

**TUGAS BESAR 1 IF3070 DASAR INTELEGensi ARTIFISIAL
PENCARIAN SOLUSI DIAGONAL *MAGIC CUBE*
DENGAN *LOCAL SEARCH***



DISUSUN OLEH KELOMPOK 17 :

ANTHONY BRYANT GOUW	18222033
CHRISTOPHER RICHARD CHANDRA	18222057
RICHIE LEONARDO	18222071
JOSIA RYAN JULIANDY SILALAHI	18222075

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

DAFTAR ISI

DAFTAR ISI.....	2
I. DESKRIPSI PERSOALAN.....	3
II. PEMBAHASAN.....	3
1. Pemilihan Objective Function.....	3
2. Penjelasan Implementasi Algoritma Local Search.....	4
2.1 Steepest Ascent Hill Climbing.....	4
2.2 Hill-Climbing with Sideways Move.....	7
2.3 Random Restart Hill-Climbing.....	9
2.4 Stochastic Hill-Climbing.....	13
2.5 Simulated Annealing.....	16
2.6 Genetic Algorithm.....	20
3. Hasil Visualisasi Eksperimen.....	27
3.1 Steepest Ascent Hill-Climbing.....	27
3.2 Hill-Climbing with Sideways Move.....	27
3.3 Random Restart Hill-Climbing.....	27
3.4 Stochastic Hill-Climbing.....	27
3.5 Simulated Annealing.....	27
3.6 Genetic Algorithm.....	27
4. Analisis.....	27
4.1 Steepest Ascent Hill-Climbing.....	27
4.2 Hill-Climbing with Sideways Move.....	27
4.3 Random Restart Hill-Climbing.....	27
4.4 Stochastic Hill Climbing.....	27
4.5 Simulated Annealing.....	28
4.6 Genetic Algorithm.....	28
III. KESIMPULAN DAN SARAN.....	28
IV. LINK SOURCE CODE.....	28
V. PEMBAGIAN TUGAS TIAP ANGGOTA KELOMPOK.....	28
VI. REFERENSI.....	29

I. DESKRIPSI PERSOALAN

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

Gambar 1.1 Diagonal Magic Cube 5x5x5

(Sumber : Spesifikasi Tugas Besar 1 Dasar Inteligensi Artifisial)

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

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

Pada dokumen ini, akan membahas mengenai **Diagonal Magic Cube** berukuran 5 x 5 x 5. Terdapat *initial state* dari kubus adalah susunan angka 1 hingga 5^3 secara acak. Kemudian, tiap iterasi pada algoritma local search, langkah yang boleh dilakukan adalah menukar posisi dari 2 angka pada kubus tersebut (2 angka yang ditukar tidak harus bersebelahan).

II. PEMBAHASAN

1. Pemilihan Objective Function

Dalam memilih *objective function*, hal yang terlebih dahulu diketahui adalah menemukan *magic number* dari magic cube berukuran 5x5x5. Dilansir dari https://en.wikipedia.org/wiki/Magic_cube, formula untuk menemukan magic number adalah sebagai berikut.

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

Sehingga, untuk kubus berukuran 5x5x5, magic numbernya adalah sebagai berikut.

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

Objective Function pada tugas ini adalah menghitung selisih mutlak antara jumlah angka pada setiap baris, kolom, tiang, dan diagonal dengan *magic number*, dan menjumlahkan semua selisih ini.

Oleh karena itu, penentuan strategi dari *objective function* adalah dengan meminimalisir perbedaan antara magic number yang sudah ditentukan dengan jumlah dari setiap angka pada masing masing pilar, baris, kolom, dan diagonal dengan total terdapat $\rightarrow 25$ baris + 25 kolom + 25 pilar + 4 tridiagonal + 10 diagonal bidang atas + 10 diagonal bidang kanan kiri + 10 bidang diagonal tengah $\leftarrow 109$ penjumlahan. Secara matematis formula tersebut dapat ditulis sebagai berikut :

$$\begin{aligned} \text{Objective Function } (h(x)) : & (\sum_{k=1}^N \sum_{j=1}^N f(\sum_{i=1}^N \text{cube}[i][j][k]) + f(\sum_{i=1}^N \text{cube}[j][k][i]) + f(\sum_{i=1}^N \text{cube}[j][i][k])) \\ & + (f(\sum_{i=1}^N \text{cube}[i][i][i]) + f(\sum_{i=1}^N \text{cube}[N-i][i][i]) + f(\sum_{i=1}^N \text{cube}[i][i][N-i]) + f(\sum_{i=1}^N \text{cube}[i][N-i][i])) \\ & + \sum_{j=1}^N (f(\sum_{i=1}^N \text{cube}[j][i][i]) + f(\sum_{i=1}^N \text{cube}[i][j][i])) + f(\sum_{i=1}^N \text{cube}[i][i][j]) + f(\sum_{i=1}^N \text{cube}[N-i][i][j]) \\ & + f(\sum_{i=1}^N \text{cube}[j][i][N-i]) + f(\sum_{i=1}^N \text{cube}[N-i][j][i])) \end{aligned}$$

M: Magic Number

cube: Array 3D Magic Cube

Dengan $f(x)$ pada formula di atas didefinisikan sebagai berikut :

$$f(x) = \begin{cases} 1 & \text{if } x \neq 315 \\ 0 & \text{if } x = 315 \end{cases}$$

2. Penjelasan Implementasi Algoritma Local Search

2.1 Steepest Ascent Hill Climbing

Steepest Ascent Hill-Climbing merupakan bentuk varian algoritma local search Hill-climbing. Algoritma ini bekerja dengan berpindah ke seluruh *successor* yang memiliki nilai lebih baik dari *current state* saat ini.

Pada metode ini, algoritma yang sudah dibuat pertama akan melakukan pemeriksaan atau penghitungan *error* yang didapatkan sekarang. Kemudian, algoritma pengacak yang sudah dibuat akan berjalan dengan memindahkan satu angka ke setiap posisi yang mungkin dan dihitung *error* terbarunya. Apabila ditemukan *error* yang lebih kecil, maka posisi tersebut akan disimpan dan algoritma akan

berulang sampai tidak ditemukan *error* yang lebih kecil. Hal ini memungkinkan algoritma untuk *stuck* pada *local minima* ataupun *flat*.

Pada intinya, algoritma ini akan melakukan *swap* terus menerus dan berhenti ketika menemukan *error* yang lebih kecil atau sama dengan *error minimum* yang pertama kali ditemukan. Algoritma ini dapat digunakan, tetapi sangat rentan terjebak pada kondisi *local maximum* dan tidak dapat menjamin sampai ke *global maximum*.

Implementasi kode adalah sebagai berikut.

```
#include "cube.hpp"
#include <steep_ascent.hpp>

void steep_ascent::hill_climbing() {

    while (true) {

        cube::errInfo neighbor;
        cube::errInfo targetVal;

        int currentErr = cube::objective_func();

        for (int i = 0; i < cube::N; i++) {
            for (int j = 0; j < cube::N; j++) {
                for (int k = 0; k < cube::N; k++) {
                    cube::errInfo temp = cube::swap_cube(i, j, k);
                    if (neighbor.error > temp.error) {
                        neighbor.error = temp.error;
                        neighbor.x = temp.x;
                        neighbor.y = temp.y;
                        neighbor.z = temp.z;

                        targetVal.x = i;
                        targetVal.y = j;
                        targetVal.z = k;
                    }
                }
            }
        }

        if (neighbor.error < currentErr) {
            cube::swap(targetVal.x, targetVal.y, targetVal.z, neighbor.x,
                      neighbor.y, neighbor.z);
        }

        if (currentErr <= neighbor.error) {
            std::cout << "==== Steepest Ascent Finished ====" <<
            std::endl;
            return;
        }
    }
}
```

Adapun penjelasan terkait implementasi kode di atas adalah sebagai berikut.

Pertama akan dideklarasikan **neighbor** dan **targetVal**. **Neighbor** akan menyimpan informasi tetangga dengan *state* terbaik dan **targetVal** adalah menyimpan posisi angka yang mungkin ditukar agar mendapatkan *error* yang lebih kecil.

```
cube::errInfo neighbor;
cube::errInfo targetVal
```

Selanjutnya, akan diambil *error* awal dari *initial state* kubus yang disimpan pada variabel **currentErr**.

```
int currentErr = cube::objective_func();
```

Lalu, akan dicari susunan dari kubus 3 dimensi dengan *error* yang paling rendah. Untuk setiap elemen **(i, j, k)** dalam kubus, dilakukan *swap* menggunakan **cube::swap_cube(i, j, k)** yang mengembalikan *error* baru. Jika *error* dari *swap* ini lebih kecil dari *error* pada **neighbor**, maka **neighbor** diperbarui dengan *error* tersebut, dan **targetVal** diset ke elemen yang akan ditukar.

```
for (int i = 0; i < cube::N; i++) {
    for (int j = 0; j < cube::N; j++) {
        for (int k = 0; k < cube::N; k++) {
            // Melakukan pertukaran elemen pada posisi (i, j, k)
            cube::errInfo temp = cube::swap_cube(i, j, k);

            // Mengecek apakah error setelah swap lebih kecil dari error
            // neighbor saat ini
            if (neighbor.error > temp.error) {
                neighbor.error = temp.error;
                neighbor.x = temp.x;
                neighbor.y = temp.y;
                neighbor.z = temp.z;

                // Menyimpan elemen yang akan ditukar
                targetVal.x = i;
                targetVal.y = j;
                targetVal.z = k;
            }
        }
    }
}
```

Jika ditemukan susunan dengan *error* lebih kecil dibandingkan *error* awal (**currentErr**), maka program melakukan pertukaran menggunakan **cube::swap()**, memperbarui posisi elemen **targetVal** dan **neighbor**.

```
if (neighbor.error < currentErr) {
    cube::swap(targetVal.x, targetVal.y, targetVal.z, neighbor.x,
               neighbor.y, neighbor.z);
}
```

Proses terminasi terjadi ketika tidak ada **neighbor** yang memiliki *error* lebih kecil dari **currentErr**.

```
if (currentErr <= neighbor.error) {
    std::cout << "==== Steepest Ascent Finished ====" << std::endl;
    return;
}
```

2.2 Hill-Climbing with Sideways Move

Pada metode ini, Sama seperti *Steepest Ascent*, dimulai dengan *initial state*. Kemudian, algoritma pengacak yang sudah dibuat akan berjalan dengan memindahkan satu angka ke setiap posisi yang mungkin dan dihitung *error* terbarunya. Algoritma ini memungkinkan perpindahan *neighbor* secara lateral (*sideways*) ketika *current state* sama dengan *error minimum* sebelumnya, sehingga dapat menghindari kondisi *flat position*.

Kesimpulannya, algoritma ini tidak jauh berbeda dari *Steepest Ascent*. Hanya saja, algoritma ini dapat menghindari *flat region*. Algoritma ini juga tidak dapat menjamin dapat sampai ke posisi *global maximum*.

Implementasi kode adalah sebagai berikut.

```
#include "cube.hpp"
#include <side_ways.hpp>

void side_ways::hill_climbing() {
    const int max_sideways_moves = 100;
    int sideways_moves = 0;
    int total_moves = 0;

    while (true) {
        cube::errInfo neighbor;
        cube::errInfo targetVal;
        int currentErr = cube::objective_func();

        for (int i = 0; i < cube::N; i++) {
            for (int j = 0; j < cube::N; j++) {
                for (int k = 0; k < cube::N; k++) {
                    cube::errInfo temp = cube::swap_cube(i, j, k);
                    if (neighbor.error > temp.error || (neighbor.error == currentErr && temp.error == currentErr)) {
                        neighbor.error = temp.error;
                        neighbor.x = temp.x;
                        neighbor.y = temp.y;
                        neighbor.z = temp.z;

                        targetVal.x = i;
                        targetVal.y = j;
                        targetVal.z = k;
                    }
                }
            }
        }

        total_moves++;

        if (neighbor.error == currentErr) {
            sideways_moves++;
        }

        if (neighbor.error <= currentErr) {
            cube::swap(targetVal.x, targetVal.y, targetVal.z, neighbor.x, neighbor.y, neighbor.z);
            currentErr = neighbor.error;
        }
        if (neighbor.error > currentErr || sideways_moves >= max_sideways_moves) {
            if (sideways_moves >= max_sideways_moves) {
```

```

        std::cout << "==== Pencarian Sideways Move dihentikan karena
mencapai Sideways Move Maksimum====" << std::endl;
    } else {
        std::cout << "==== Pencarian Sideways dihentikan karena
mencapai local/global optima ====" << std::endl;
        std::cout << "==Algoritma diterminasi pada iterasi sideways
move ke: " << sideways_moves <<"=="<<std::endl;
    }
    return;
}
}
}

```

Adapun penjelasan terkait implementasi kode di atas adalah sebagai berikut.

Pertama, deklarasi beberapa variabel yang dibutuhkan. variabel yang dibutuhkan adalah **max_sideways_moves** yaitu batas maksimal sideways move yang diizinkan, **sideways_moves** adalah penghitung berapa kali sideways move sudah dilakukan dan **total_moves** adalah penghitung total iterasi atau langkah yang telah dilakukan selama pencarian.

```

const int max_sideways_moves = 100;
int sideways_moves = 0;
int total_moves = 0;

```

Selanjutnya, deklarasikan struktur neighbor. **neighbor** dan **targetVal** adalah struktur yang menyimpan informasi tentang tetangga dari keadaan saat ini dan target perubahan. **currentErr** menyimpan nilai error atau ketidakoptimalan dari keadaan saat ini, dihitung dengan fungsi **objective_func()**.

```

cube::errInfo neighbor;
cube::errInfo targetVal;
int currentErr = cube::objective_func();

```

Selanjutnya, dilakukan pencarian terhadap seluruh kemungkinan tetangga yang dapat diakses melalui fungsi **swap_cube**. Jika *error* dari tetangga yang baru lebih kecil atau sama dengan *error* saat ini, maka tetangga baru tersebut disimpan sebagai kandidat yang mungkin untuk langkah berikutnya.

```

for (int i = 0; i < cube::N; i++) {
    for (int j = 0; j < cube::N; j++) {
        for (int k = 0; k < cube::N; k++) {
            cube::errInfo temp = cube::swap_cube(i, j, k);
            if (neighbor.error > temp.error || (neighbor.error ==
currentErr && temp.error == currentErr)) {
                neighbor.error = temp.error;
                neighbor.x = temp.x;
                neighbor.y = temp.y;
                neighbor.z = temp.z;
                targetVal.x = i;
                targetVal.y = j;
                targetVal.z = k;
            }
        }
    }
}

```

```
}
```

Jika *error* pada tetangga sama dengan *error* saat ini, akan dilakukan “sideways move”, dan akan dihitung sudah berapa kali melakukan *sideways move*.

```
if (neighbor.error == currentErr) {
    sideways_moves++;
}
```

Jika *error* tetangga lebih kecil atau sama dengan *error* saat ini, maka pertukaran dilakukan untuk beralih ke keadaan yang lebih optimal atau setara.

```
if (neighbor.error <= currentErr) {
    cube::swap(targetVal.x, targetVal.y, targetVal.z, neighbor.x,
    neighbor.y, neighbor.z);
    currentErr = neighbor.error;
}
```

Proses pencarian dihentikan ketika program sudah mencapai local/global optima atau ketika batas maksimum *sideways move* tercapai yang pada program ini diset pada 100.

```
if (neighbor.error > currentErr || sideways_moves >=
max_sideways_moves) {
    if (sideways_moves >= max_sideways_moves) {
        std::cout << "===== Pencarian Sideways Move dihentikan karena
mencapai Sideways Move Maksimum====" << std::endl;
    } else {
        std::cout << "===== Pencarian Sideways dihentikan karena
mencapai local/global optima ====" << std::endl;
        std::cout << "==Algoritma diterminasi pada iterasi sideways
move ke: " << sideways_moves << "==" << std::endl;
    }
    return;
}
```

2.3 Random Restart Hill-Climbing

Pada metode ini, pencarian akan dilakukan berulang dengan melakukan *restart* pada *cube*. Ketika pencarian sudah sampai pada *local maxima*, algoritma akan mencoba mencari lagi dengan mengacak ulang *cube* dan mengecek *error*-nya sebagai *current state*. Maka dari itu, jumlah *restart* akan dibatasi untuk mencegah pencarian tanpa kemajuan.

Metode ini juga mengurangi kemungkinan terjebak di *local maxima* karena mengacak ulang *cube*. Semakin banyak *restart* yang dilakukan, semakin tinggi kemungkinan algoritma mencapai keadaan *global maxima*. Oleh karena itu, akan lebih efisien jika jumlah *restart* dibatasi dengan perhitungan tertentu.

Implementasi kode adalah sebagai berikut.

```
#include "cube.hpp"
#include <random_restart.hpp>
#include <unordered_set>
```

```

int random_restart::numOfRestart = 0;

void random_restart::setRestartNum(int n) {
random_restart::numOfRestart = n; }

void random_restart::hill_climbing() {

    int count = 0;
    if (std::remove("randomrestart.txt") == 0) {
        std::cout << "randomrestart.txt was deleted successfully.\n";
    } else {
        std::cout << "randomrestart.txt did not exist or couldn't be
deleted.\n";
    }

    std::ofstream result("randomrestart.txt", std::ios::app);
    if (!result) {
        std::cerr << "Error opening randomrestart.txt for writing." <<
std::endl;
        return;
    }

    if (std::remove("swap.txt") == 0) {
        std::cout << "swap.txt was deleted successfully.\n";
    } else {
        std::cout << "swap.txt did not exist or couldn't be deleted.\n";
    }

    // Open swap.txt in append mode for writing scores
    std::ofstream file("swap.txt", std::ios::app);
    if (!file) {
        std::cerr << "Error opening swap.txt for writing." << std::endl;
        return;
    }
    // Set restart to 2
    setRestartNum(2);

    int bestCube[cube::N][cube::N][cube::N];
    int bestCubeErr = 9999;
    int noteScore[numOfRestart];

    for (int r = 0; r < numOfRestart; r++) {

        while (true) {
            cube::errInfo neighbor;
            cube::errInfo targetVal;

            int currentErr = cube::objective_func();
            count++;

            if (!file) {
                std::cerr << "Error opening file." << std::endl;
                return;
            }

            file << currentErr << ",";
            // Loop trough all elements until finish condition satisfied
            for (int i = 0; i < cube::N; i++) {
                for (int j = 0; j < cube::N; j++) {
                    for (int k = 0; k < cube::N; k++) {
                        // For each elements
                        // Count the error after swapping, then find the smallest
                    }
                }
            }
        }
    }
}

```

```

one
    cube::errInfo temp = cube::swap_cube(i, j, k);
    if (neighbor.error > temp.error) {
        neighbor.error = temp.error;
        neighbor.x = temp.x;
        neighbor.y = temp.y;
        neighbor.z = temp.z;

        // Which Element to be swapped
        targetVal.x = i;
        targetVal.y = j;
        targetVal.z = k;
    }
}
}

if (neighbor.error < currentErr) {
    // std::cout << "Swapped" << std::endl;
    cube::swap(targetVal.x, targetVal.y, targetVal.z, neighbor.x,
               neighbor.y, neighbor.z);
    // std::cout << "Current Err : " << neighbor.error <<
std::endl;
}

if (currentErr <= neighbor.error) {
    std::cout << "====Hill Climbing Finished ====" << std::endl;
    break;
}
}

file.close();

// Read the content from swap.txt and prepend a new line in
// randomrestart.txt
std::ifstream read_file("swap.txt");
if (!read_file) {
    std::cerr << "Error opening swap.txt for reading." <<
std::endl;
    return;
}

std::stringstream buffer;
buffer << read_file.rdbuf();

std::cout << "Hello : " << buffer.str() << std::endl;
read_file.close();

result << count << std::endl;
result << buffer.str();
result.close();

// Find best Cube
noteScore[r] = cube::objective_func();
if (bestCubeErr > noteScore[r]) {
    bestCubeErr = noteScore[r];
    cube::copyCube(cube::cube, bestCube);
}
cube::restart_cube();
}

cube::copyCube(bestCube, cube::cube);

```

Pertama, dilakukan penghapusan file teks yang mungkin sudah ada dari operasi sebelumnya agar memastikan program dimulai dengan lingkungan yang bersih.

```
if (std::remove("randomrestart.txt") == 0) {
    std::cout << "randomrestart.txt was deleted successfully.\n";
} else {
    std::cout << "randomrestart.txt did not exist or couldn't be
deleted.\n";
}
std::ofstream result("randomrestart.txt", std::ios::app);
```

Selanjutnya, fungsi `setRestartNum()` mengatur jumlah restart yang dilakukan dalam optimasi. Kode ini juga membuka file `swap.txt` untuk mencatat skor dari setiap operasi.

```
setRestartNum(2);
std::ofstream file("swap.txt", std::ios::app);
```

Selanjutnya, akan dilakukan loop sejumlah kali (berdasarkan `numOfRestart`). Dalam setiap loop, kode akan mencoba mengubah posisi elemen dalam kubus (`cube::swap_cube`) untuk mendapatkan konfigurasi dengan kesalahan/error yang lebih rendah yang berarti lebih optimal atau lebih baik.

```
for (int r = 0; r < numOfRestart; r++) {
    while (true) {
        cube::errInfo neighbor;
        int currentErr = cube::objective_func();
        ...
        if (neighbor.error < currentErr) {
            cube::swap(targetVal.x, targetVal.y, targetVal.z, neighbor.x,
neighbo.y, neighbor.z);
        }
    }
}
```

Setelah setiap iterasi, algoritma mencatat kesalahan/error saat ini ke dalam file `swap.txt`. Jika tidak ada peningkatan lagi (tidak ada kesalahan yang lebih kecil yang ditemukan), loop akan berhenti dan program mencatat hasilnya ke `randomrestart.txt`.

```
file << currentErr << ";";
if (currentErr <= neighbor.error) {
    std::cout << "====Hill Climbing Finished ===" << std::endl;
    break;
}
```

Setelah melakukan semua restart, algoritma ini membandingkan semua skor yang diperoleh dan menyimpan konfigurasi terbaik dari kubus tersebut. Ini dilakukan dengan membandingkan skor dari setiap restart dan menyimpan yang terbaik.

```
noteScore[r] = cube::objective_func();
if (bestCubeErr > noteScore[r]) {
    bestCubeErr = noteScore[r];
    cube::copyCube(cube::cube, bestCube);
}
cube::restart_cube();
```

Pada akhir semua iterasi, konfigurasi terbaik disalin kembali ke kubus utama untuk digunakan selanjutnya atau sebagai hasil akhir.

```
cube::copyCube(bestCube, cube::cube);
```

2.4 Stochastic Hill-Climbing

Stochastic Hill-Climbing merupakan salah satu bentuk varian dari metode Hill Climbing dimana varian ini bekerja dengan melakukan pengambilan nilai secara acak sebagai *neighbor* dari *current state* berdasarkan suatu probabilitas dan *neighbor* ini akan dibandingkan dengan *current value* saat ini, apabila nilai *neighbor* ini lebih besar dari *current value* saat ini, maka nilai *neighbor* tersebut yang akan menjadi *current value* sekarang. Varian ini akan melakukan terminasi terhadap program ketika sudah mencapai suatu maksimum iterasi (nmax).

Apabila dibandingkan dengan Hill Climbing Steepest Ascent dan Hill Climbing Sideway Moves, Stochastic Hill Climbing merupakan algoritma *local search* yang paling cepat karena algoritma ini tidak melakukan pengecekan ke segala kemungkinan *neighbor* Magic Cube. Pengambilan *neighbor* yang dilakukan secara random juga menjadi cara yang dilakukan oleh algoritma ini untuk lepas dari *local maxima*.

