434.13 SECONDS STEPS TAKEN : 10000 =====</pre>
Plot Objective Function	



Pembahasan

Selanjutnya setelah melakukan eksperimen dengan sesuai ketentuan spesifikasi, kami melakukan analisis lebih lanjut untuk dapat meninjau algoritma terbaik. Peninjauan dilakukan berdasarkan efektivitas dan efisiensi performa dari setiap algoritma, yaitu **Jarak dengan Global Optima, Kubus hasil pencarian, Durasi Pencarian, Konsistensi data serta Banyak Iterasi.**

- **Jarak dengan Global Optima**

Algoritma	Steepest Ascent Hill Climbing			Hill-climbing with Sideways Move			Stochastic Hill-climbing		
Percobaan	1	2	3	1	2	3	1	2	3

Durasi pencarian	594	636	956	634	802	511	1166	694	587
Rata-rata Durasi Pencarian	728.6666667			649			815.6666667		

Algoritma	Simulated Annealing			Random Restart Hill Climbing			Genetic Algorithm					
	1	2	3	1	2	3	1	2	3	4	5	6
Percobaan												
Jarak Global Optimal (Objective-0)	330	564	675	469	542	428	225 1	874	589	798	419	575
Rata-rata Jarak Global Optima	523			479.6666667			917.6666667					

Analisis pertama kami lakukan dengan mempertimbangkan Jarak antara Score Objective Function dengan jarak terhadap global optima (score Objective Function =0). Berdasarkan tinjauan tersebut kami melakukan pencatatan jarak global dengan mengurangkan objective function dengan 0 (Global Optima). Sebagai penjelasan, kami melakukan pendekatan minimize dengan meninjau nilai error terhadap 315 dan memiliki Global Optima yaitu 0. sedangkan menurut analisis inisialisasi awal didapatkan score mencapai >8000, oleh karena itu score yang didapatkan cukup baik dengan tinjauan error < 10% dibanding error keseluruhan. Selanjutnya, berdasarkan perbandingan setiap algoritma dengan kondisi ideal kami melakukan pendekatan rata-rata dalam mengambil nilai eksak dari hasil percobaan. Setelah ditinjau rata-rata didapatkan yang paling mendekati global optima yaitu **Random Restart Hill Climbing** dengan memiliki score yaitu **479.6666667**

Setelah dilakukan analisis lebih lanjut, didapatkan bahwa pendekatan **Random Restart Hill Climbing** lebih efektif mendekati global optima karena memungkinkan algoritma menghindari jebakan local optima melalui beberapa inisialisasi acak. Selain itu, dengan pendekatan restart pada algoritma tersebut, algoritma mengeksplorasi solusi dari berbagai titik awal,

meningkatkan peluang menemukan solusi lebih optimal. Ini juga membantu mengurangi error secara bertahap dan memberikan hasil rata-rata yang lebih stabil, terutama pada ruang pencarian yang luas atau penuh local optima.

- **Perbandingan Objective Function Algoritma**

Algoritma	Steepest Ascent Hill Climbing	Hill-climbing with Sideways Move	Stochastic Hill-climbing	Simulated Annealing	Random Restart Hill Climbing	Genetic Algorithm
Rata-rata Objective Function	728.6666667	649	815.6666667	523	479.6666667	917.6666667

Selanjutnya, Analisis kami lakukan dengan membandingkan Score Objective Function dengan sesama Algoritma Selanjutnya, berdasarkan perbandingan setiap algoritma dengan kondisi ideal kami melakukan pendekatan rata-rata dalam mengambil nilai eksak dari hasil percobaan. Setelah ditinjau rata-rata didapatkan yang paling mendekati global optima yaitu **Random Restart Hill Climbing** dengan memiliki score yaitu **479.6666667**

- **Perbandingan Durasi Pencarian Setiap Algoritma**

Algoritma	Steepest Ascent Hill Climbing			Hill-climbing with Sideways Move			Stochastic Hill-climbing		
Percobaan	1	2	3	1	2	3	1	2	3
Durasi pencarian	9.761	12.25	9.948	10.03	11.863	12.104	0.696	1.108	1.632
Rata-rata Durasi Pencarian	10.653			11.33233333			1.145333333		

Algoritma	Simulated Annealing			Random Restart Hill Climbing			Genetic Algorithm					
	1	2	3	1	2	3	1	2	3	4	5	6
Percobaan	118.3 61	113. 067	25.2 53	49.697 6	94.02 4	181.48 4	9.23	45.216	88.97 87.53	806.6 1	1434. 13	
Rata-rata Durasi Pencarian	85.56033333			108.4023333			411.9476667					

Selanjutnya, Analisis dilakukan dengan membandingkan durasi pencarian dari setiap algoritma dengan sesama Algoritma. Pada pendekatan ini kami melakukan perhitungan rata-rata dari seluruh percobaan untuk setiap algoritma sebelum dibandingkan. Setelah dibandingkan, berdasarkan perbandingan antar setiap algoritma didapatkan yang paling mendekati efisien yaitu **Stochastic Hill-climbing** dengan durasi waktu yaitu 1.145333333

Berdasarkan analisis, dalam hal ini Stochastic Hill Climbing, Algoritma dapat berjalan lebih efisien karena memakai acuan pada jumlah batasan iterasi pada algoritma, terutama pada algoritma hanya dapat melakukan perubahan kalau nilai lebih baik dari solusi saat ini. Jika tidak dapat diabaikan. Pendekatan ini menghemat waktu pencarian dibandingkan metode lain yang mungkin mengevaluasi lebih banyak solusi atau melakukan restarts jika terjebak di local optima.