Implementasi kode adalah sebagai berikut :

```
#include "cube.hpp"
#include <cstdlib>
#include <stochastic.hpp>

int stochastic::max_iteration = 5000;
void stochastic::setMaxIter(int n) { max_iteration = n; }

// Randomizer works by taking some 10 random swap and choose best
void stochastic::random_swap() {

    int count = 0;
    // Seed the random number generator with the current time
    cube::errInfo bestData;
    bestData.error = cube::objective_func();
    bool improved = false;
    cube::errInfo target;
    for (int r = 0; r < 100; r++) {
        // Generate a random number between 0 and 124
        int i_1 = std::rand() % 5; // 125 gives a range of 0-124
        int j_1 = std::rand() % 5;
        int k_1 = std::rand() % 5;

        int i_2 = std::rand() % 5;
        int j_2 = std::rand() % 5;
        int k_2 = std::rand() % 5;

        cube::swap(i_1, j_1, k_1, i_2, j_2, k_2);
        int currentErr = cube::objective_func();
        if (bestData.error > currentErr) {
            bestData.error = currentErr;
```

```

        bestData.x = i_1;
        bestData.y = j_1;
        bestData.z = k_1;

        target.x = i_2;
        target.y = j_2;
        target.z = k_2;
        improved = true;
    }
    cube::swap(i_1, j_1, k_1, i_2, j_2, k_2);
}

// std::cout << bestData.x << " " << bestData.y << " " <<
bestData.z << " "
//           << target.x << " " << target.y << " " << target.z <<
std::endl;
if (improved)
    cube::swap(bestData.x, bestData.y, bestData.z, target.x,
target.y,
            target.z);
}
void stochastic::hill_climbing() {
    setMaxIter(10000);
    int count = 0;

    if (std::remove("stochastic.txt") == 0) {
        std::cout << "stochastic.txt was deleted successfully.\n";
    } else {
        std::cout << "stochastic.txt did not exist or couldn't be
deleted.\n";
    }

    std::ofstream result("stochastic.txt", std::ios::app);
    if (!result) {
        std::cerr << "Error opening stochastic.txt for writing." <<
std::endl;
        return;
    }

    if (std::remove("swap.txt") == 0) {
        std::cout << "swap.txt was deleted successfully.\n";
    } else {
        std::cout << "swap.txt did not exist or couldn't be deleted.\n";
    }

    // Open swap.txt in append mode for writing scores
    std::ofstream file("swap.txt", std::ios::app);
    if (!file) {
        std::cerr << "Error opening swap.txt for writing." << std::endl;
        return;
    }
    for (int i = 0; i < max_iteration; i++) {
        int currentErr = cube::objective_func();
        random_swap();
        if (!file) {
            std::cerr << "Error opening file." << std::endl;
            return;
        }

        file << currentErr << ";";
    }
    std::cout << "Final Error : " << cube::objective_func() << "\n"
           << "===== Stochastic Hill Climbing Done =====" <<

```

```

std::endl;

file.close();

// Read the content from swap.txt and prepend a new line in
// steepAscent.txt
std::ifstream read_file("swap.txt");
if (!read_file) {
    std::cerr << "Error opening swap.txt for reading." << std::endl;
    return;
}

std::stringstream buffer;
buffer << read_file.rdbuf();

std::cout << "Hello : " << buffer.str() << std::endl;
read_file.close();

result << count << std::endl;
result << buffer.str();
result.close();
return;
}

```

Adapun penjelasan terkait implementasi kode di atas adalah sebagai berikut.

Pada metode ini, algoritma yang sudah dibuat pertama akan dideklarasi terlebih dahulu jumlah iterasi atau *looping* maksimum yang ingin dilakukan, kelompok mendeklarasi jumlah iterasi maksimum sebesar 10,000 kali. Setelah menentukan jumlah iterasi, program akan melaksanakan fungsi *random swap* sesuai dengan jumlah iterasi yang telah ditetapkan di awal.

```

int stochastic::max_iteration = 5000;
void stochastic::setMaxIter(int n) { max_iteration = n; }

```

```

void stochastic::hill_climbing() {
    setMaxIter(10000);
    for (int i = 0; i < max_iteration; i++) {
        random_swap();
    }
    std::cout << "Final Error : " << cube::objective_func() << "\n"
           << "===== Stochastic Hill Climbing Done =====" <<
    std::endl;
}

```

Output dari algoritma Stochastic Hill Climbing berupa nilai *final error* dari *objective function cube* tersebut.

```

std::cout << "Final Error : " << cube::objective_func() << "\n"
       << "===== Stochastic Hill Climbing Done =====" <<
std::endl;
}

```

Random swap ini bekerja dengan melakukan operasi *swap* pada *current state cube*. *Cube* yang telah diswap ini yang akan menjadi *neighbor*. Jika *neighbor* yang dipilih secara acak tersebut tidak menghasilkan jumlah *error* yang lebih minimum dari *current state*, maka akan dipilih kembali *neighbor* secara acak (melakukan *swap*).

```

void stochastic::random_swap() {

    int count = 0;
    // Seed the random number generator with the current time
    cube::errInfo bestData;
    bestData.error = cube::objective_func();
    bool improved = false;
    cube::errInfo target;
    for (int r = 0; r < 100; r++) {
        // Generate a random number between 0 and 124
        int i_1 = std::rand() % 5; // 125 gives a range of 0-124
        int j_1 = std::rand() % 5;
        int k_1 = std::rand() % 5;

        int i_2 = std::rand() % 5;
        int j_2 = std::rand() % 5;
        int k_2 = std::rand() % 5;

        cube::swap(i_1, j_1, k_1, i_2, j_2, k_2);
        int currentErr = cube::objective_func();
        if (bestData.error > currentErr) {
            bestData.error = currentErr;
            bestData.x = i_1;
            bestData.y = j_1;
            bestData.z = k_1;

            target.x = i_2;
            target.y = j_2;
            target.z = k_2;
            improved = true;
        }
        cube::swap(i_1, j_1, k_1, i_2, j_2, k_2);
    }

    // std::cout << bestData.x << " " << bestData.y << " " <<
    bestData.z << " "
    //           << target.x << " " << target.y << " " << target.z <<
    std::endl;
    if (improved)
        cube::swap(bestData.x, bestData.y, bestData.z, target.x,
target.y,
                    target.z);
}

```

Algoritma akan berhenti ketika menemukan kondisi *error* pada *neighbor* yang paling kecil atau ketika batas iterasi sudah mencapai batas maksimum.

2.5 Simulated Annealing

Simulated Annealing merupakan versi lain dari Stochastic Hill Climbing dimana beberapa pergerakan yang dilakukan secara *downhill* diperbolehkan. *Downhill* ini ditunjukkan dengan kemampuan metode ini untuk bergerak ke posisi *neighbor* yang lebih buruk dengan perhitungan probabilitas tertentu. Temperatur tinggi pada proses *annealing* menunjukkan akan tingkat toleransi *neighbor* yang lebih buruk pada algoritma *simulated annealing*. Jika temperatur sudah mencapai 0 atau batas bawah yang ditetapkan, maka itu merupakan solusi yang terbaik.

Implementasi kode adalah sebagai berikut :

```

#include "cube.hpp"
#include <cmath>
#include <simulated_annealing.hpp>

double simulated_annealing::alpha = 0.9;
void setAlpha(int n) { simulated_annealing::alpha = n; }
void simulated_annealing::work_func() {
    double T = 9999.0;

    int count = 0;

    if (std::remove("simulatedannealing.txt") == 0) {
        std::cout << "simulatedannealing.txt was deleted
successfully.\n";
    } else {
        std::cout
            << "simulatedannealing.txt did not exist or couldn't be
deleted.\n";
    }

    std::ofstream result("simulatedannealing.txt", std::ios::app);
    if (!result) {
        std::cerr << "Error opening simulatedannealing.txt for writing."
            << std::endl;
        return;
    }

    if (std::remove("swap.txt") == 0) {
        std::cout << "swap.txt was deleted successfully.\n";
    } else {
        std::cout << "swap.txt did not exist or couldn't be deleted.\n";
    }

    // Open swap.txt in append mode for writing scores
    std::ofstream file("swap.txt", std::ios::app);
    if (!file) {
        std::cerr << "Error opening swap.txt for writing." << std::endl;
        return;
    }
    while (true) {

        int currentErr = cube::objective_func();
        count++;
        if (T <= 0.00000001) {
            break;
        }

        if (!file) {
            std::cerr << "Error opening file." << std::endl;
            return;
        }

        file << currentErr << ",";

        cube::errInfo bestData;
        bool improved = false;
        cube::errInfo target;
        // Choose best candidate between all candidates
        for (int r = 0; r < 1000; r++) {
            // Generate a random number between 0 and 124
            int i_1 = std::rand() % 5; // 125 gives a range of 0-124
            int j_1 = std::rand() % 5;
            int k_1 = std::rand() % 5;
    }

```

```

        int i_2 = std::rand() % 5;
        int j_2 = std::rand() % 5;
        int k_2 = std::rand() % 5;

        cube::swap(i_1, j_1, k_1, i_2, j_2, k_2);
        int swapErr = cube::objective_func();
        if (bestData.error > swapErr) {
            bestData.error = swapErr;
            bestData.x = i_1;
            bestData.y = j_1;
            bestData.z = k_1;

            target.x = i_2;
            target.y = j_2;
            target.z = k_2;
        }
        cube::swap(i_1, j_1, k_1, i_2, j_2, k_2);
    }
    // Count Probs for choosing bad
    double ap = exp((bestData.error - currentErr) / T);
    T *= alpha;

    // std::cout << bestData.x << " " << bestData.y << " " <<
bestData.z << " "
    //           << target.x << " " << target.y << " " << target.z <<
std::endl;

    // std::cout << "Selected : " << currentErr << "/" <<
bestData.error
    //           << std::endl;
    if (bestData.error < currentErr) {
        cube::swap(bestData.x, bestData.y, bestData.z, target.x,
target.y,
                    target.z);
    } else if (ap > (double)rand() / RAND_MAX) {
        // std::cout << "Bad Selected : " << currentErr << "/" <<
bestData.error
        //<< std::endl;
        cube::swap(bestData.x, bestData.y, bestData.z, target.x,
target.y,
                    target.z);
    }
}
std::cout << " ===== Simulated Annealing Finished ===== " << "\n"
    << "Final Error : " << cube::objective_func() <<
std::endl;

file.close();

// Read the content from swap.txt and prepend a new line in
// steepAscent.txt
std::ifstream read_file("swap.txt");
if (!read_file) {
    std::cerr << "Error opening swap.txt for reading." << std::endl;
    return;
}

std::stringstream buffer;
buffer << read_file.rdbuf();

std::cout << "Hello : " << buffer.str() << std::endl;
read_file.close();

```

```

    result << count << std::endl;
    result << buffer.str();
    result.close();
    return;
}

```

Adapun penjelasan terkait implementasi kode di atas adalah sebagai berikut:
Pertama-tama, kelompok mendefinisikan *alpha*, yang merupakan nilai faktor penurunan suhu dengan nilai 0,9.

```

double simulated_annealing::alpha = 0.9;
void setAlpha(int n) { simulated_annealing::alpha = n; }
void simulated_annealing::work_func() {
    double T = 9999.0;
}

```

Algoritma dari Simulated Annealing dimulai dengan menyimpan nilai *error* awal dari kondisi *cube* saat ini ke dalam **currentErr**. T merupakan temperatur atau suhu awal yang ditetapkan ke 9999.0. Selama T masih bernilai lebih besar dari 0.00000001, maka algoritma akan terus mencari solusi yang terbaik.

```

while (true) {

    int currentErr = cube::objective_func();
    count++;
    if (T <= 0.00000001) {
        break;
    }
}

```

Bagian kode yang ada di bawah ini memampukan untuk melakukan operasi *swapping* pada *cube*. Proses *swapping* ini memiliki tujuan untuk mendapatkan solusi lain yang sekiranya bisa lebih baik dari solusi *cube* yang ada saat ini (*current state cube*) dengan nilai *error* yang lebih minim.

```

cube::errInfo bestData;
bool improved = false;
cube::errInfo target;
// Choose best candidate between all candidates
for (int r = 0; r < 1000; r++) {
    // Generate a random number between 0 and 124
    int i_1 = std::rand() % 5; // 125 gives a range of 0-124
    int j_1 = std::rand() % 5;
    int k_1 = std::rand() % 5;

    int i_2 = std::rand() % 5;
    int j_2 = std::rand() % 5;
    int k_2 = std::rand() % 5;

    cube::swap(i_1, j_1, k_1, i_2, j_2, k_2);
    int swapErr = cube::objective_func();

    if (swapErr < currentErr) {
        currentErr = swapErr;
        bestData = target;
        target = swapErr;
        improved = true;
    }
}

```

Kode ini melakukan pengecekan apakah solusi *swap* yang terbaru (*best data error*) lebih baik dibandingkan solusi saat ini (*current error*). Apabila solusi terbaru lebih

baik, maka *cube* solusi saat ini akan benar-benar dilakukan penukaran menjadi kondisi *cube* hasil *swap*. Di sisi lain apabila solusi *swap* yang terbaru (*best data error*) lebih buruk dibandingkan solusi saat ini (*current error*), masih diberikan kesempatan melalui rumus probabilitas tertentu (pada bagian kode ditulis dengan variabel *ap*). Apabila nilai probabilitas ini lebih besar dari random value, maka *cube* solusi saat ini akan benar-benar dilakukan penukaran menjadi kondisi *cube* hasil *swap*.

```
if (bestData.error < currentErr) {
    cube::swap(bestData.x, bestData.y, bestData.z, target.x,
target.y,
            target.z);
} else if (ap > (double)rand() / RAND_MAX) {
    // std::cout << "Bad Selected : " << currentErr << "/" <<
bestData.error
    //<< std::endl;
    cube::swap(bestData.x, bestData.y, bestData.z, target.x,
target.y,
            target.z);
}
}
```

Rumus dari probabilitas yang digunakan bisa dilihat di bawah ini. Selain itu, perlu diketahui juga bahwa setiap kali iterasi dijalankan, *T* atau temperatur akan dikalikan dengan nilai *alpha* yang bernilai 0.9.

```
// Count Probs for choosing bad
double ap = exp((bestData.error - currentErr) / T);
T *= alpha;
```

Output dari algoritma Simulated Annealing berupa nilai *final error* dari *objective function cube* tersebut.

```
std::cout << " ===== Simulated Annealing Finished ===== " << "\n"
        << "Final Error : " << cube::objective_func() <<
std::endl;
}
```

2.6 Genetic Algorithm

Genetic algorithm merupakan teknik dari *Local Search* yang mirip dengan persilangan pada makhluk hidup. Genetic algorithm pada umumnya menggunakan teknik *crossover* yang dimulai dengan melakukan seleksi *fitness* terlebih dahulu pada populasi *state* sebelum dilakukan penyilangan. Hasil persilangan antara 2 individu tersebut akan menghasilkan *child* yang merupakan *successor* baru yang ditambahkan ke populasi lama. Persilangan ini akan berhenti apabila terbentuk suatu individual *state* yang bagus atau mungkin dikarenakan pencarian sudah berlangsung terlalu lama. Selain persilangan pada dua individu, Genetic Algorithm juga memberikan probabilitas mutasi pada *child* hasil persilangan.

Implementasi kode adalah sebagai berikut :

```

#include "cube.hpp"
#include <algorithm>
#include <cstdlib>
#include <genetic_algorithm.hpp>
#include <iostream>
#include <numeric>
#include <random>
#include <set>
#include <vector>

int population_num = 10;
int N = 5;
double MUTATION_PROB = 0.20;

// FUNCTION FOR CROSSOVER
void genetic_algorithm::crossover(int
parent1[cube::N][cube::N][cube::N],
int
parent2[cube::N][cube::N][cube::N],
int
cube_target1[cube::N][cube::N][cube::N],
int allowedValues[], int
allowedSize) {
    std::vector<int> flatcube1;
    std::vector<int> flatcube2;
    std::vector<int> child;

    flatcube1 = cube::flatCube(parent1);
    flatcube2 = cube::flatCube(parent2);
    child = flatcube1;

    int size = flatcube1.size();
    int start = rand() % size;
    int end = rand() % size;

    if (start > end) {
        int temp = end;
        end = start;
        start = temp;
    }

    for (int i = start; i < end; ++i) {
        int parent2_value = flatcube2[i];
        int child_value = child[i];

        if (parent2_value != child_value) {
            auto it = std::find(child.begin(), child.end(), parent2_value);
            if (it != child.end()) {
                int index = std::distance(child.begin(), it);
                child[index] = child_value;
            }
            child[i] = parent2_value;
        }
    }
    int cube1[cube::N][cube::N][cube::N];
    // std::cout<<"unflattenCube"<<std::endl;
    cube::unflattenCube(child, cube1);
    applyMutation(cube1, allowedValues, allowedSize);
    // std::cout<<"copying cube"<<std::endl;
    cube::copyCube(cube_target1, cube1);
}

// Function to perform Cycle Crossover on a 2D array (one layer of

```

```

the 3D array)
void genetic_algorithm::cycleCrossover2D(int
parent1Layer[cube::N] [cube::N],
                                         int
parent2Layer[cube::N] [cube::N],
                                         int
offspringLayer[cube::N] [cube::N]) {
    bool visited[cube::N] [cube::N] = {false}; // To track visited
elements

    // Cycle Crossover for 2D array
    for (int start = 0; start < N * N; ++start) {
        // std::cout<<"Masuk loop"<<std::endl;
        int row = start / N;
        int col = start % N;

        if (!visited[row][col]) {
            int current = start;
            int cycleLimit = N * N;
            int cycleCount = 0;
            do {

                row = current / N;
                col = current % N;
                // std::cout<<"test"<<row<<col<<std::endl; //DEBUG
                // Copy element from parent1 to offspring
                offspringLayer[row][col] = parent1Layer[row][col];
                visited[row][col] = true;

                // Find the next element in the cycle (from parent2 to
parent1)
                int nextElem = parent2Layer[row][col];
                // std::cout<<"masuk"<<std::endl;
                // Search for the element in parent1
                bool found = false;
                for (int r = 0; r < N && !found; ++r) {
                    for (int c = 0; c < N && !found; ++c) {
                        if (parent1Layer[r][c] == nextElem) {
                            // std::cout<<r<<c<<N<<std::endl;
                            current = r * N + c;
                            found = true;
                            // std::cout<<"selesailoop"<<std::endl;
                        }
                    }
                }
            }

            } while (current != start); // Cycle ends when we reach the
start index
        }
    }
    std::cout << "finish2D" << std::endl;
}

// Function to apply mutation based on a probability
void genetic_algorithm::applyMutation(int
offspring[cube::N] [cube::N] [cube::N],
                                         int allowedValues[], int
allowedSize) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            for (int k = 0; k < N; ++k) {
                // Randomly mutate based on mutation probability
                if ((double)rand() / RAND_MAX < MUTATION_PROB) {

```

```

        // Select a random element from allowedValues (ensure no
        duplicates)
        int newVal;
        bool duplicate;
        do {
            newVal = allowedValues[rand() % allowedSize];
            duplicate = false;
            // Check for duplicates in the current layer
            for (int l = 0; l < N; ++l) {
                if (offspring[i][j][l] == newVal) {
                    duplicate = true;
                    break;
                }
            }
        } while (duplicate);

        offspring[i][j][k] = newVal; // Mutate the element
    }
}
}

// Cycle Crossover for 3D arrays with mutation
void genetic_algorithm::cycleCrossoverWithMutation(
    int parent1[cube::N][cube::N][cube::N],
    int parent2[cube::N][cube::N][cube::N],
    int offspring[cube::N][cube::N][cube::N], int allowedValues[],
    int allowedSize) {

    // Apply cycle crossover layer by layer (for each 2D array in the
    3D cube)
    for (int i = 0; i < N; ++i) {
        // std::cout<<"masuk"<<i<<std::endl;
        cycleCrossover2D(parent1[i], parent2[i], offspring[i]);
    }

    // Apply mutation after crossover
    applyMutation(offspring, allowedValues, allowedSize);
}

void genetic_algorithm::work_func() {

    if (std::remove("genetic.txt") == 0) {
        std::cout << "genetic.txt was deleted successfully.\n";
    } else {
        std::cout << "genetic.txt did not exist or couldn't be
        deleted.\n";
    }

    std::ofstream result("genetic.txt", std::ios::app);
    if (!result) {
        std::cerr << "Error opening genetic.txt for writing." <<
        std::endl;
        return;
    }

    if (std::remove("swap.txt") == 0) {
        std::cout << "swap.txt was deleted successfully.\n";
    } else {
        std::cout << "swap.txt did not exist or couldn't be deleted.\n";
    }
}

```

```

// Open swap.txt in append mode for writing scores
std::ofstream file("swap.txt", std::ios::app);
if (!file) {
    std::cerr << "Error opening swap.txt for writing." << std::endl;
    return;
}
std::set<int> new_population;
// int population_num = 10;

std::vector<genetic_algorithm::individual> population;

// Current generation
int generation = 0;

// Create randomized initial cube
for (int i = 0; i < population_num; i++) {
    individual new_cube;
    new_cube.fitness =
        110 - cube::objective_func(); // Lower objective function
value mean
                                            // better fitness so we inverse
    cube::copyCube(cube::cube, new_cube.cube);
    population.push_back(new_cube);
    cube::restart_cube();
}

// Start main looping
// Do roulette n times, cross over
for (int i = 0; i < 1000; i++) {
    // Sort by fitness in ascending order
    std::sort(population.begin(), population.end(),
              [] (const genetic_algorithm::individual &a,
                  const genetic_algorithm::individual &b) {
                  return a.fitness > b.fitness;
              });
    double probabilities[population_num]; // for relative probability
of each
                                            // individual
    // int all_score[population_num];

    double sum_score = 0;
    for (int i = 0; i < population_num; i++) {
        sum_score += population[i].fitness;
    }

    for (int i = 0; i < population_num; i++) {
        probabilities[i] = population[i].fitness / (sum_score);
    }

    double cummulative_prob = 0;
    for (int i = 0; i < population_num; i++) {
        cummulative_prob += probabilities[i];
        population[i].probabilities = cummulative_prob;
    }

    if (population[0].fitness <= 0) {
        break;
    }

    int allowedValues[N * N * N];
    for (int i = 0; i < N * N * N; ++i) {
        allowedValues[i] = i + 1;
    }
}

```

```

int allowedSize = N * N * N;

int firstparent = 0;
int secondparent = 0;
bool foundfirst;
bool foundsecond;

int max_obj = -99999;
for (int i = 0; i < population_num / 2; i++) {
    foundfirst = false;
    foundsecond = false;
    double r = ((double)rand() / (RAND_MAX)); // rand value (0,1)
    for (int j = 0; j < population_num; j++) {
        if (population[j].probabilities <= r &&
            !foundfirst) { // if found lowerbound
            firstparent = j;
            foundfirst = true;
            std::cout << "foundfirst" << j << std::endl; // DEBUG
        }
    }

    for (int j = 0; j < population_num; j++) {
        if (population[j].probabilities >= r &&
            !foundsecond) { // if found upperbound
            secondparent = j;
            std::cout << "foundsecond" << j << std::endl; // DEBUG
            foundsecond = true;
        }
    }
}

// Offspring array
int offspring1[cube::N][cube::N][cube::N] = {{{0}}}; // Initialize with 0s
// Perform cycle crossover with mutation
// cycleCrossoverWithMutation(population[firstparent].cube,
//                             population[secondparent].cube,
//                             offspring1, allowedValues,
allowedSize);

firstparent = 0;
secondparent = 0;
crossover(population[firstparent].cube,
population[secondparent].cube,
offspring1, allowedValues, allowedSize);
std::cout << "crossover finish" << std::endl; // DEBUG
int num1 = cube::objective_func(offspring1); // first cube
max_obj = std::max(num1, max_obj);

individual new_one1;
cube::copyCube(offspring1, new_one1.cube);
new_one1.fitness = 110 - num1; // relative fitness
population.push_back(new_one1);

// int num2 = cube::objective_func(offspring2); //second cube
// individual new_one2;
// new_one2.fitness = 110 - num2; //relative fitness
// cube::copyCube(offspring2, new_one2.cube);
// population.push_back(new_one2);
}

if (!file) {
    std::cerr << "Error opening file." << std::endl;
    return;
}

```

```

        file << max_obj << ";" ;

        std::cout << "Generation: " << generation << "\t" << std::endl;

        generation++;
        // Update population size
    }

individual smallestOne;
smallestOne.fitness = -9999;
// std::cout << "Current Err :" << cube::objective_func() <<
std::endl;
for (individual i : population) {
    if (i.fitness > smallestOne.fitness) {
        smallestOne = i;
    }
}

file.close();

// Read the content from swap.txt and prepend a new line in
// sideways.txt
std::ifstream read_file("swap.txt");
if (!read_file) {
    std::cerr << "Error opening swap.txt for reading." << std::endl;
    return;
}

std::stringstream buffer;
buffer << read_file.rdbuf();

std::cout << "Hello : " << buffer.str() << std::endl;
read_file.close();

result << 1000 << std::endl;
result << buffer.str();
result.close();
cube::copyCube(cube::cube, smallestOne(cube));
cube::displayCube();
return;

// std::cout << "New Err :" << smallestOne.fitness << std::endl;
}

```

Adapun penjelasan terkait implementasi kode di atas adalah sebagai berikut:

Algoritma Genetic Algorithm yang diimplementasikan terdiri dari beberapa tahapan penting yang mencakup *initialize population*, *fitness function & selection*, *crossover*, dan *mutation*. Pada tahap *initialize population*, berbagai kubus yang tersusun secara acak dibuat sebagai individu dalam populasi awal dan disimpan dalam objek **individual**. *Fitness* setiap kubus diperhitungkan dengan menggunakan fungsi **cube::objective_func()**. *Fitness* yang lebih tinggi biasanya menjadi solusi yang lebih baik.

Pada *fitness function & selection*, algoritma ini menerapkan seleksi dengan probabilitas relatif dimana individu yang memiliki *fitness* lebih tinggi akan lebih cenderung terpilih. Probabilitas relatif dihitung oleh setiap individu berdasarkan total

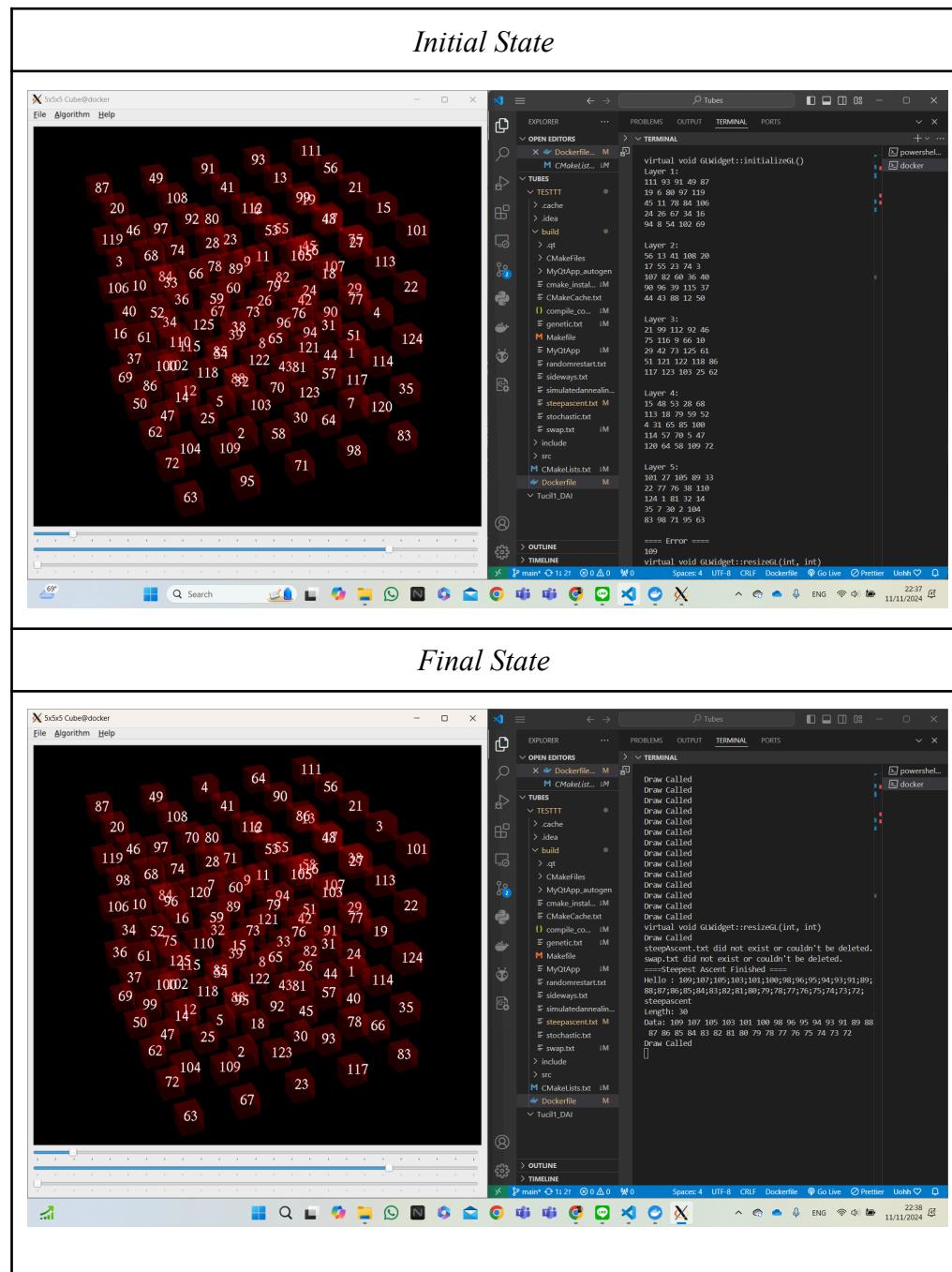
fitness populasi. Sementara, nilai *fitness* kumulatif juga perlu diperhitungkan untuk mencari rentang probabilitas seleksi setiap individu dalam populasi.

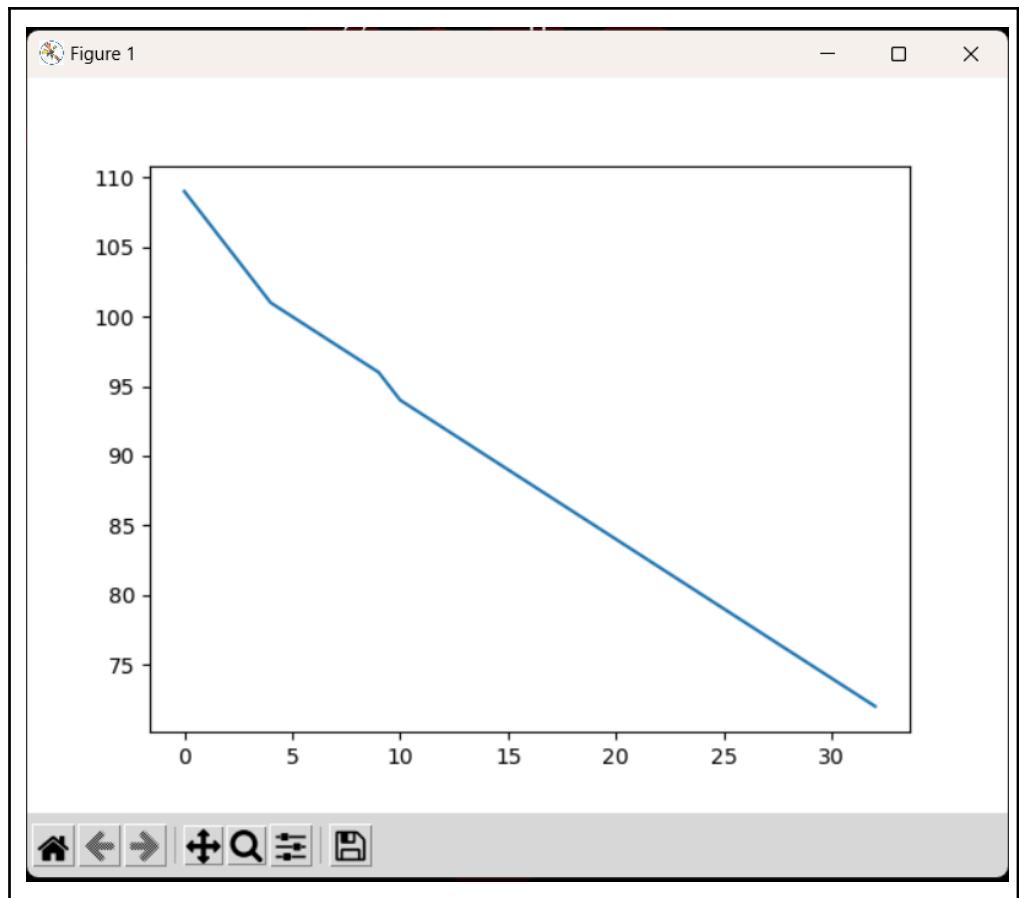
Crossover dilakukan dengan mengambil sebagian dari elemen *parent* pertama dan sebagian dari *parent* kedua dengan fungsi **`cycleCrossoverWithMutation`**. Setelah *crossover*, dilanjutkan dengan mutasi yang berdasarkan probabilitas **`mutation prob`**. Setelah semua tahapan dilalui, keturunan berhasil ditambahkan ke *population*. Dengan ditambahkannya ke dalam populasi, proses ini akan diulang berkali-kali hingga kondisi yang telah ditetapkan di kode terpenuhi.