- Perbandingan Seberapa konsisten

Algoritma			Steepest Ascent Hill Climbing			Hill-climbing with Sideways Move			Stochastic Hill-climbing		
Percobaan			1	2	3	1	2	3	1	2	3
Jarak Global Optimal			594	636	956	634	802	511	1166	694	587

(Objective-0)												
Rata-rata Durasi Pencarian	728.6666667			649			815.6666667					
Standar Deviasi	197.9932659			146.0787459			308.0784532					
Persentase perubahan (%)	27.1719944			22.50828134			37.77014138					
Algoritma	Simulated Annealing			Random Restart Hill Climbing			Genetic Algorithm					
Percobaan	1	2	3	1	2	3	1	2	3	4	5	6
Jarak Global Optimal (Objective-0)	330	564	675	469	542	428	2251	874	589	798	419	575
Rata-rata Jarak Global Optima	523			479.6666667			917.6666667					
Standar Deviasi	176.1164388			57.74368652			673.5154539					
Persentase perubahan (%)	33.67427127			12.03829462			73.39434659					

Selanjutnya, Analisis dilakukan dengan membandingkan konsisten hasil dari setiap pencarian dari setiap algoritma dengan sesama Algoritma. Pada pendekatan ini kami menghitung rata-rata dari setiap score objective function dan meninjau standar deviasi dari seluruh algoritma untuk setiap percobaan. Selanjutnya, untuk meninjau seberapa baik perubahan tersebut, kami melakukan perhitungan standar deviasi dibandingkan dengan rata-rata objective function untuk meninjau persentase perubahan terhadap rata-rata objective function. Setelah dibandingkan, berdasarkan perbandingan antar setiap algoritma didapatkan yang paling mendekati konsisten yaitu **Random Restart Hill Climbing** dengan durasi waktu yaitu **12.03829462**

Dalam hal ini, setelah dilakukan analisis Random Restart Hill Climbing lebih konsisten karena melakukan restart dari berbagai titik awal dengan menggunakan algoritma steepest ascent hill climbing berulang kali sesuai max

iterations-nya. Hal ini membuat tentu hasil yang didapatkan akan lebih konsisten dan memiliki galat perubahan yang rendah dibanding yang lain.

- (*Genetic Algorithm*) Perbandingan banyak iterasi dan jumlah populasi

Algoritma	Genetic Algorithm		
Percobaan	1	2	3
Banyak Populasi	100	100	100
Banyak Generasi	1000	5000	10000
Hasil Objective Function	2251	874	589

Algoritma	Genetic Algorithm		
Percobaan	1	2	3
Banyak Populasi	100	500	1000
Banyak Generasi	10000	10000	10000
Hasil Objective Function	798	419	575

Selanjutnya untuk melakukan analisis daripada hasil *genetic algorithm*, maka dilakukan analisis perbandingan banyak iterasi dan jumlah populasi terhadap *objective function*. Pada percobaan dengan variasi jumlah generasi, terlihat bahwa semakin banyak generasi yang dijalankan, semakin baik hasil yang diperoleh. Dengan jumlah generasi yang lebih tinggi, algoritma memiliki lebih banyak kesempatan untuk melakukan proses seleksi, *crossover*, dan mutasi, sehingga dapat memperbaiki solusi secara bertahap dan mendekati solusi optimal.

Sementara itu, pada percobaan dengan variasi jumlah populasi, peningkatan populasi juga memberikan dampak positif terhadap hasil. Populasi yang lebih besar memungkinkan algoritma mengeksplorasi lebih banyak variasi solusi dalam setiap generasi, sehingga meningkatkan peluang untuk menemukan

solusi yang lebih baik. Dengan populasi yang besar, keberagaman solusi tetap terjaga, dan algoritma memiliki potensi lebih besar untuk menghindari solusi lokal yang kurang optimal.

Hasil terbaik dicapai pada kombinasi jumlah populasi 500 dan jumlah generasi 10000, karena kombinasi ini memberikan keseimbangan antara eksplorasi variasi solusi dan jumlah iterasi yang cukup untuk memperbaiki solusi secara bertahap. Populasi 500 memberikan keragaman yang memadai tanpa mengorbankan efisiensi, sementara 10000 generasi memastikan algoritma memiliki cukup waktu untuk mencapai solusi yang optimal.