3. Hasil Visualisasi Eksperimen

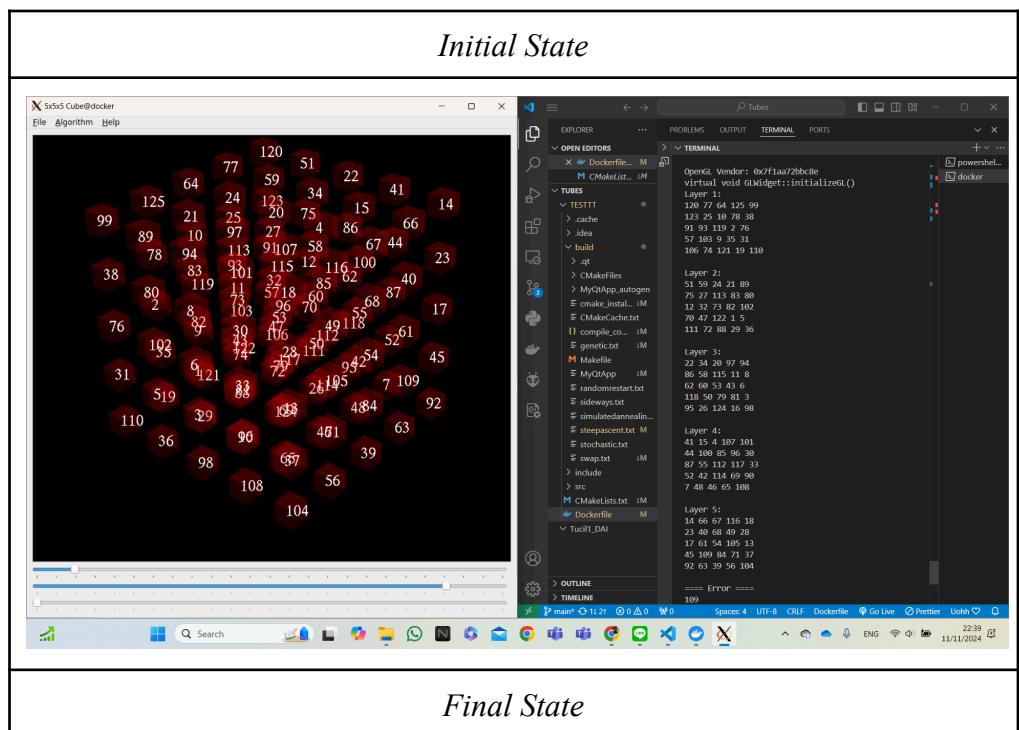
3.1 Steepest Ascent Hill-Climbing

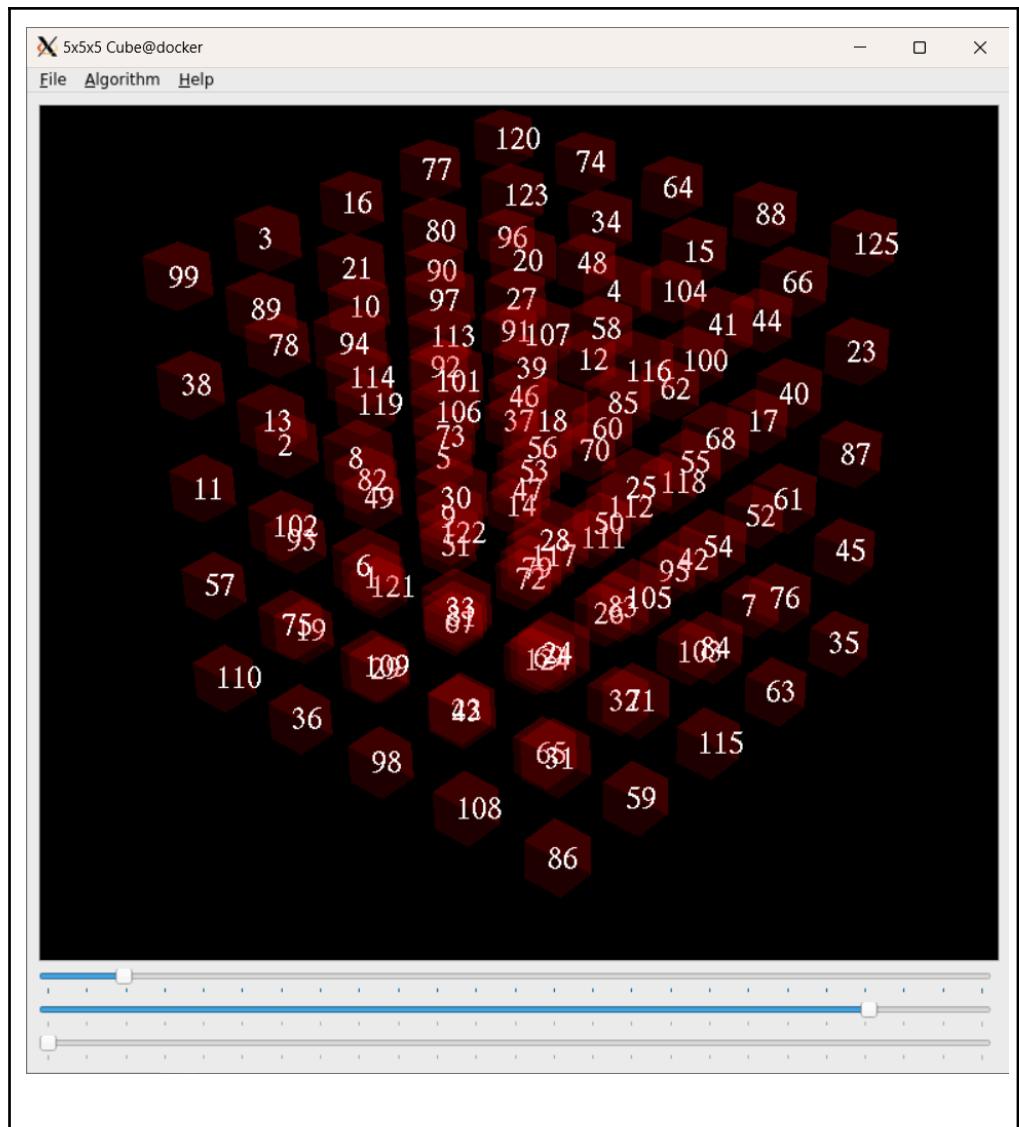
a. Percobaan 1

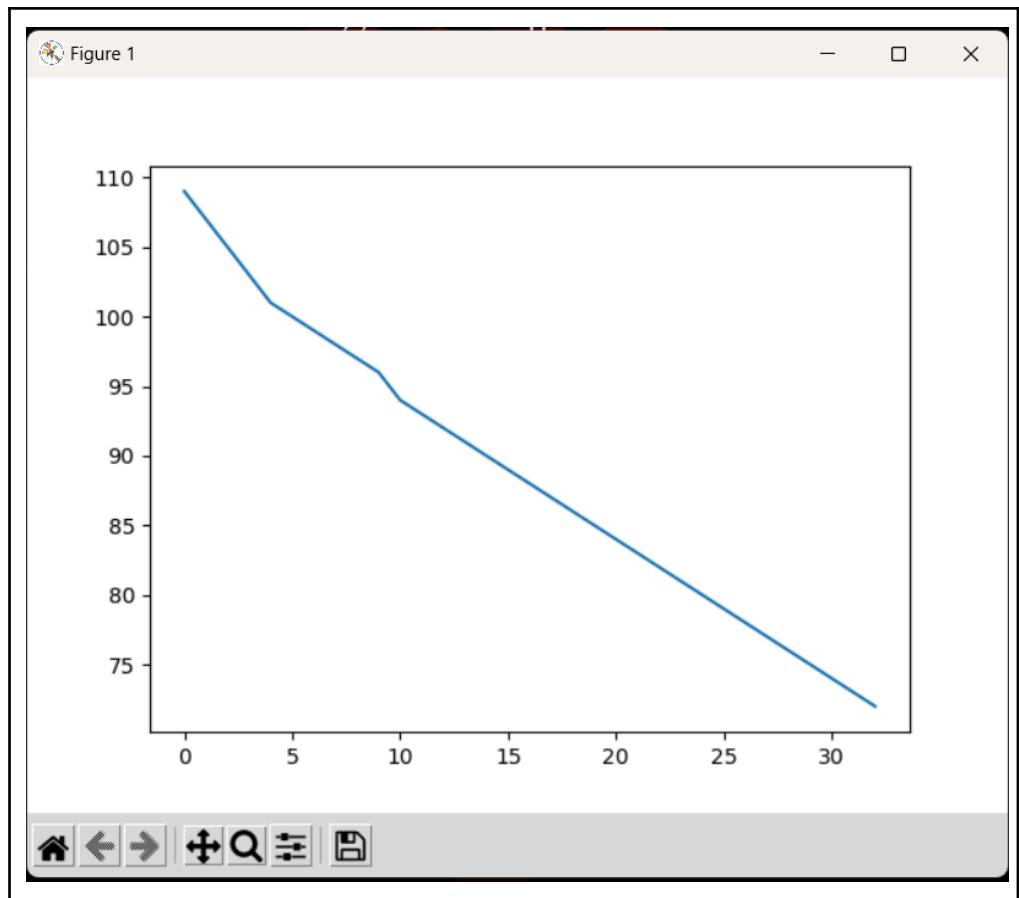




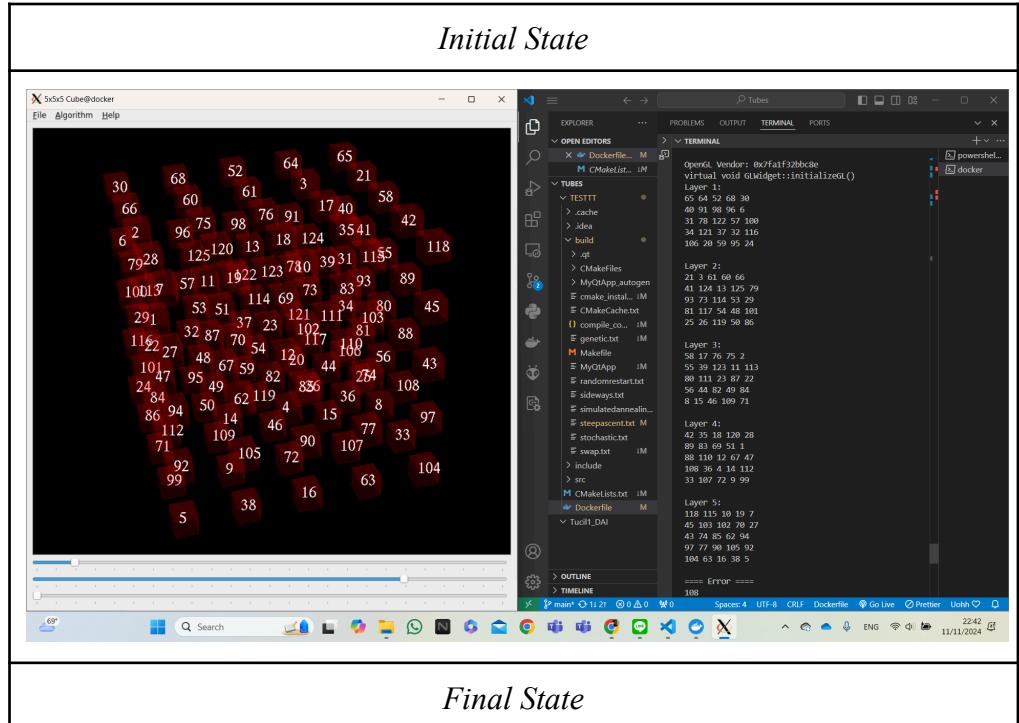
b. Percobaan 2

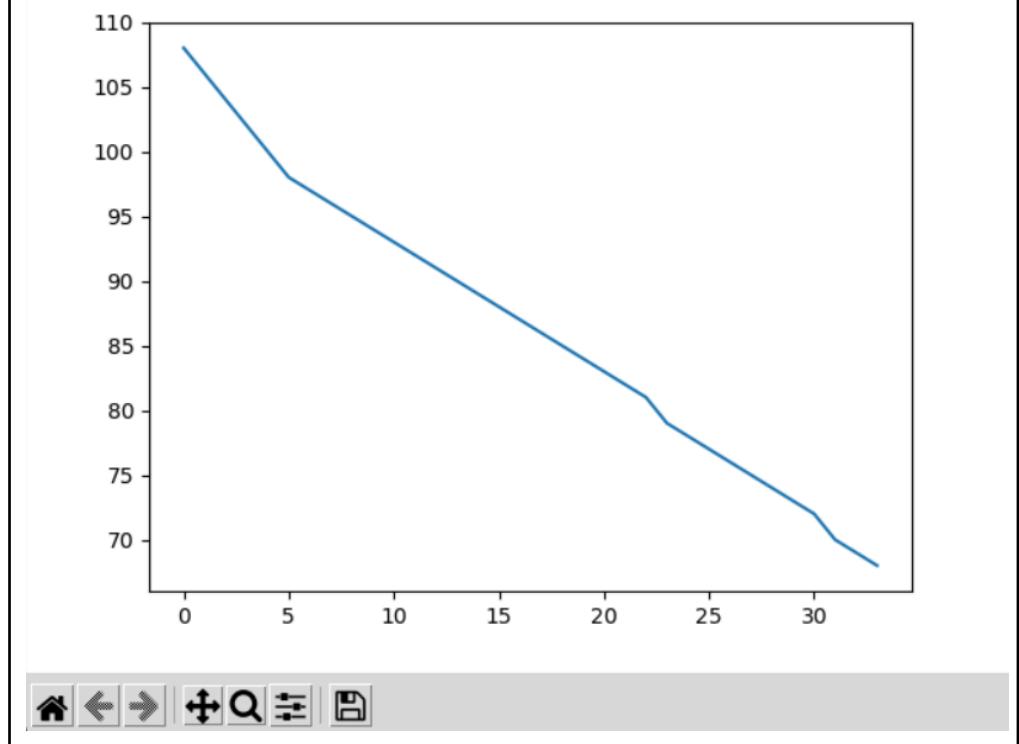
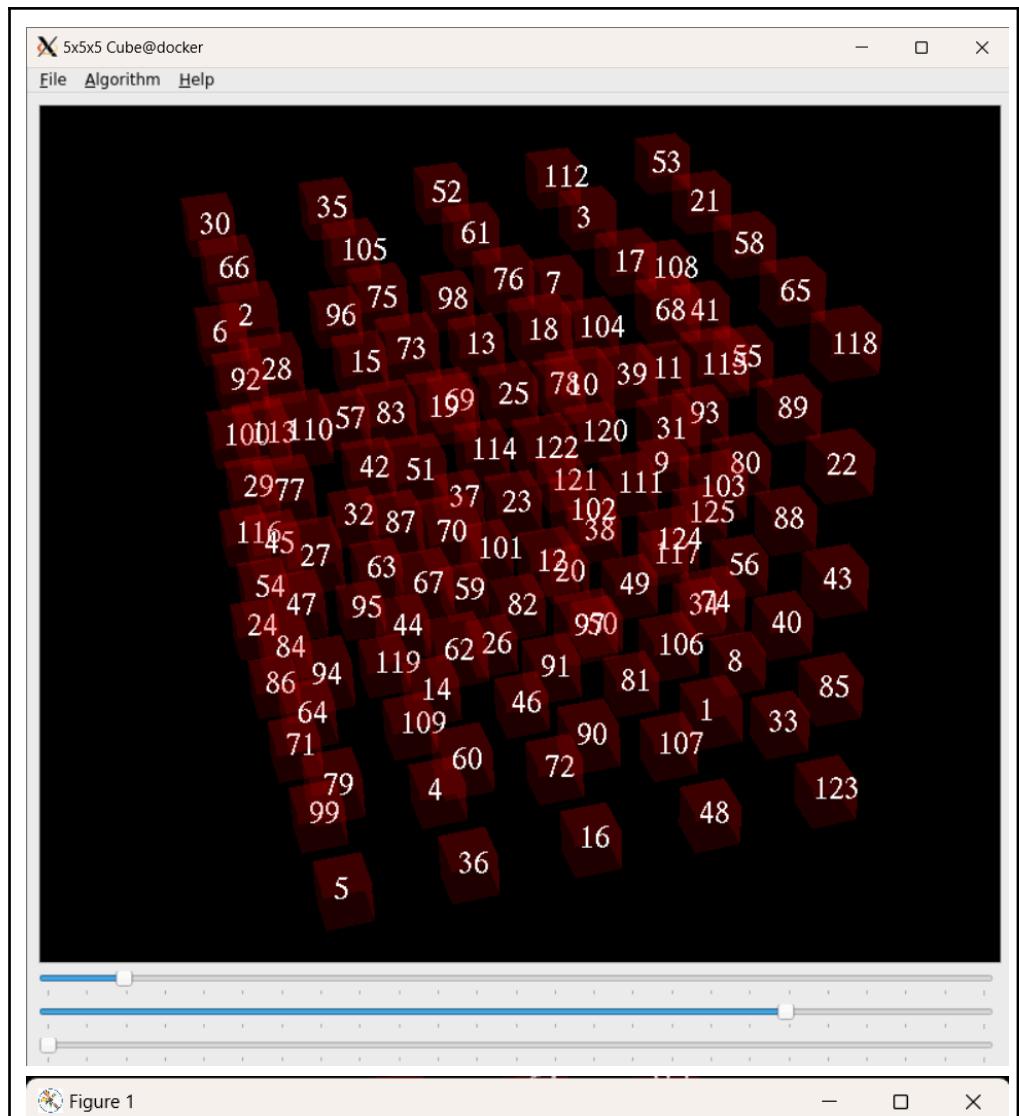






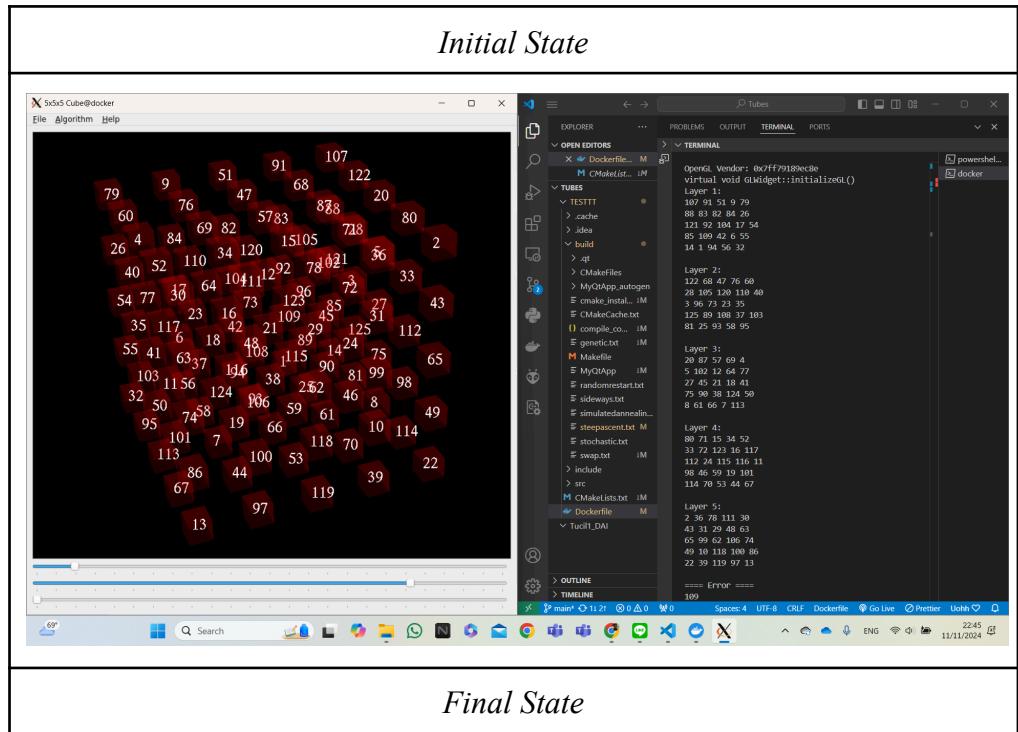
c. Percobaan 3

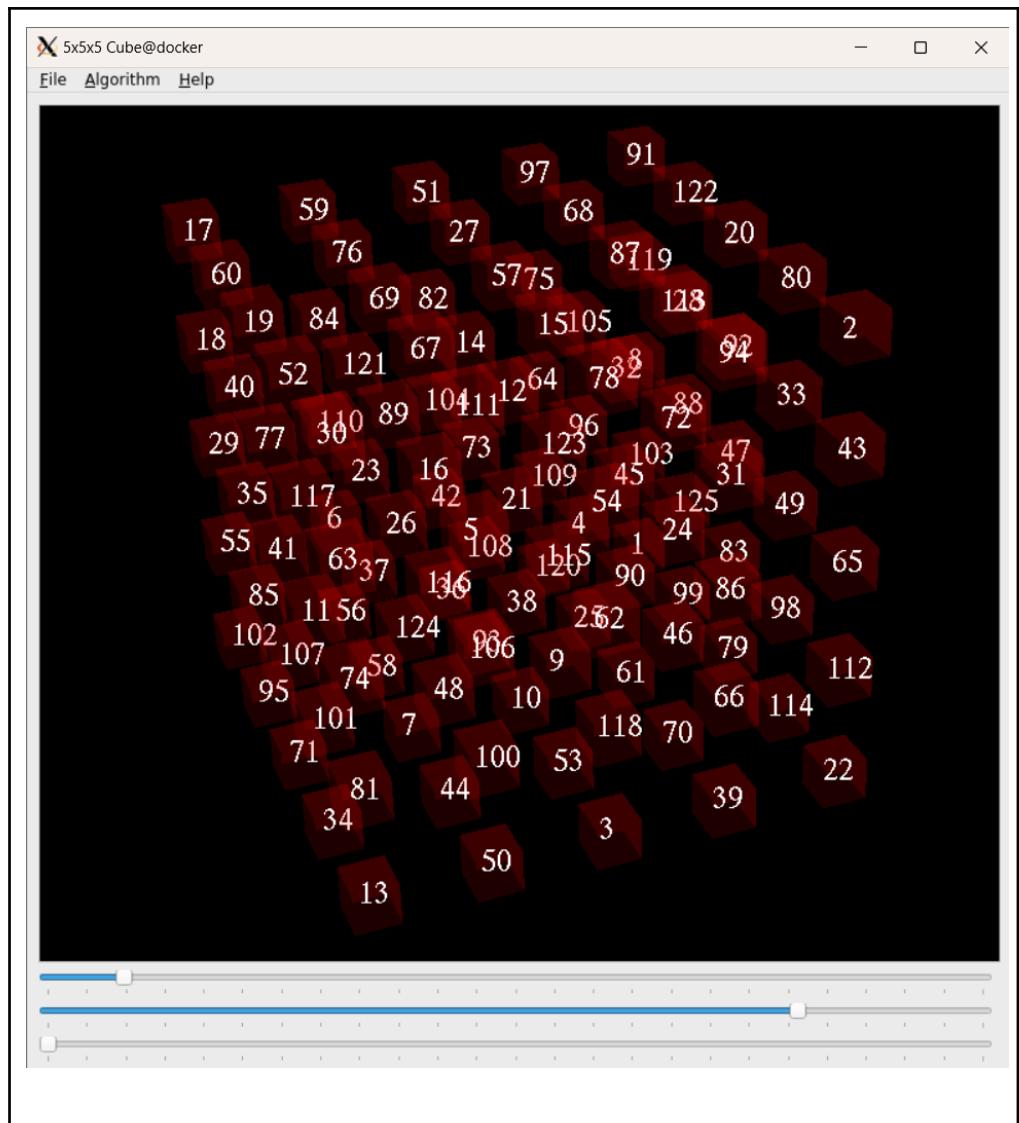


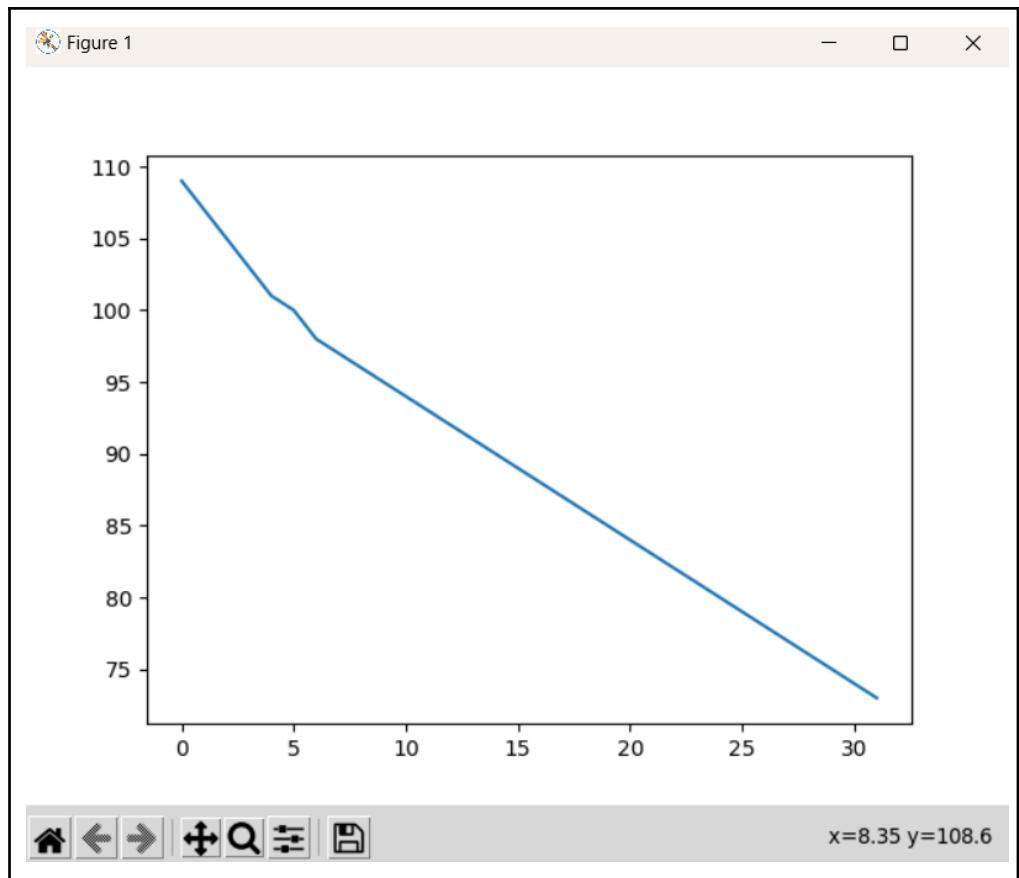


3.2 Hill-Climbing with Sideways Move

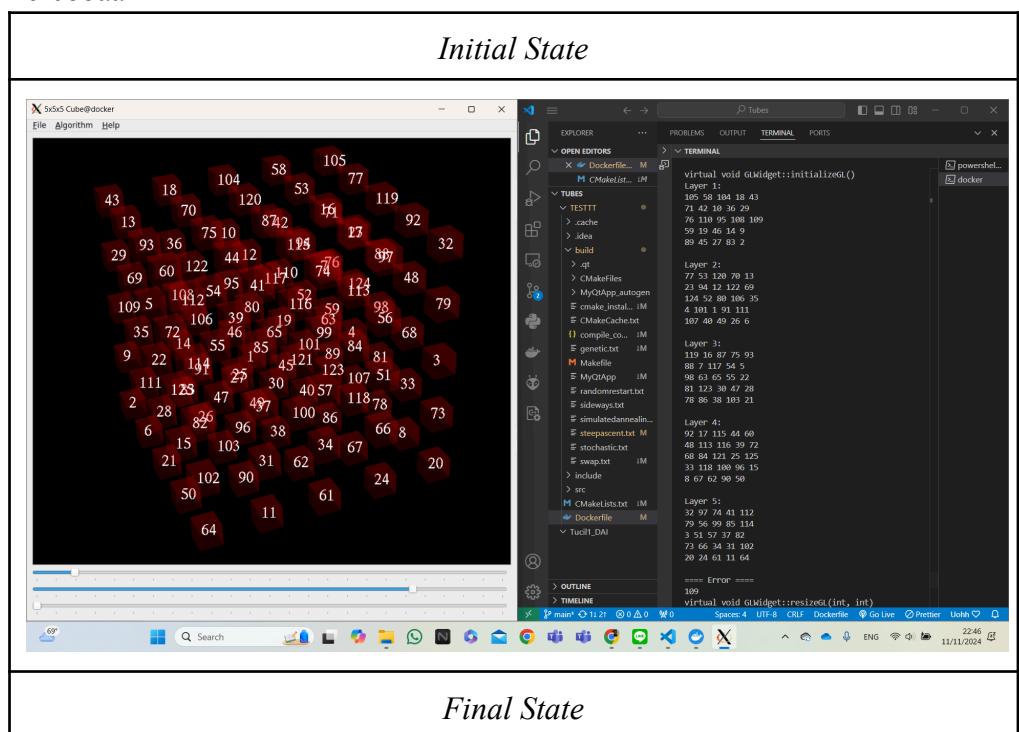
a. Percobaan 1

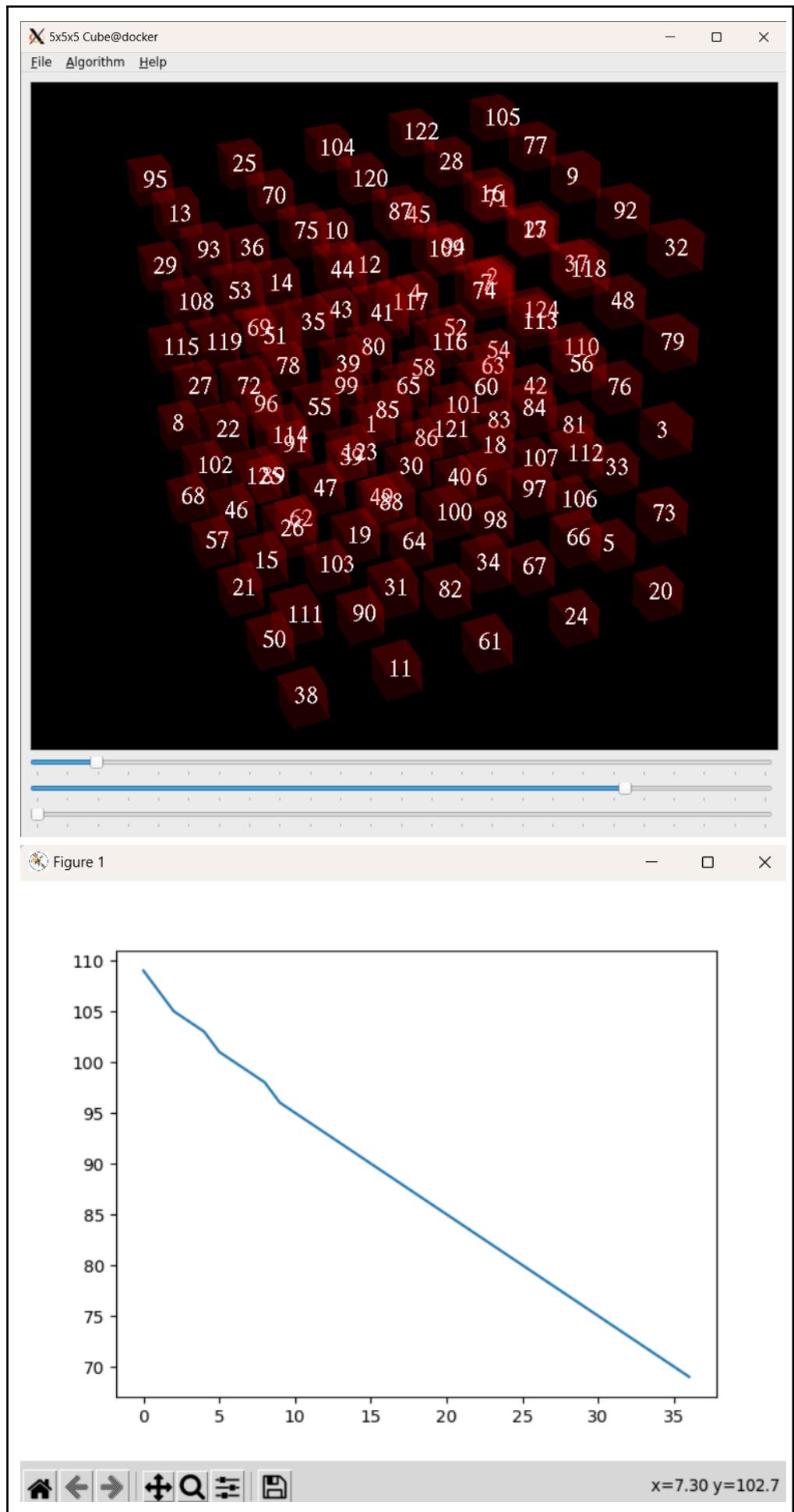




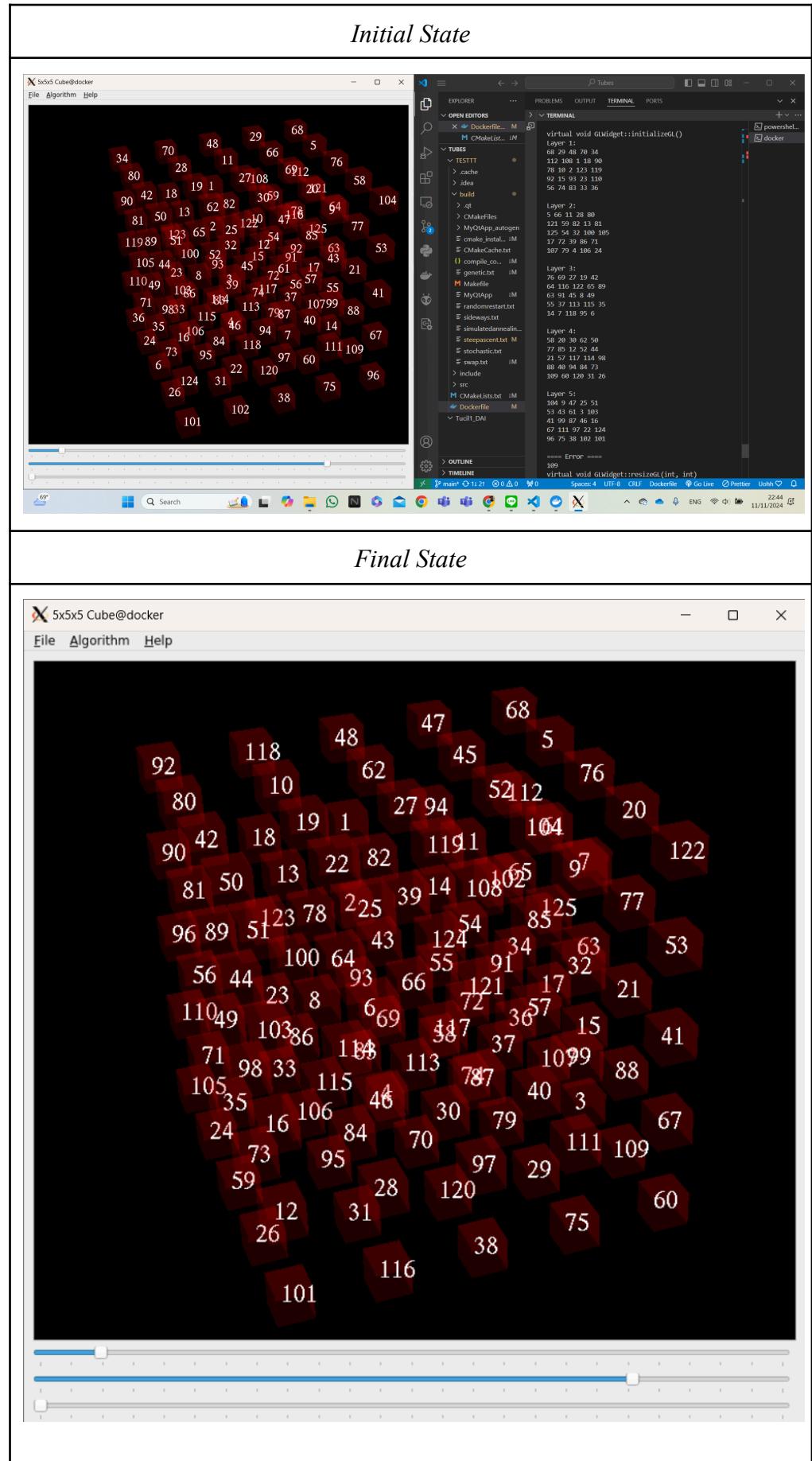


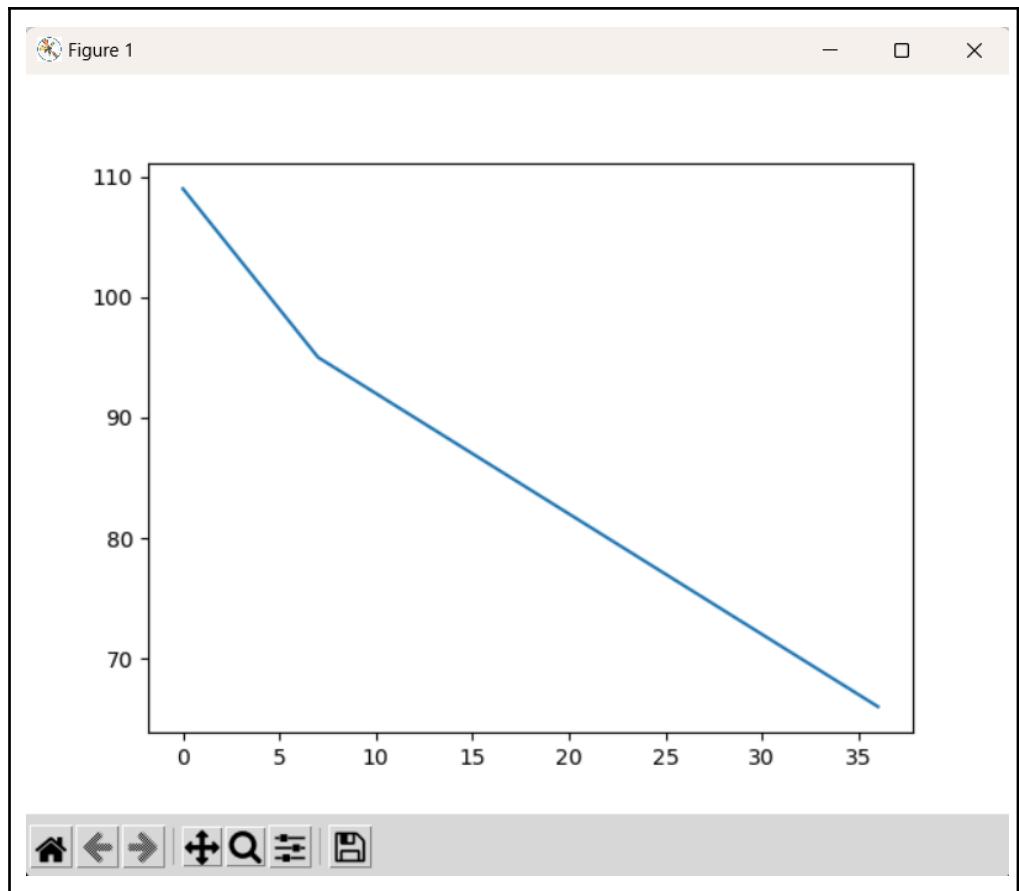
b. Percobaan 2





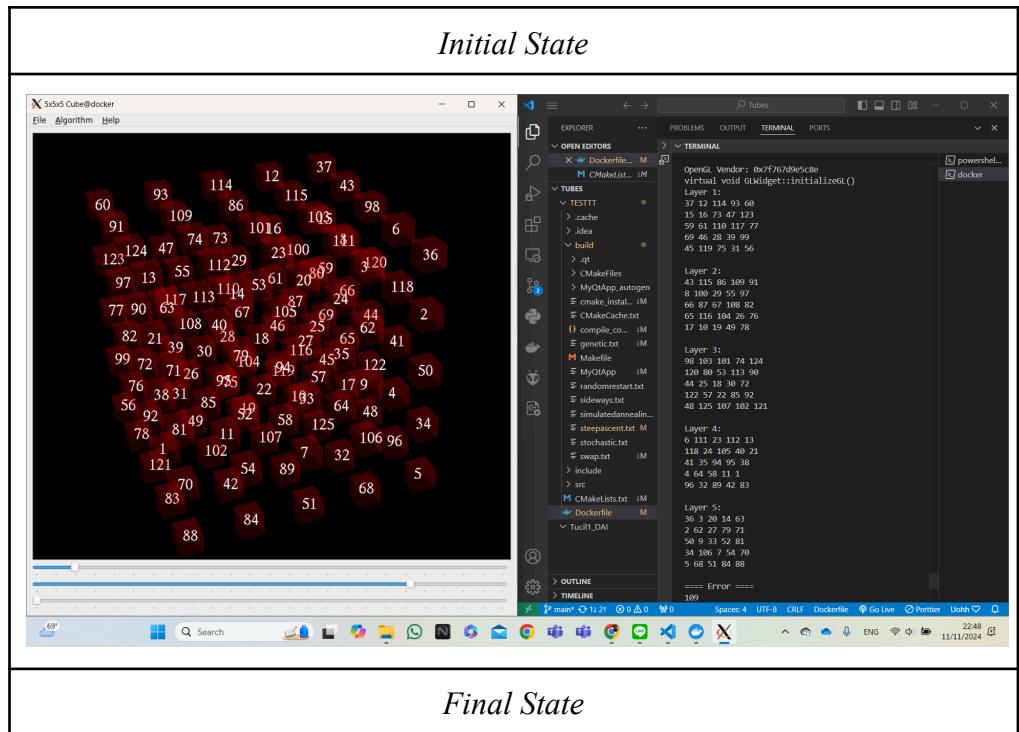
c. Percobaan 3

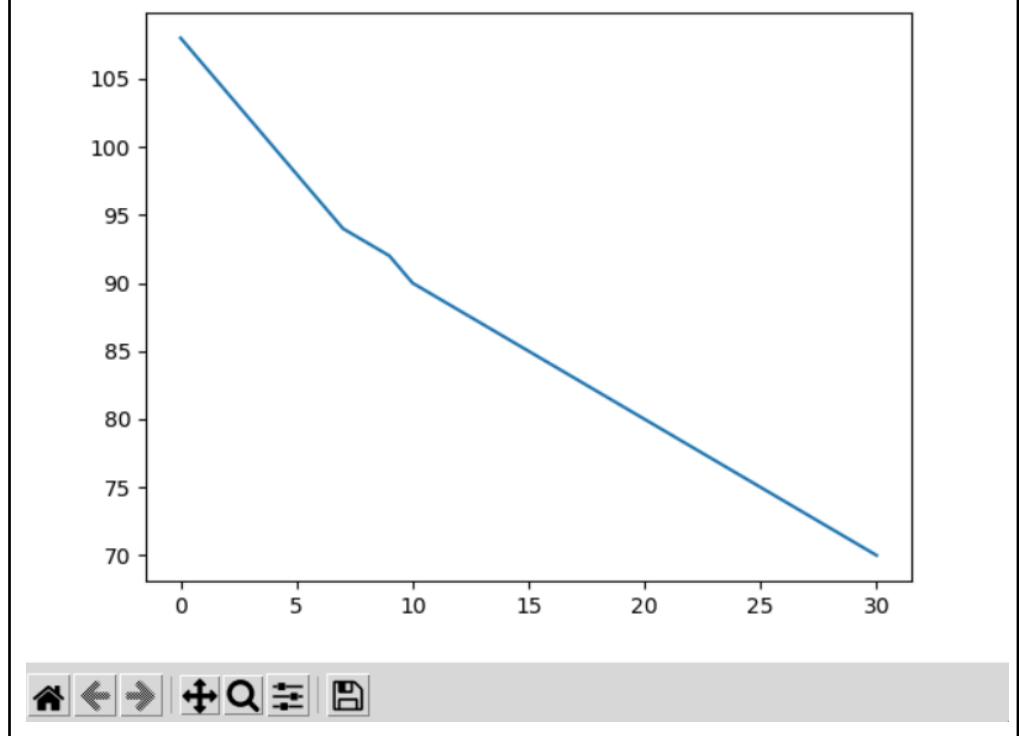
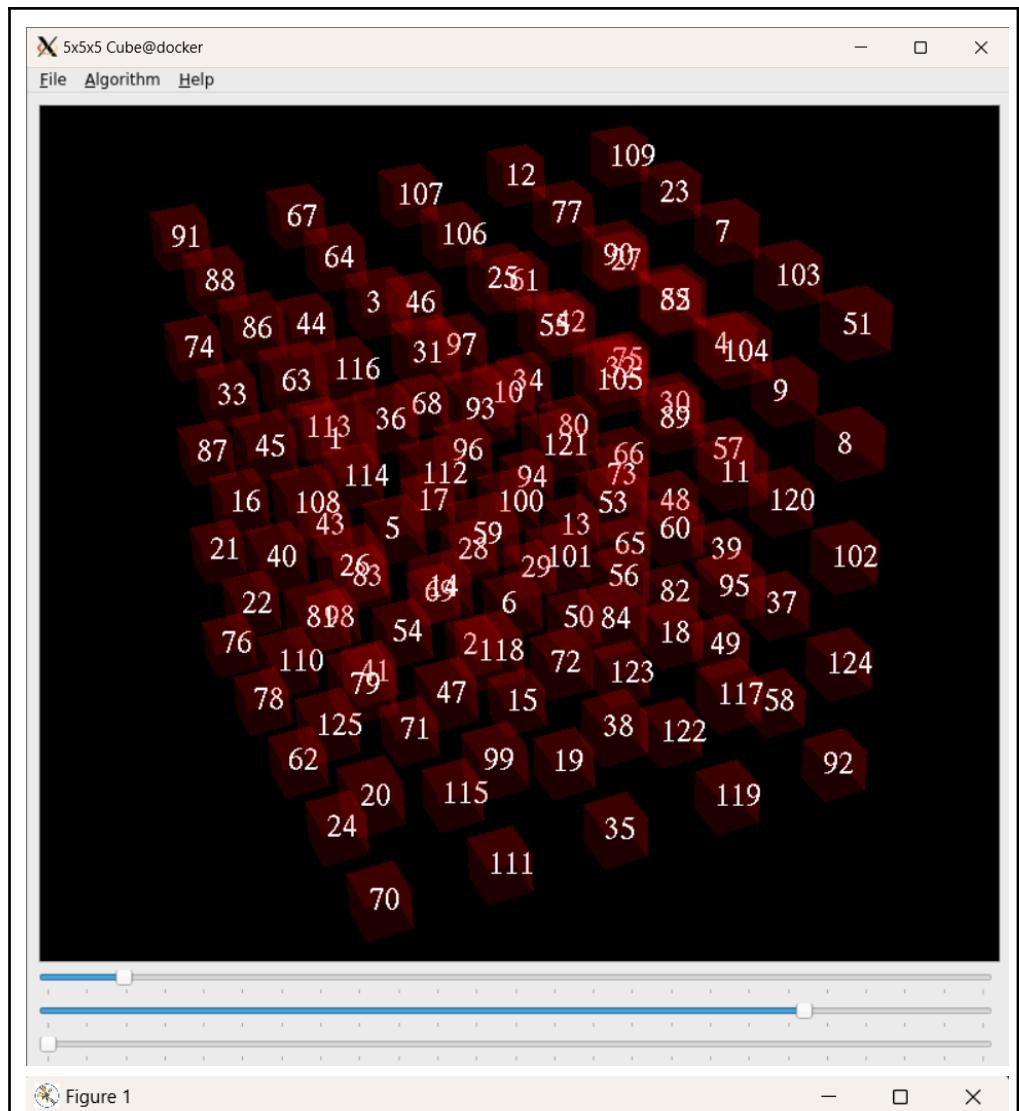