III. Kesimpulan dan Saran

Berdasarkan penerapan dan analisis yang sudah dilakukan pada setiap algoritma *local search* untuk kasus “Pencarian Solusi Diagonal Magic Cube dengan Local Search”, kami menyimpulkan bahwa algoritma *random restart hill climbing* memiliki performa paling baik dalam mendekati solusi global optimal. Hal tersebut ditandai dengan rata-rata error terendah yang didapat dari seluruh eksperimen yang sudah kami lakukan. Ini menunjukkan bahwa algoritma *random restart hill* yang kami terapkan mampu memberikan hasil yang paling stabil. Di sisi lain, jika kami meninjau efisiensi yang lebih tinggi dari segi waktu pencarian, maka algoritma *stochastic hill climbing* menunjukkan efisiensi yang paling tinggi. Hal tersebut disebabkan karena algoritma ini hanya mengevaluasi suksesor secara acak dan bergerak ketika hanya terdapat perbaikan.

Algoritma yang paling boros dalam segi efisiensi waktu pencarian adalah *genetic algorithm*. Hal tersebut dapat terjadi akibat algoritma ini memerlukan waktu komputasi yang lebih besar karena menerapkan konsep yang terbilang paling kompleks, yaitu pengolahan populasi yang besar, operasi mutasi, dan operasi *crossover*.

Dalam pengimplementasian lebih lanjut, disarankan untuk menggunakan algoritma sesuai dengan kebutuhan penggunaan. Jika aspek yang diprioritaskan adalah kecepatan dalam mencari solusi, maka penggunaan *stochastic hill climbing* sangat disarankan. Di sisi lain, jika aspek yang

diprioritaskan adalah mencari solusi yang lebih akurat dan stabil, maka penggunaan *random restart hill climbing* sangat dianjurkan karena terbukti menghasilkan solusi yang paling mendekati kondisi optimal. Kami juga menyarankan untuk menyesuaikan setiap parameter algoritma sesuai dengan kebutuhan kasus. Contohnya pada *genetic algorithm*, jumlah populasi dan tingkat mutasi dapat disesuaikan agar memperoleh keseimbangan antara kecepatan dan akurasi pencarian.

IV. Pembagian Tugas

NIM	Tugas
18222060	<ol style="list-style-type: none">1. Membuat dan mengimplementasikan algoritma <i>genetic algorithm</i>2. Menyusun dokumen bagian pembahasan3. Menyusun dokumen laporan bagian hasil eksperimen dan analisis untuk algoritma <i>hill climbing with sideways move, stochastic hill climbing, random restart hill climbing, simulated annealing, genetic algorithm</i>
18222072	<ol style="list-style-type: none">1. Membuat dan mengimplementasikan algoritma <i>hill climbing with sideways move</i>.2. Membuat dan mengimplementasikan algoritma <i>stochastic hill climbing</i>.3. Menyusun dokumen laporan bagian abstrak dan deskripsi persoalan4. Menyusun dokumen laporan bagian implementasi algoritma <i>steepest ascent hill climbing, hill climbing with sideways move, stochastic hill climbing</i>5. Menyusun dokumen laporan bagian hasil eksperimen dan analisis untuk algoritma <i>steepest ascent hill climbing, hill climbing with sideways move, dan stochastic hill climbing</i>

	<ol style="list-style-type: none"> 6. Menyusun dokumen laporan bagian kesimpulan dan saran 7. Membuat README pada repository github
18222083	<ol style="list-style-type: none"> 1. Menyusun dan membuat Algoritma Objective Function 2. Menyusun dan Membuat Algoritma Local search Steepest Ascent Hill Climbing & Simulated Annealing 3. Membuat Function JumlahSkor (Heuristics) , Inisialisasi Random Cube , generate_neighbors & swap elements 4. Menyusun Dokumen penjelasan bagian Objective Function 5. Menyusun Dokumen penjelasan bagian Steepest Ascent Hill Climbing & Simulated Annealing 6. Menyusun Pembahasan terkait analisis hasil eksperimen yang terdiri dari 5 pertanyaan utama dalam membandingkan
18222091	<ol style="list-style-type: none"> 1. Membuat struktur file & repository Github untuk branch master dan opsi menjalankan program tanpa plot (without-plot) 2. Membuat visualisasi kubus menggunakan OPENGL (cpp) dengan library GLAD, GLFW dan GText 3. Membuat CLI interface untuk program secara keseluruhan 4. Melakukan integrasi pada setiap algoritma agar dapat ditampilkan pada OPENGL interface 5. Membuat algoritma random restart hill climbing 6. Menyusun dan memperbaiki struct Result untuk kebutuhan visualisasi plot 7. Membuat visualisasi plot probabilitas dan objective function menggunakan python matplotlib cpp wrapper 8. Melakukan testing dan menulis laporan pada setiap algoritma meliputi ketentuan spek, initial state, final state, report, serta visualisasi plot. 9. Membuat laporan bagian penjelasan program dan visualisasi

	warna pada kubus
--	------------------

V. Referensi

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

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

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

Complexica. (n.d.). Local search. Retrieved from
<https://www.complexica.com/narrow-ai-glossary/local-search>

GeeksforGeeks. (n.d.). *Local search algorithm in artificial intelligence*. Retrieved from
<https://www.geeksforgeeks.org/local-search-algorithm-in-artificial-intelligence/>