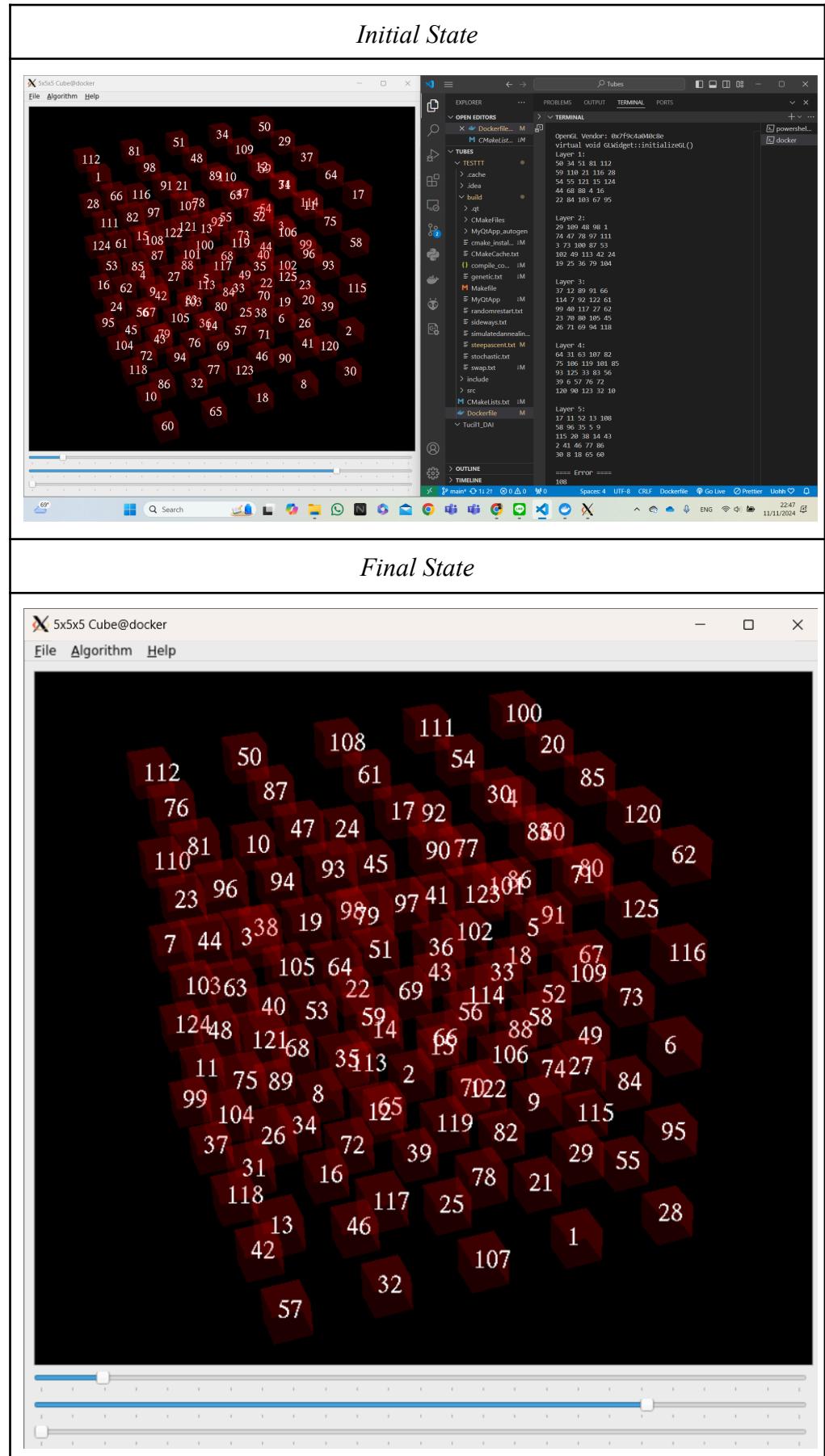
3.3 Random Restart Hill-Climbing

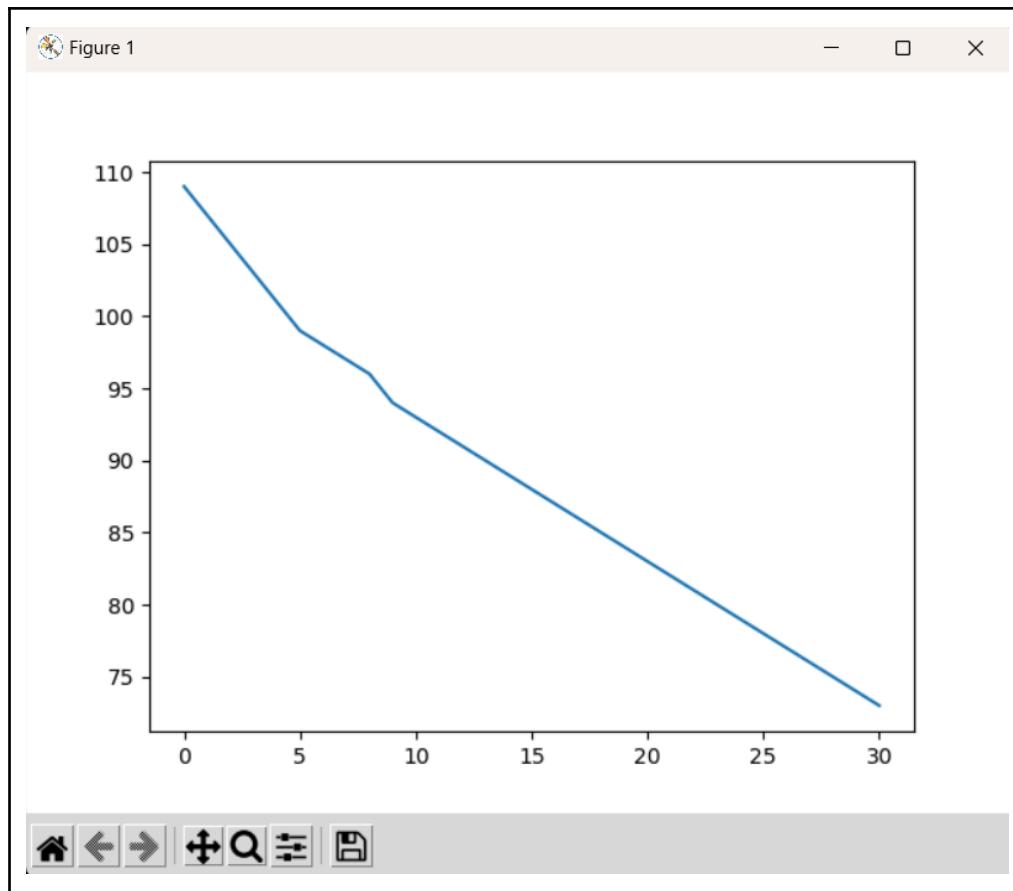
a. Percobaan 1



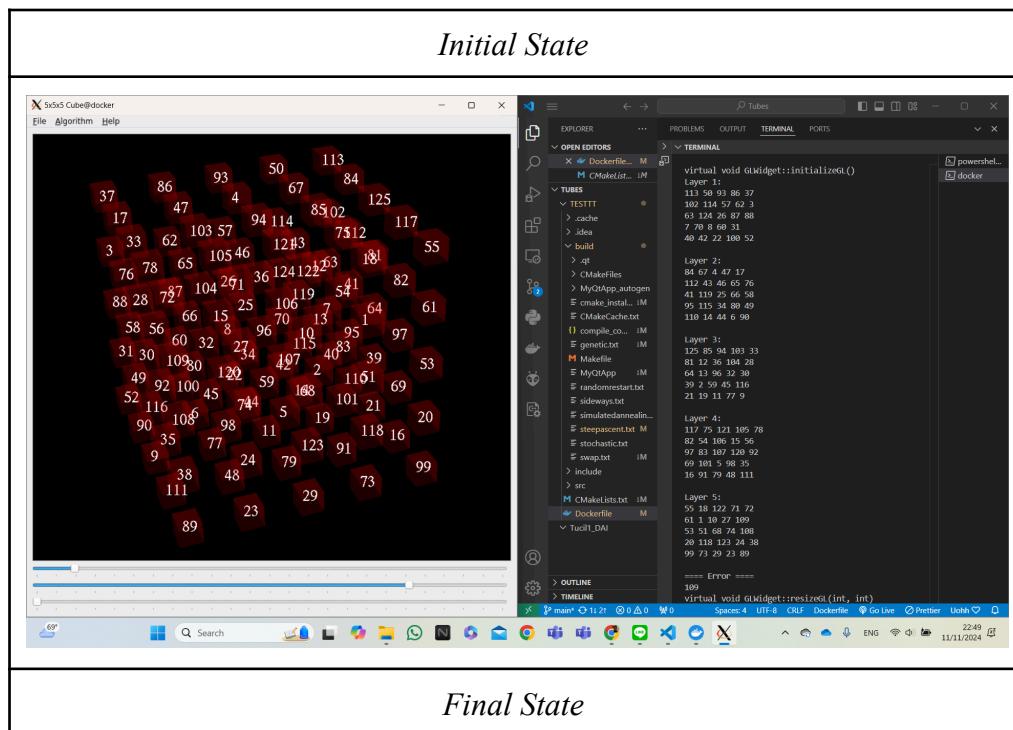


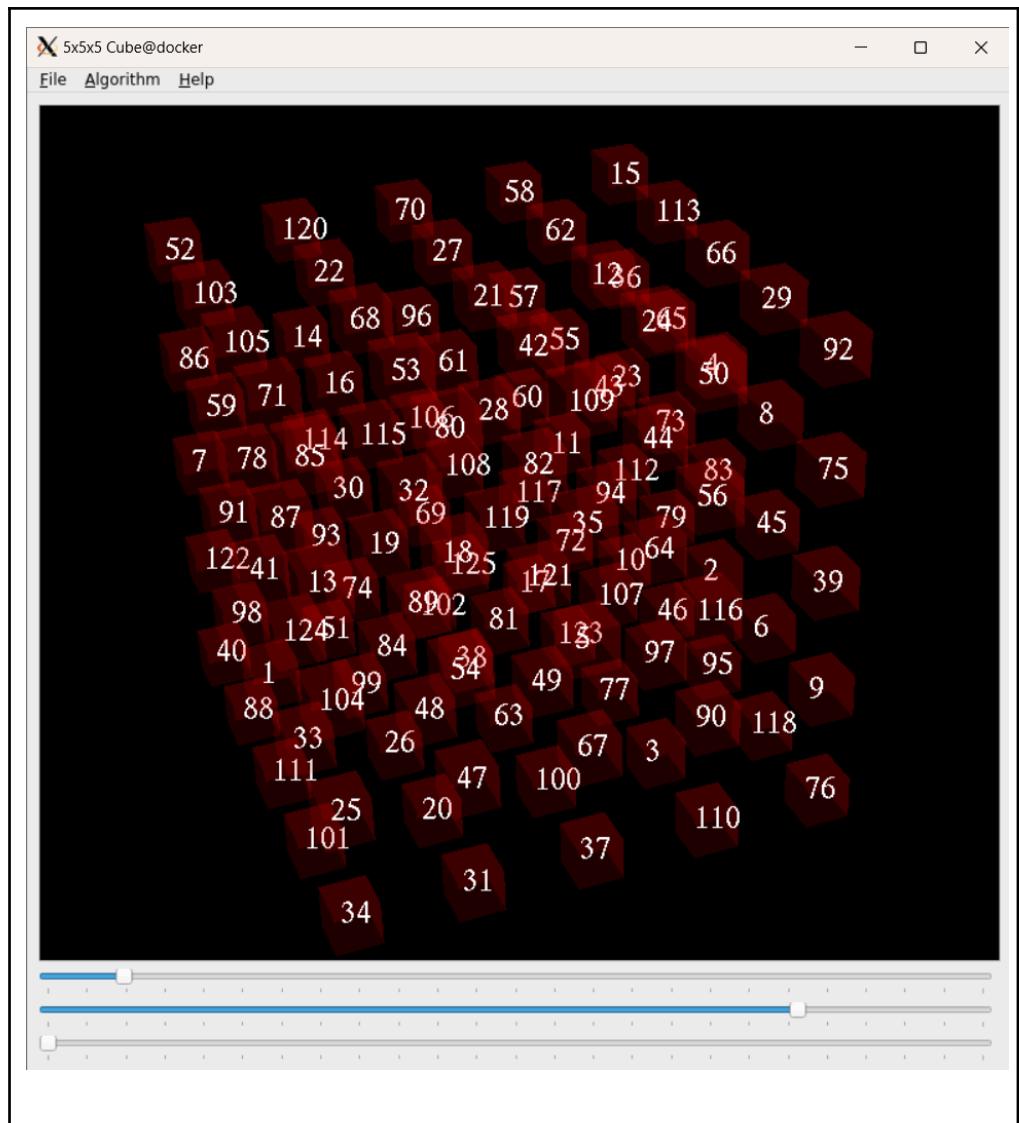
b. Percobaan 2

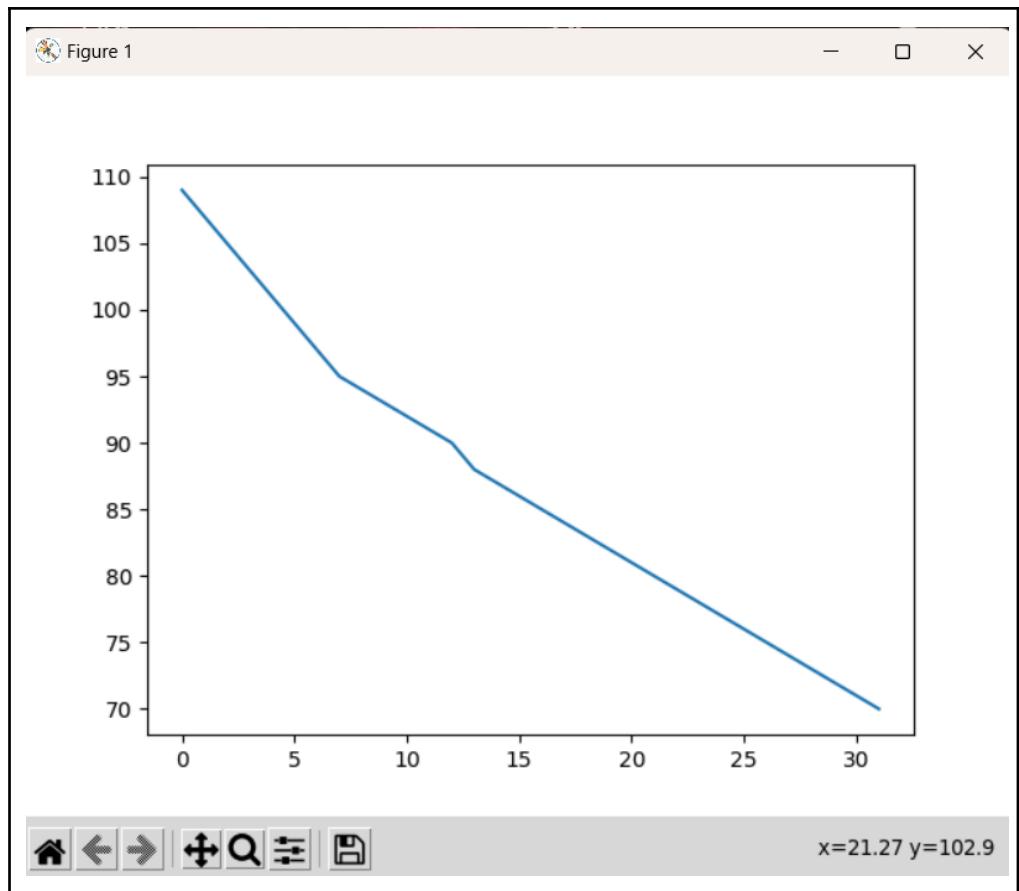




c. Percobaan 3

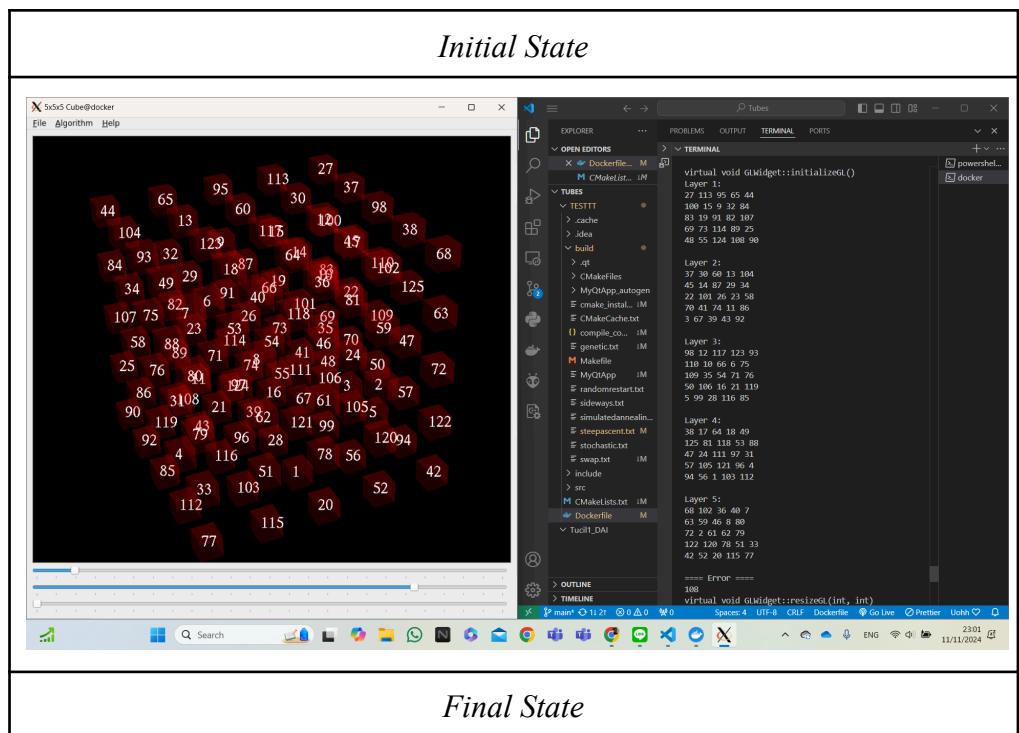


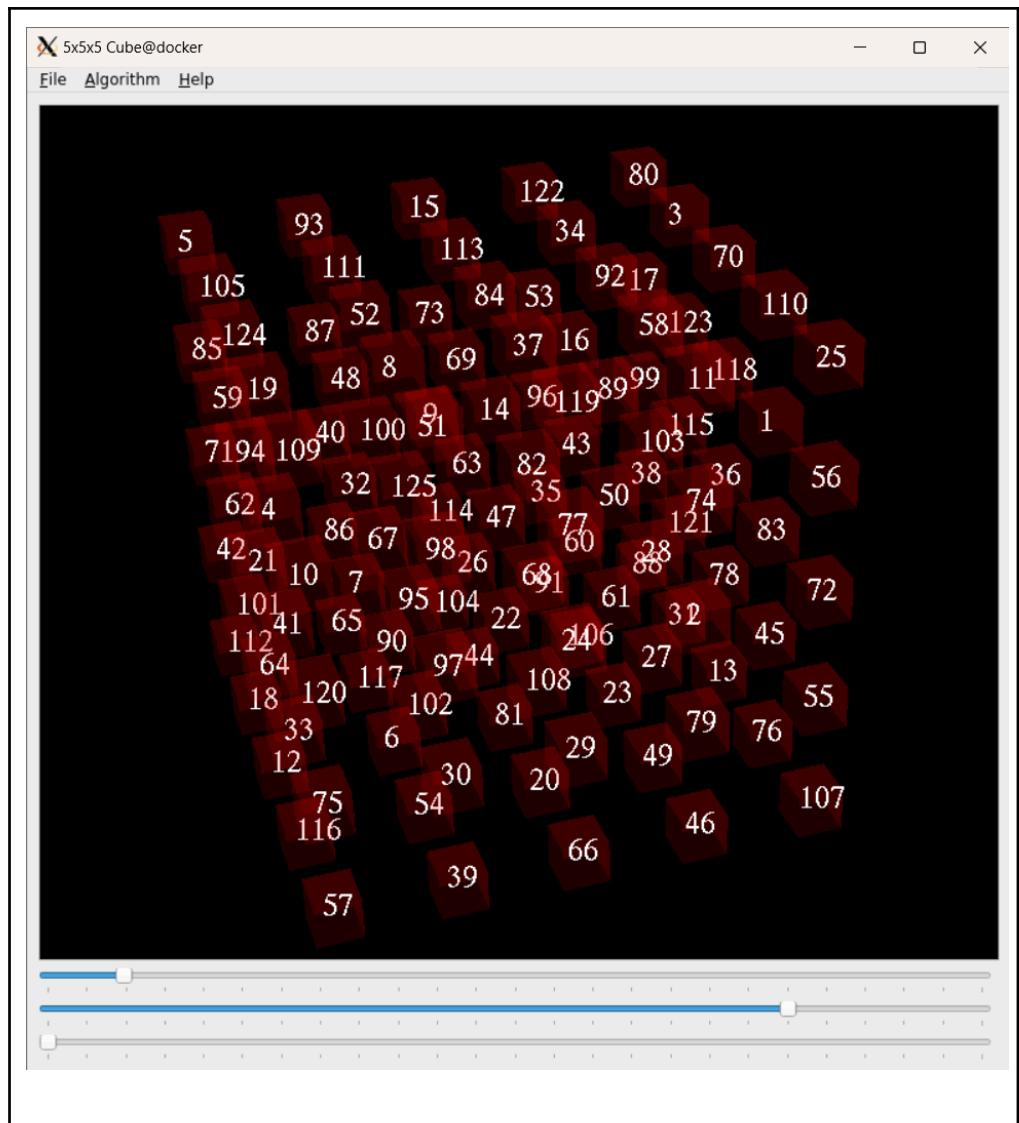


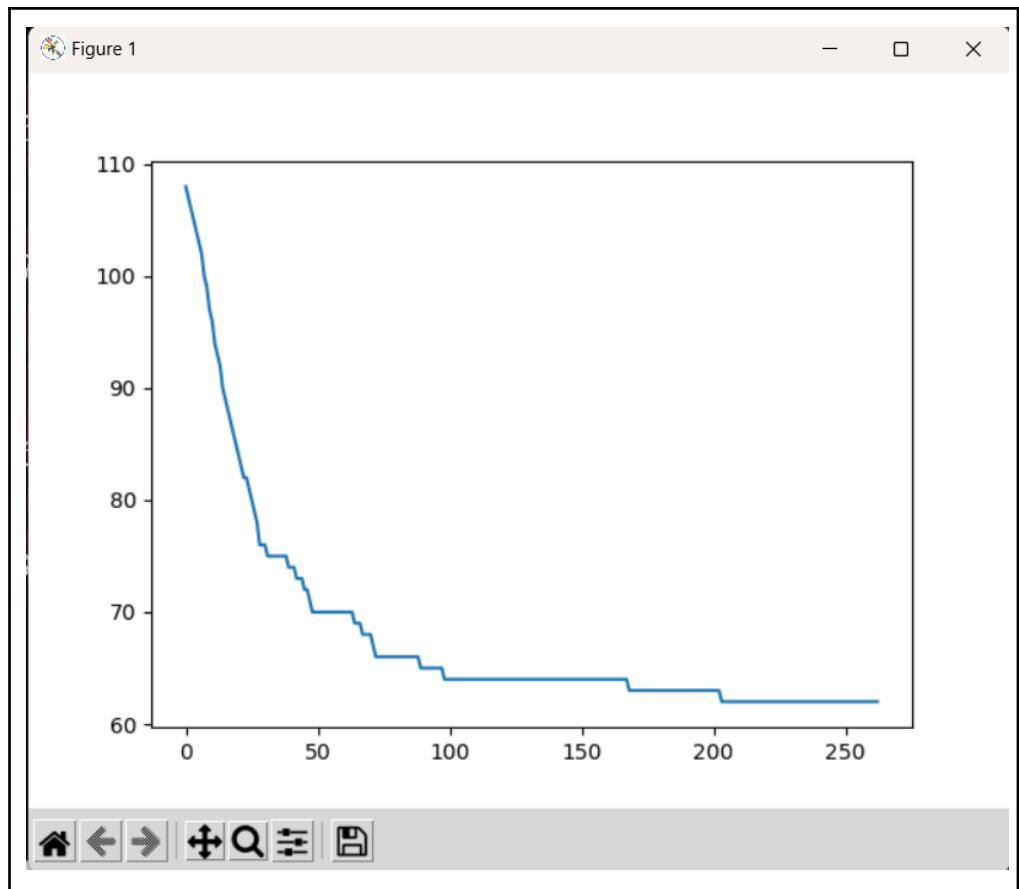


3.4 Simulated Annealing

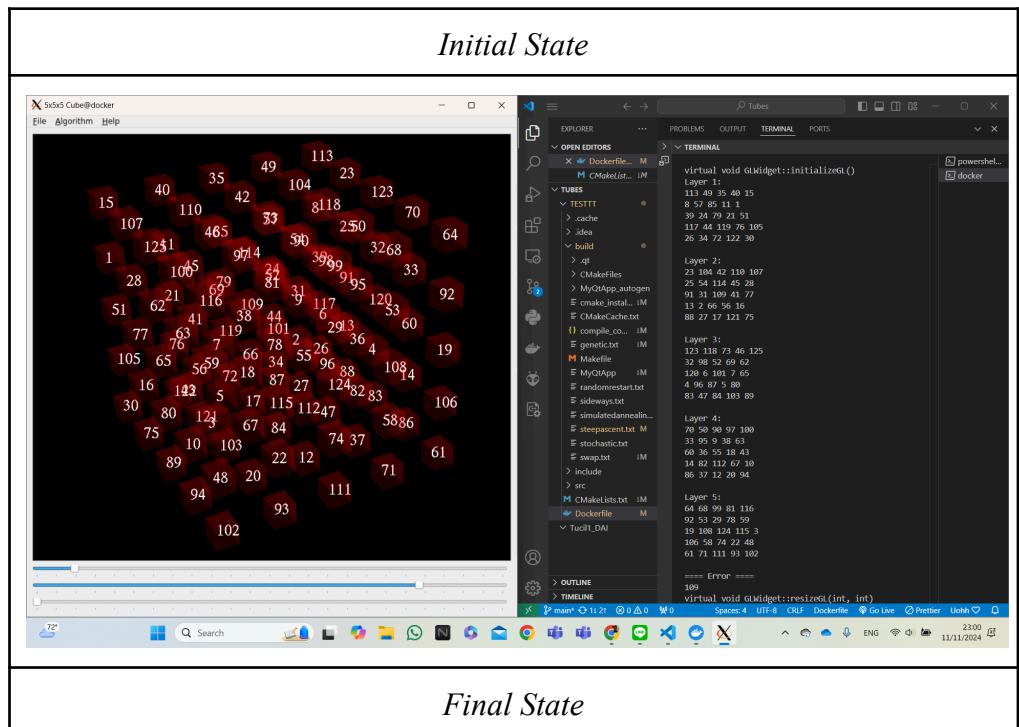
a. Percobaan 1

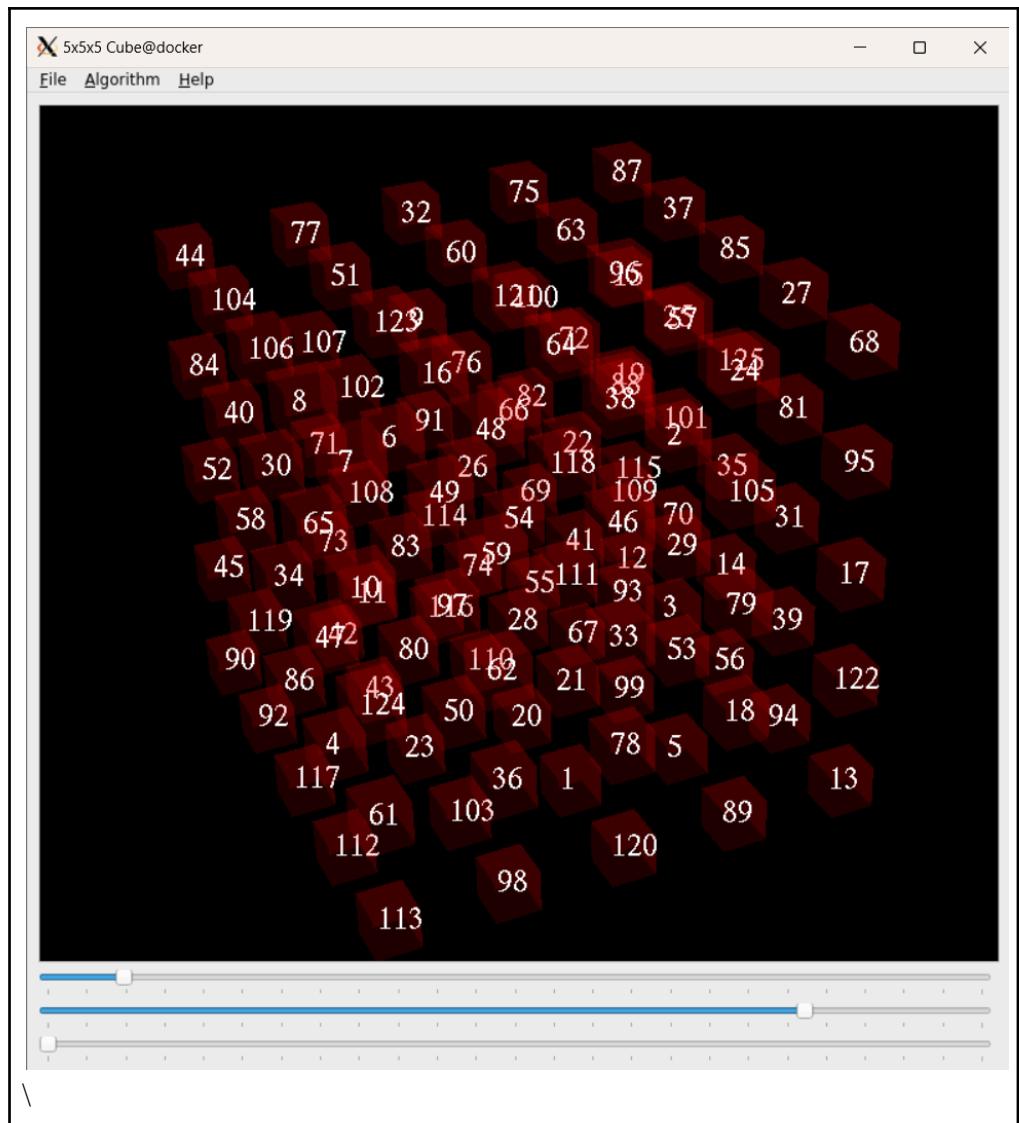


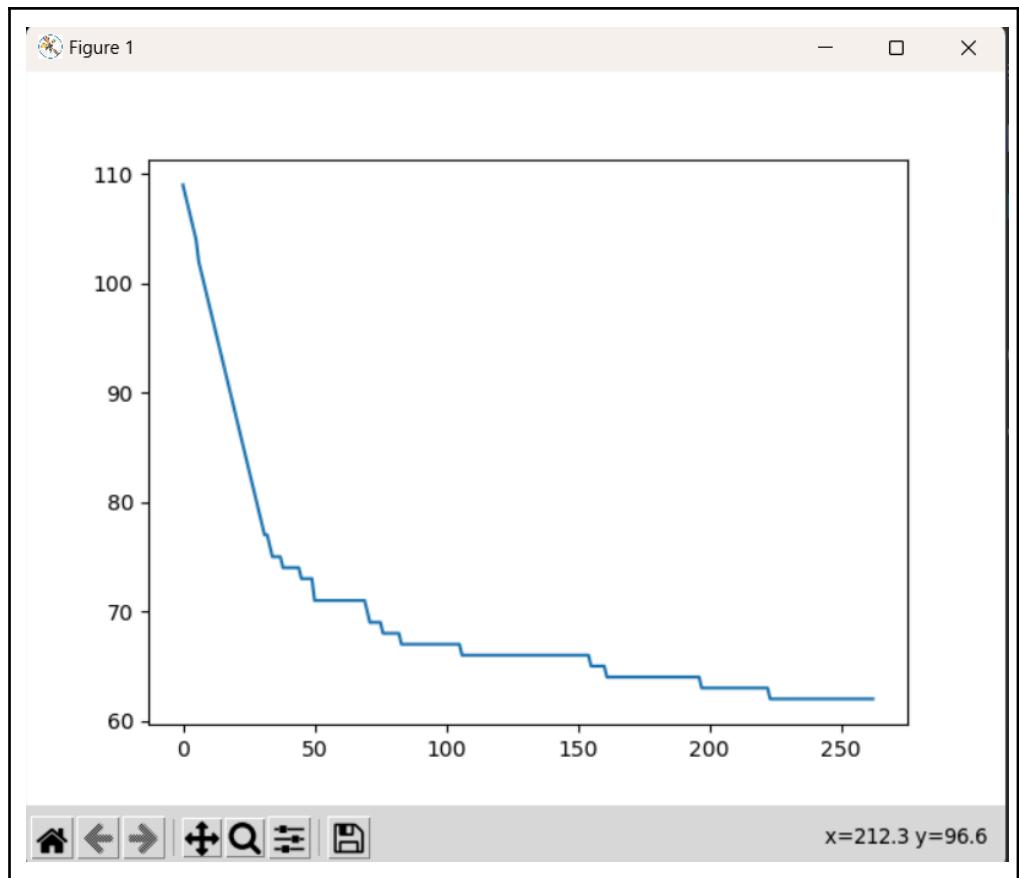




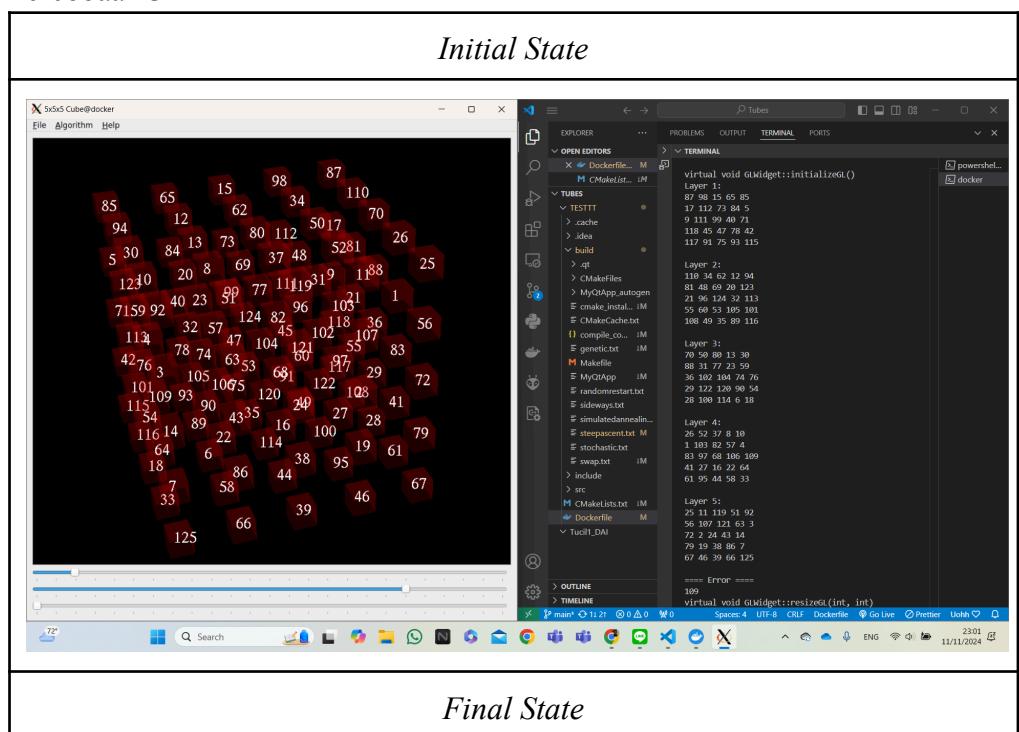
b. Percobaan 2

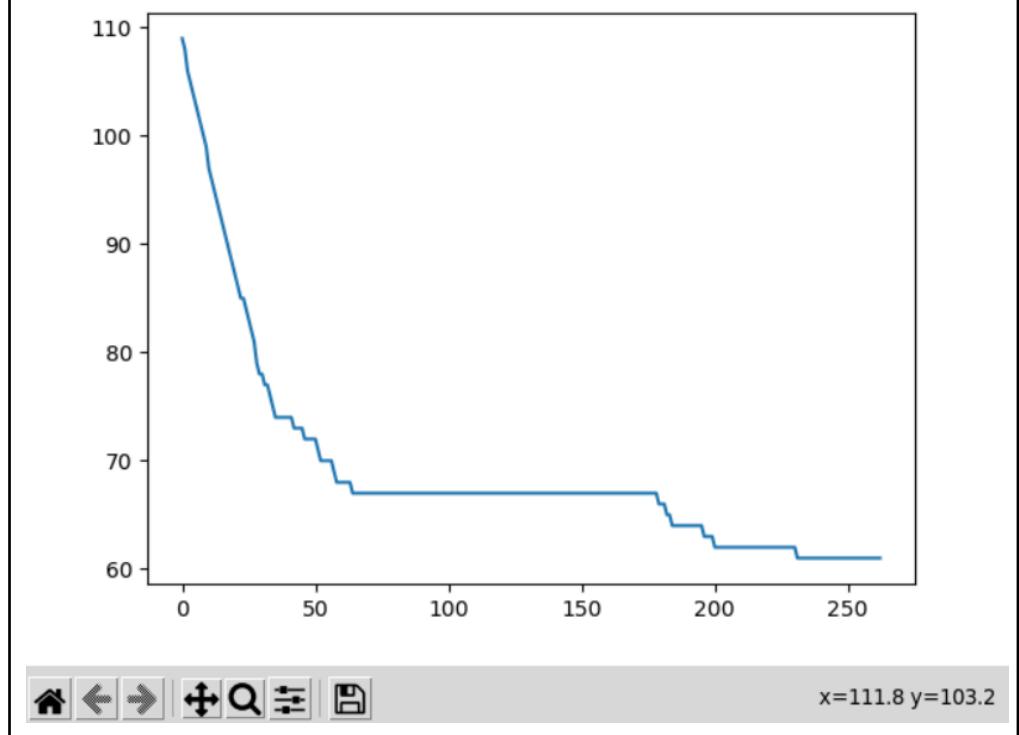
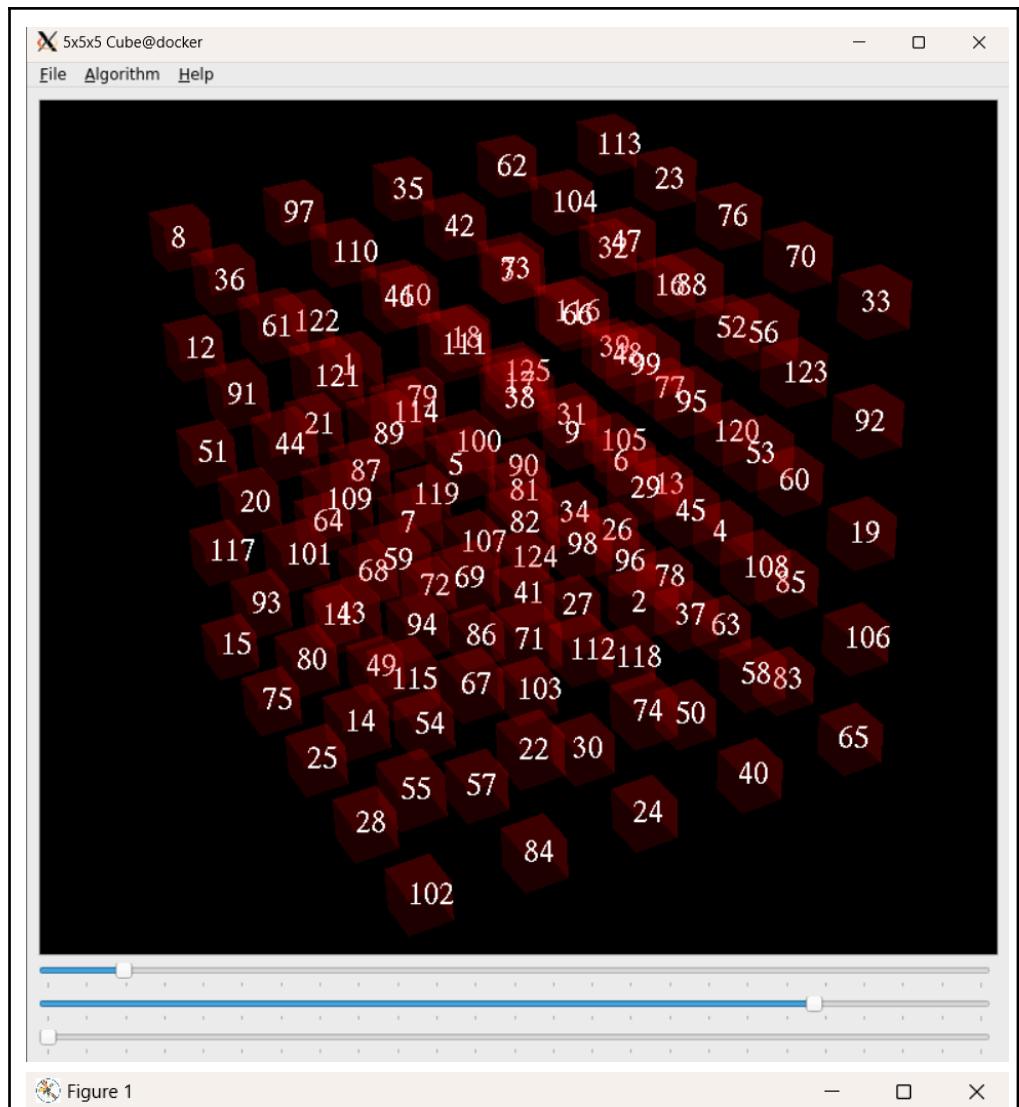






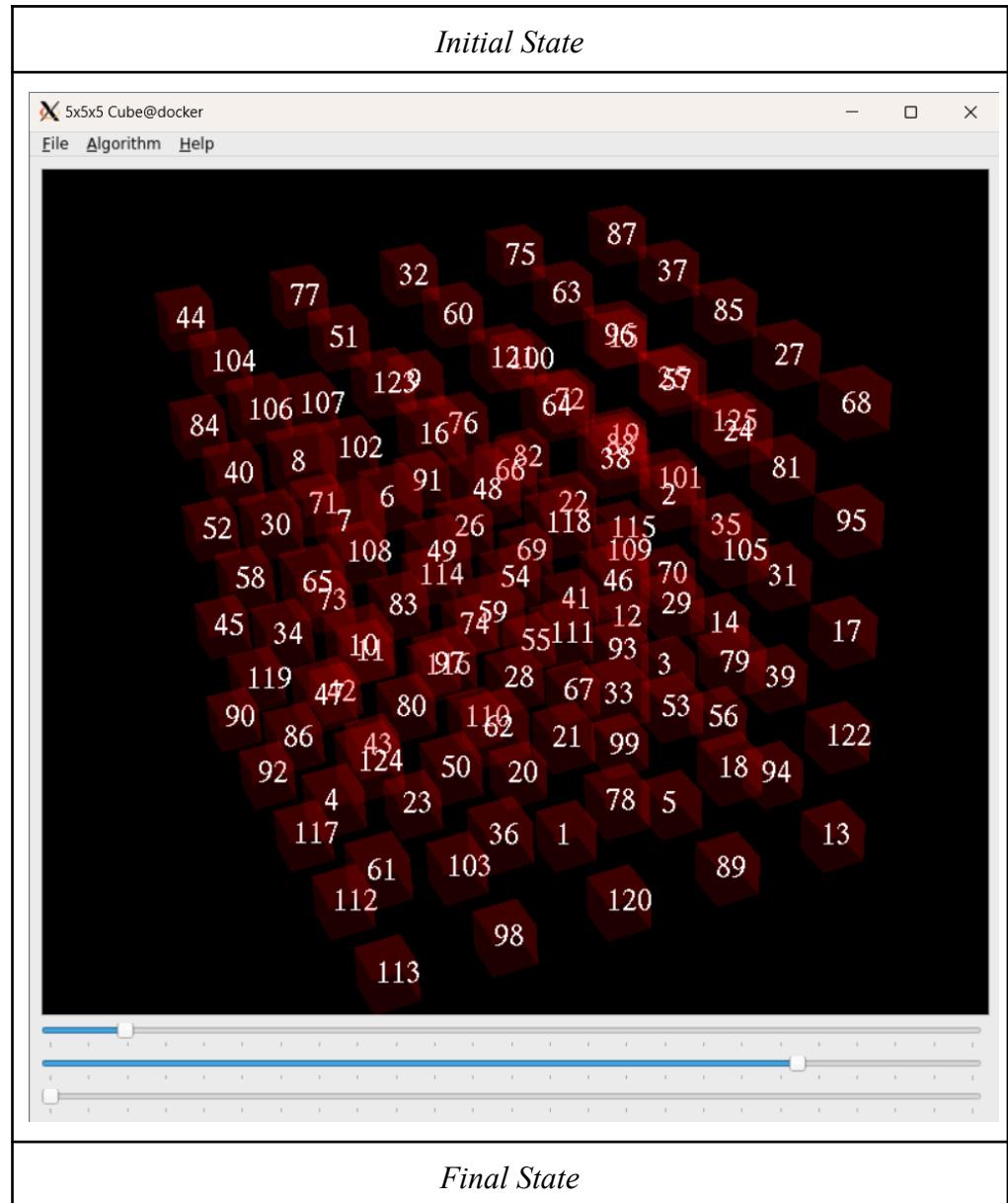
c. Percobaan 3

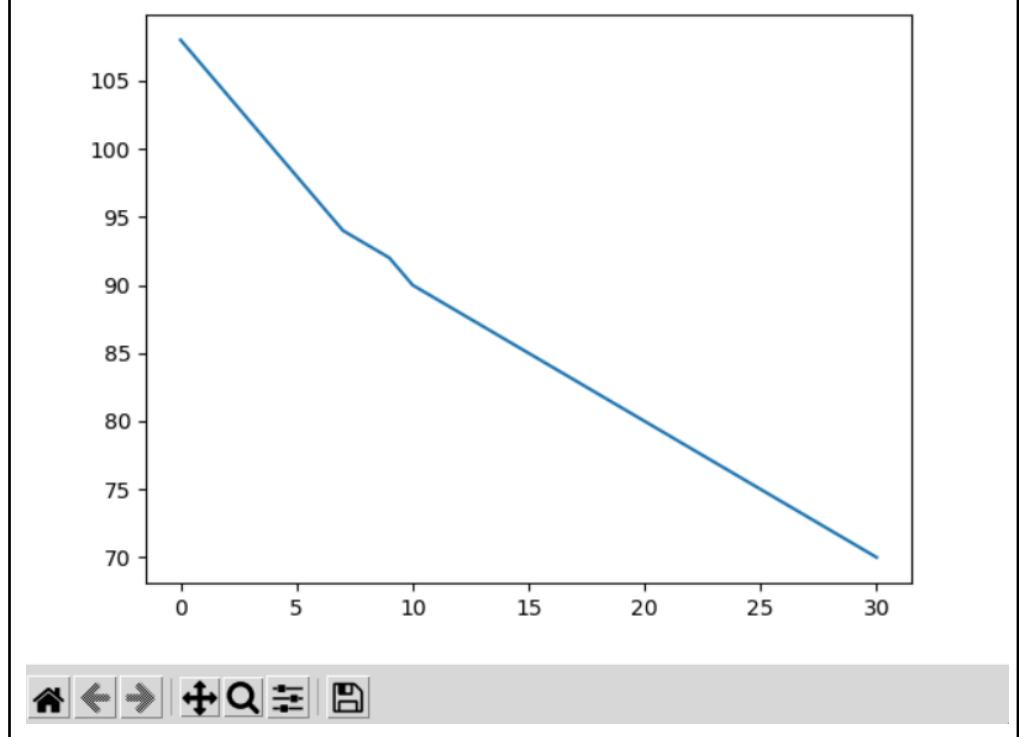
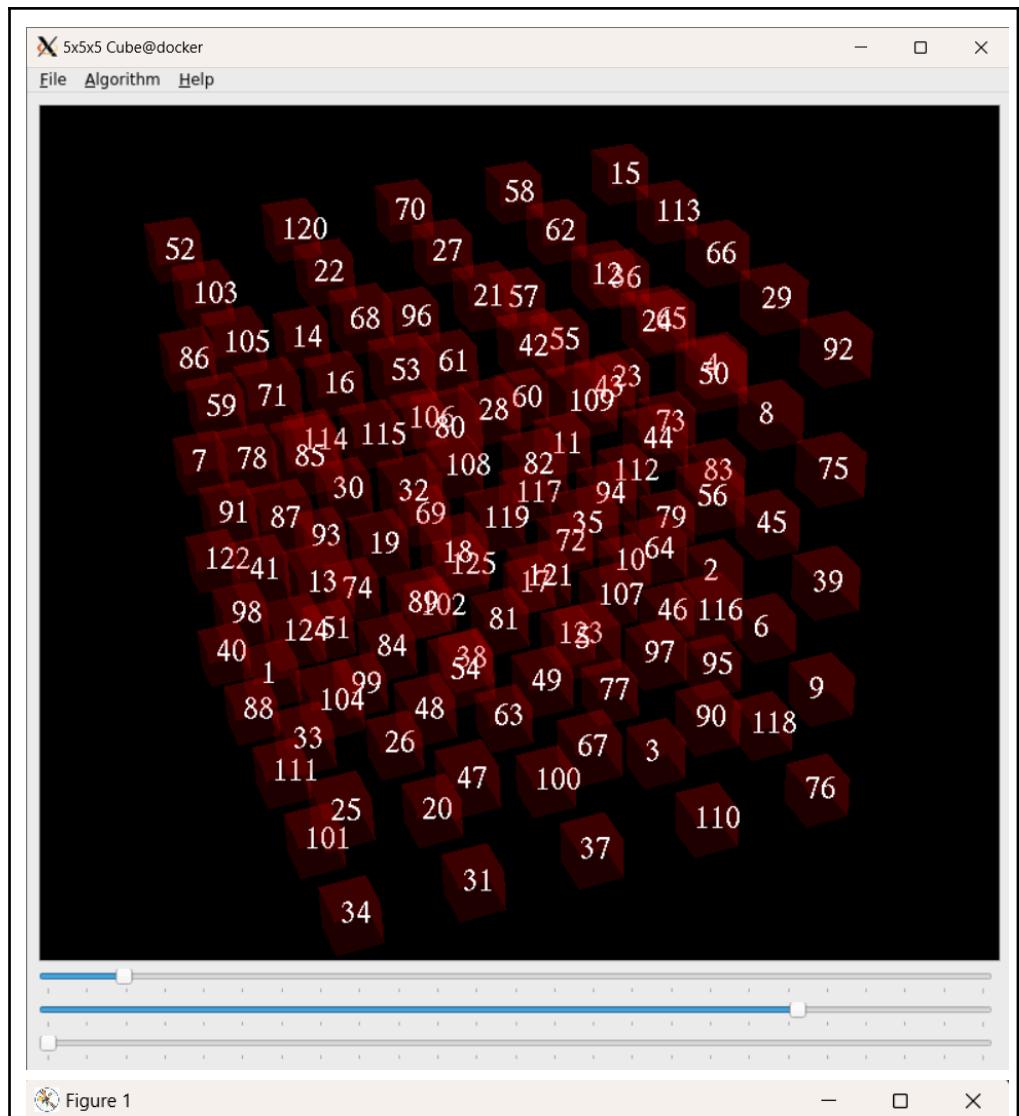




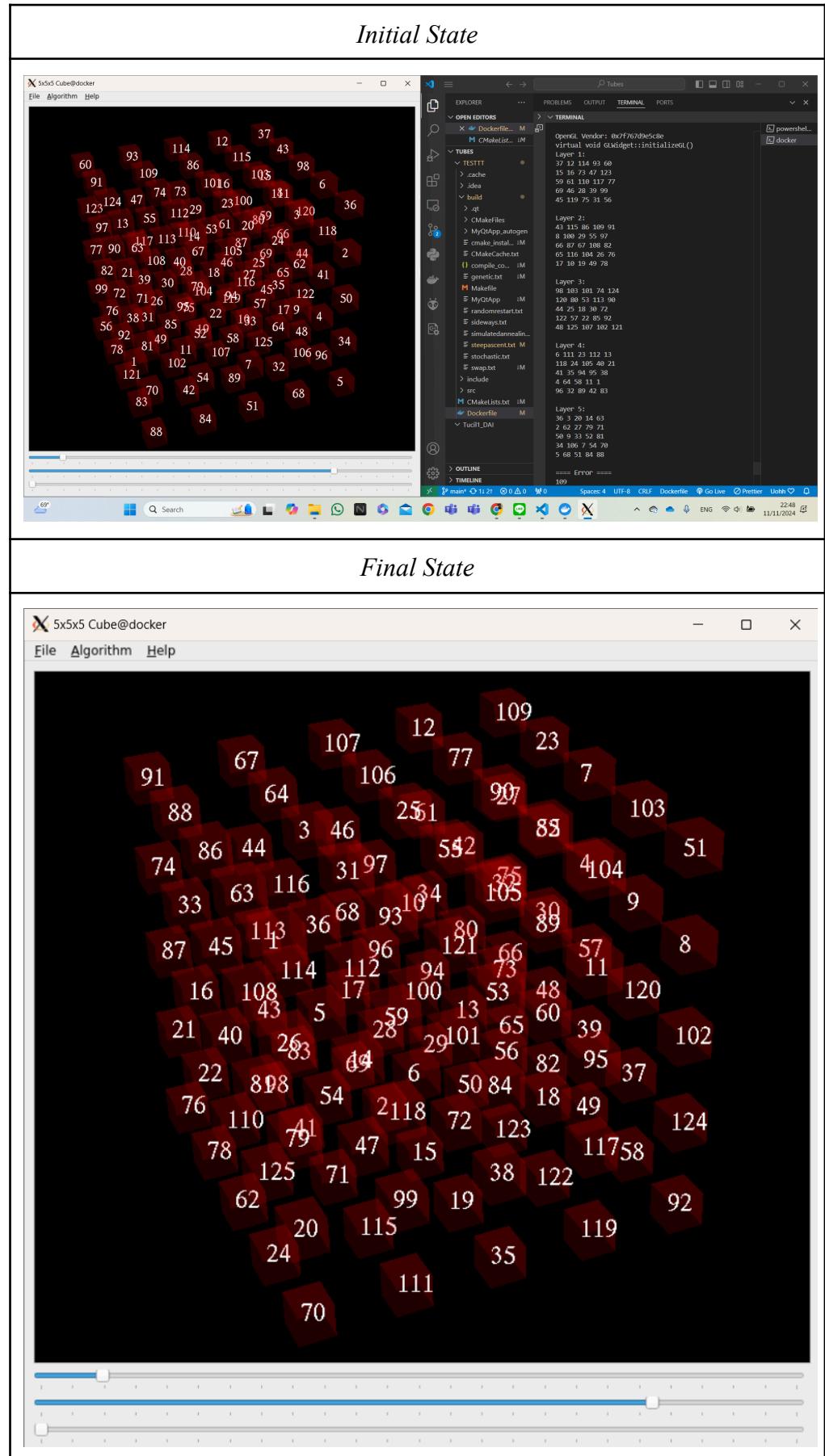
3.5 Stochastic Hill-Climbing

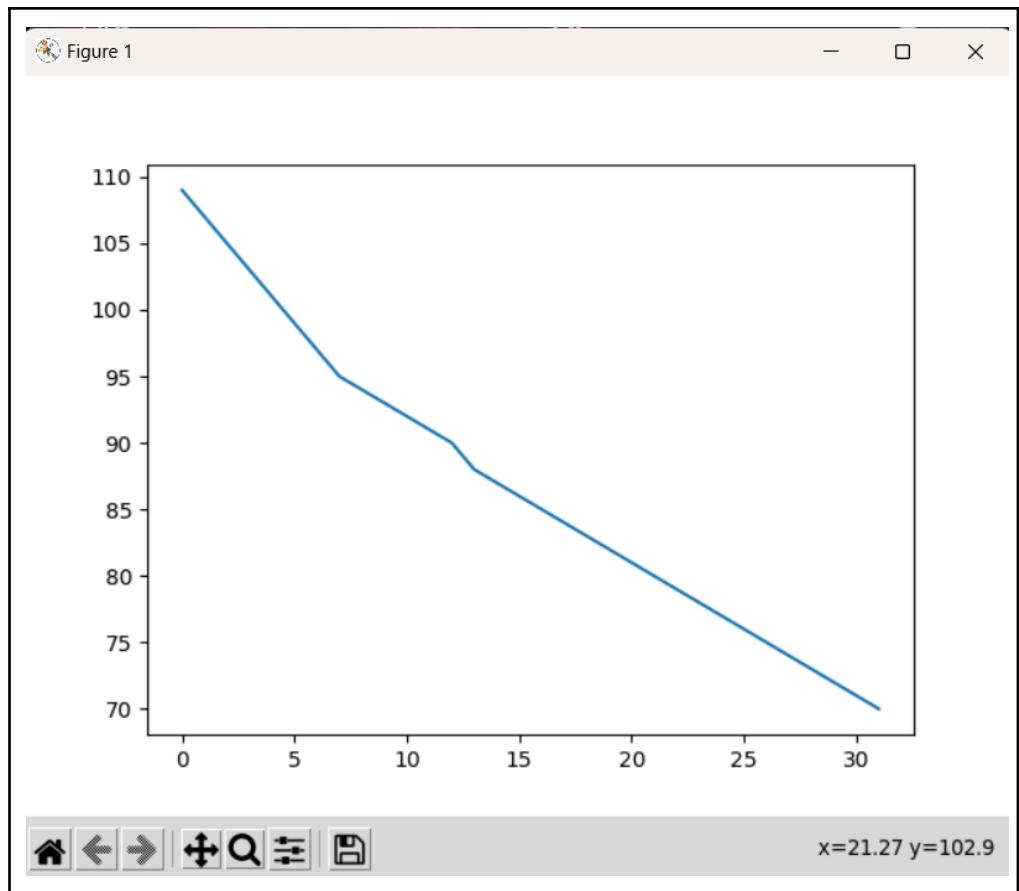
a. Percobaan 1



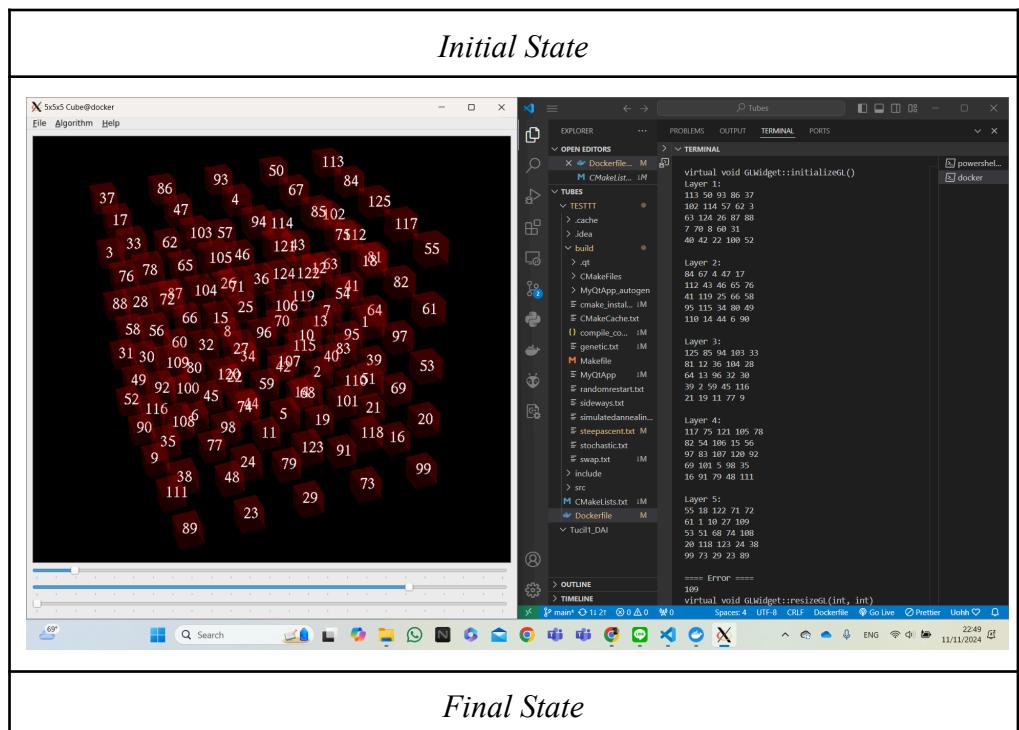


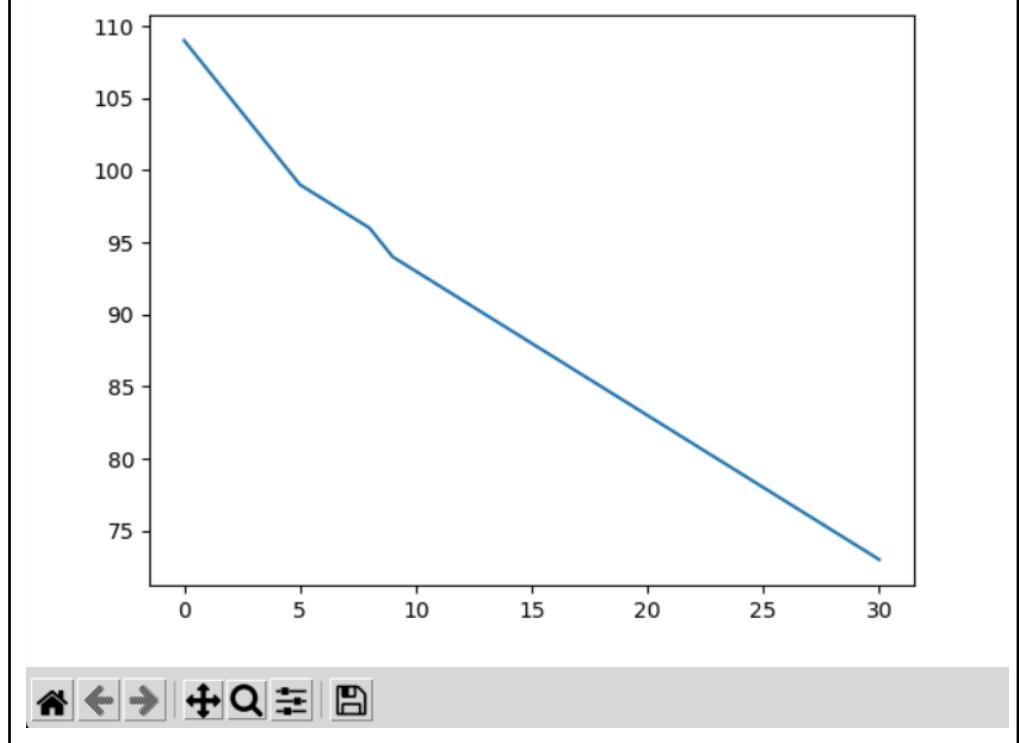
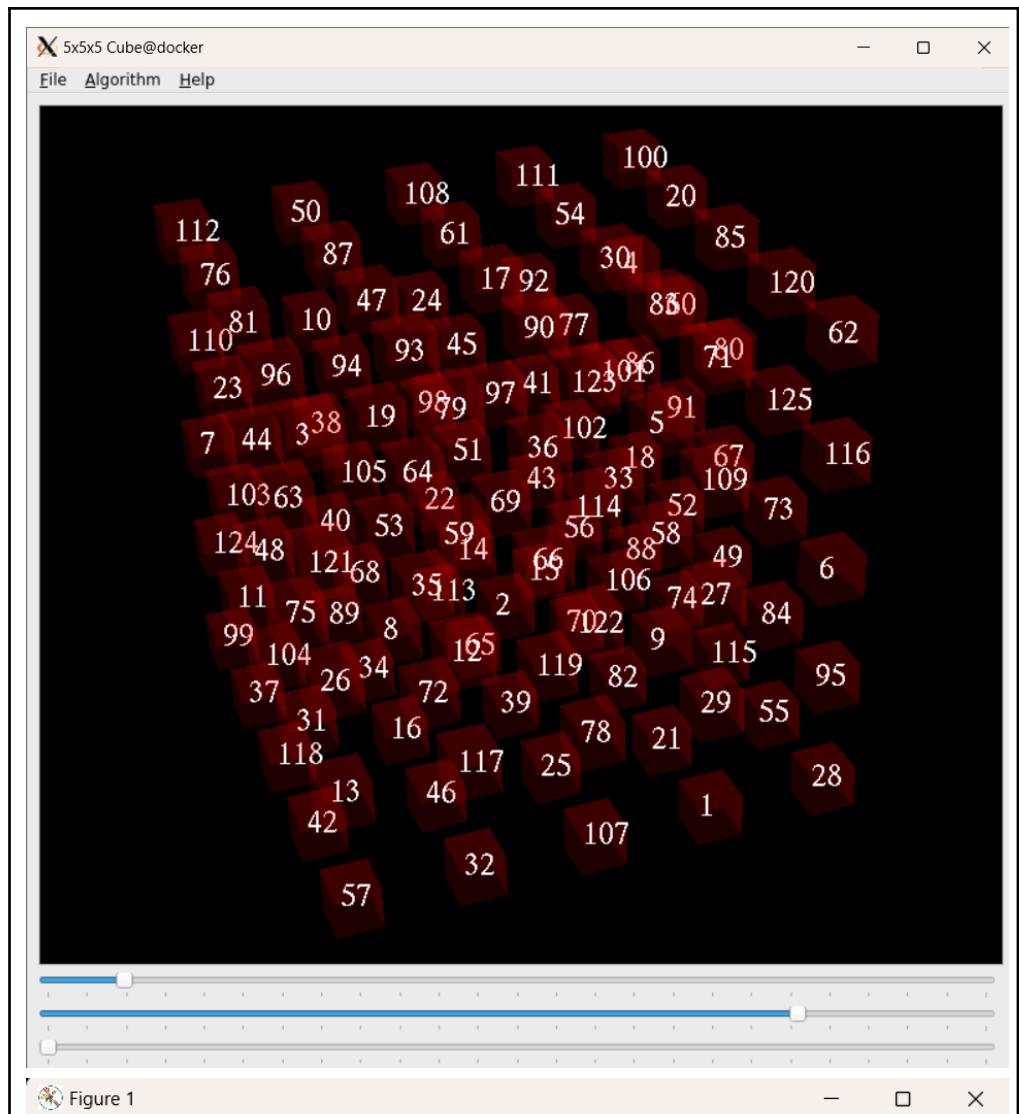
b. Percobaan 2





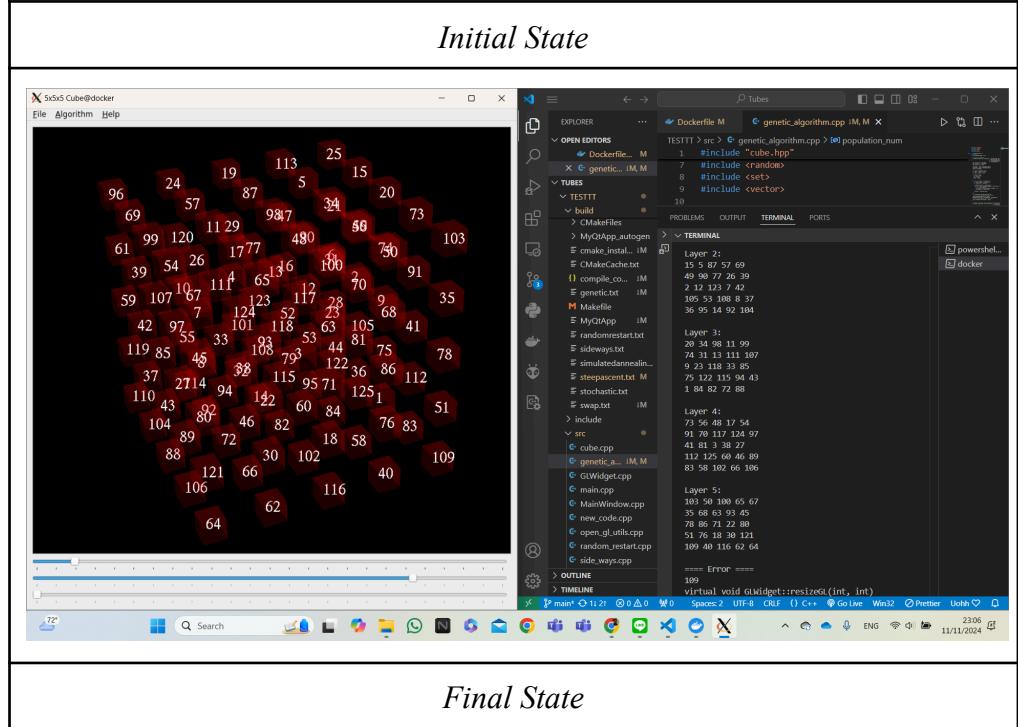
c. Percobaan 3

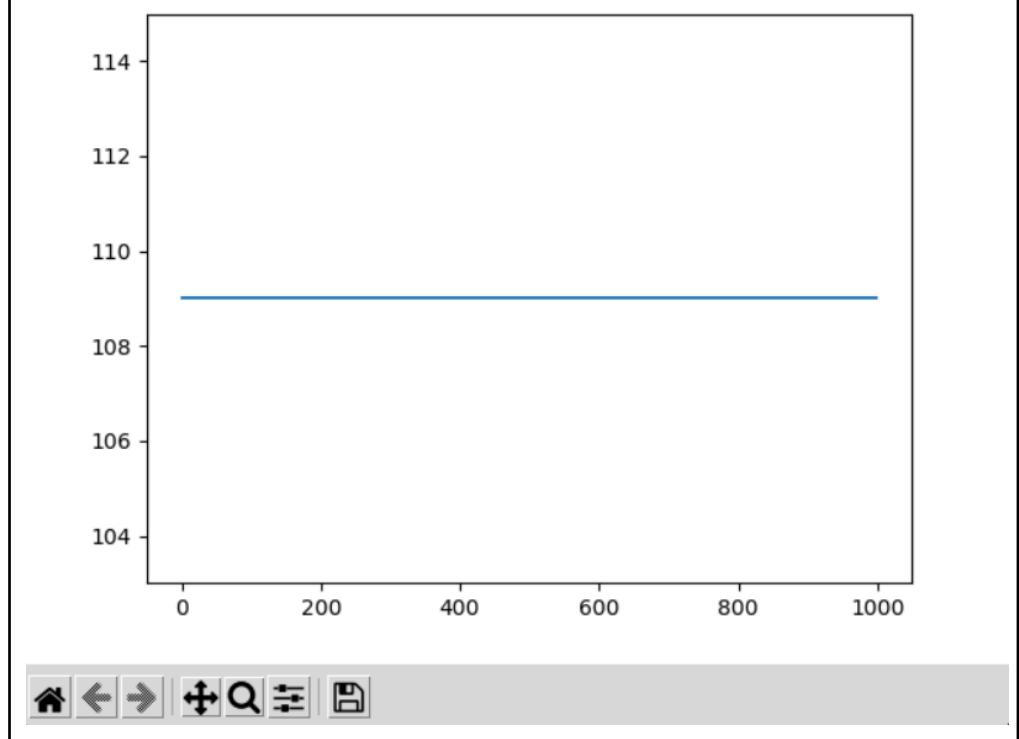
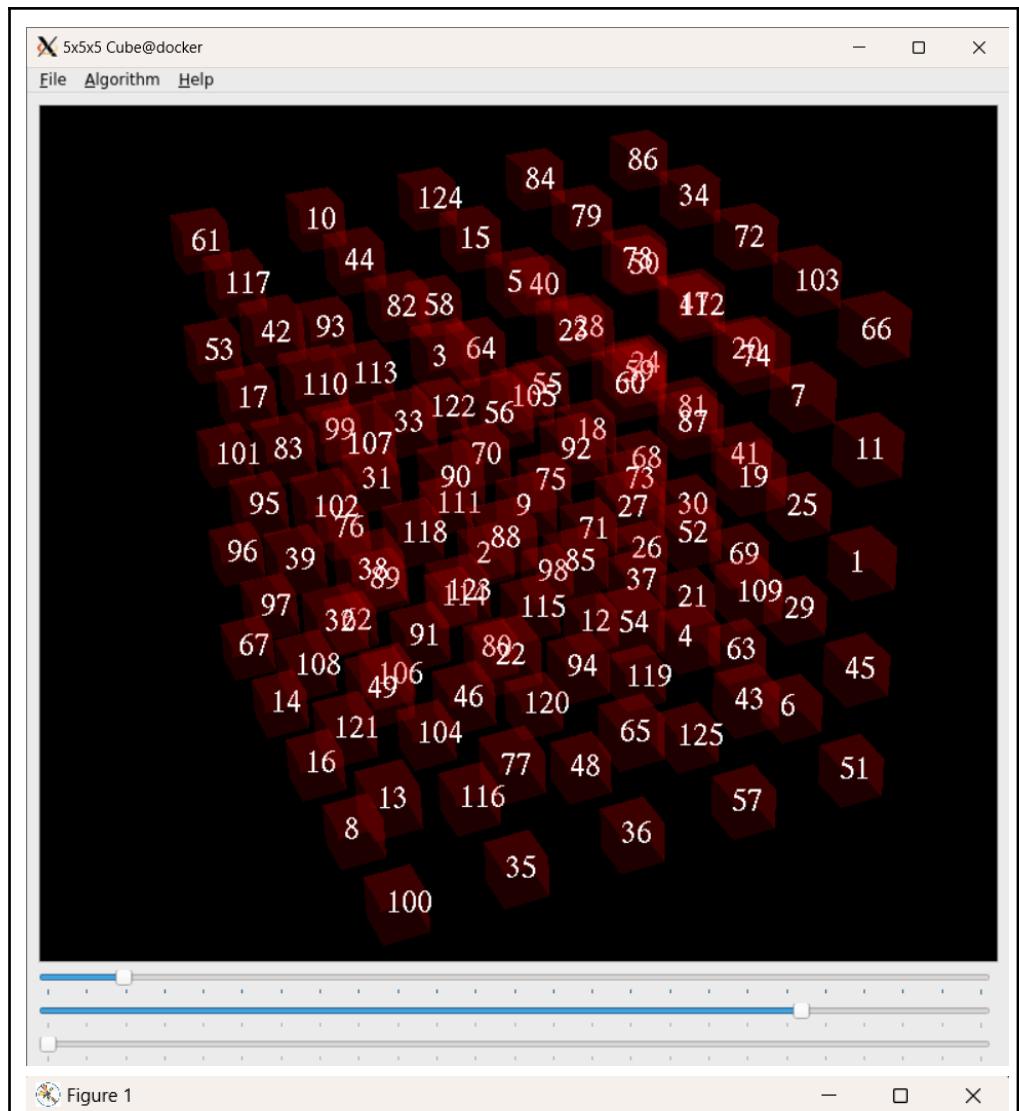




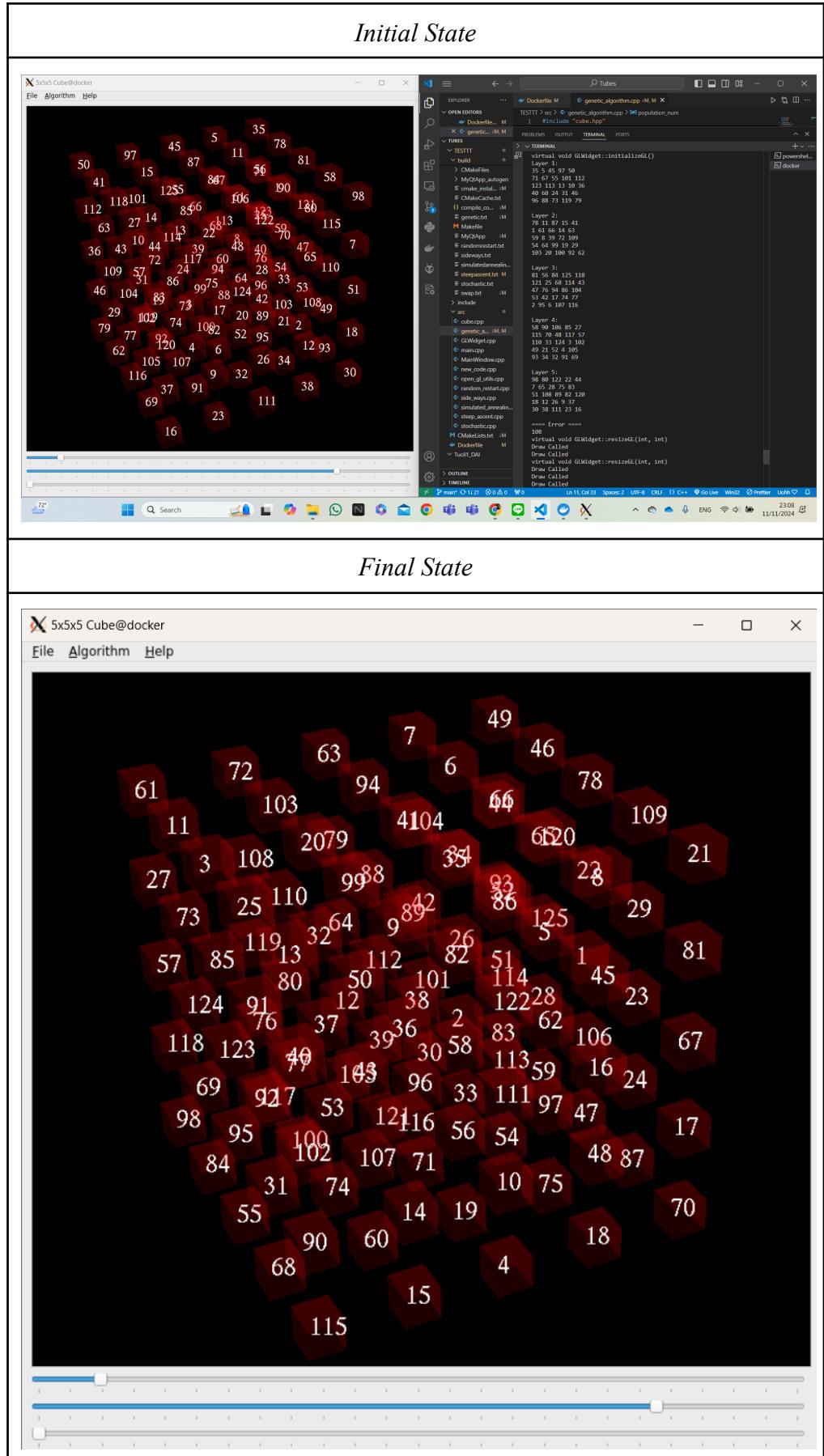
3.6 Genetic Algorithm

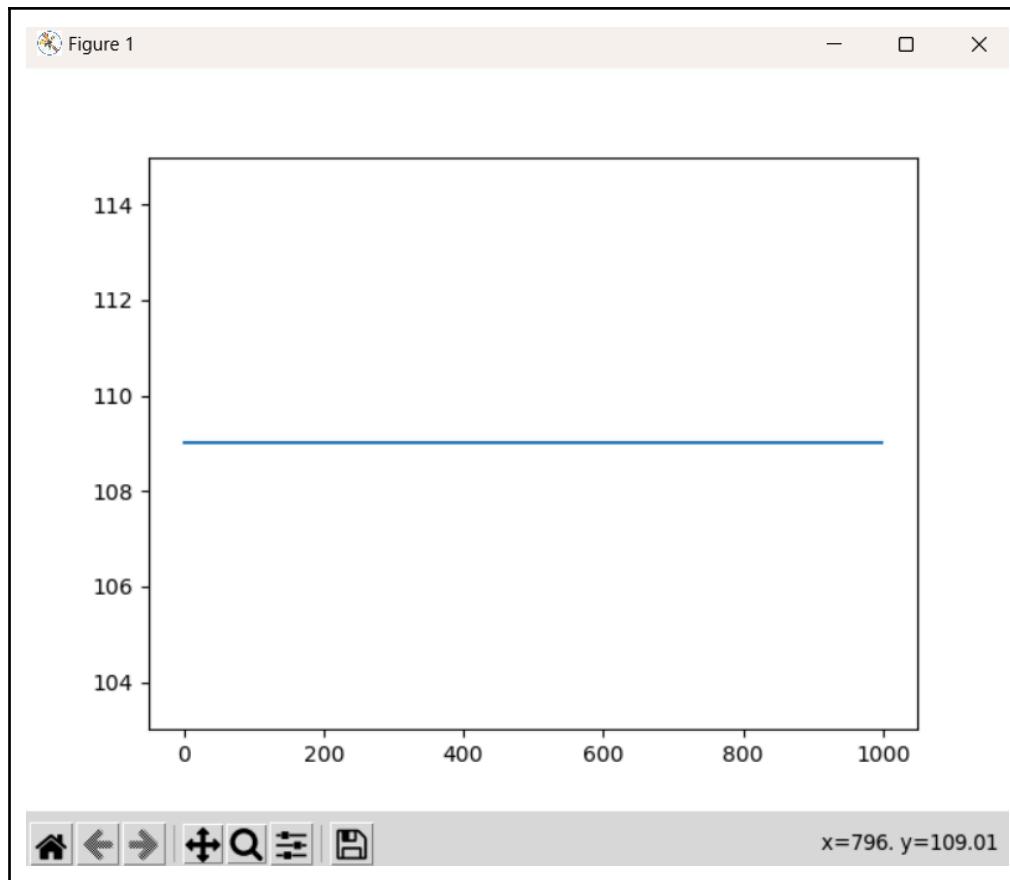
a. Populasi 6



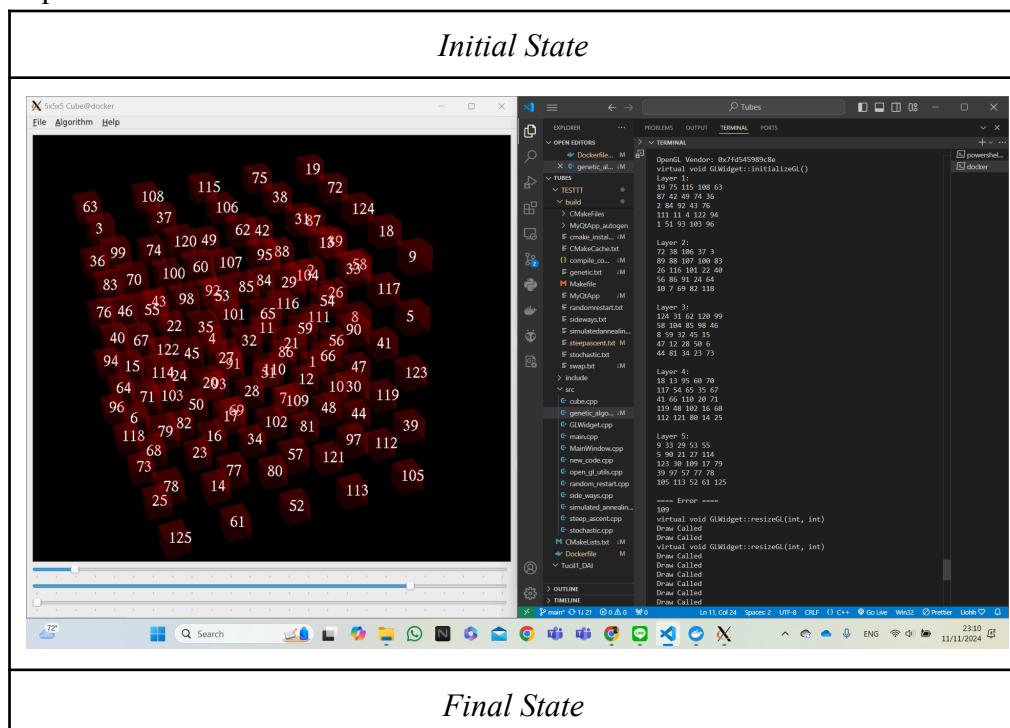


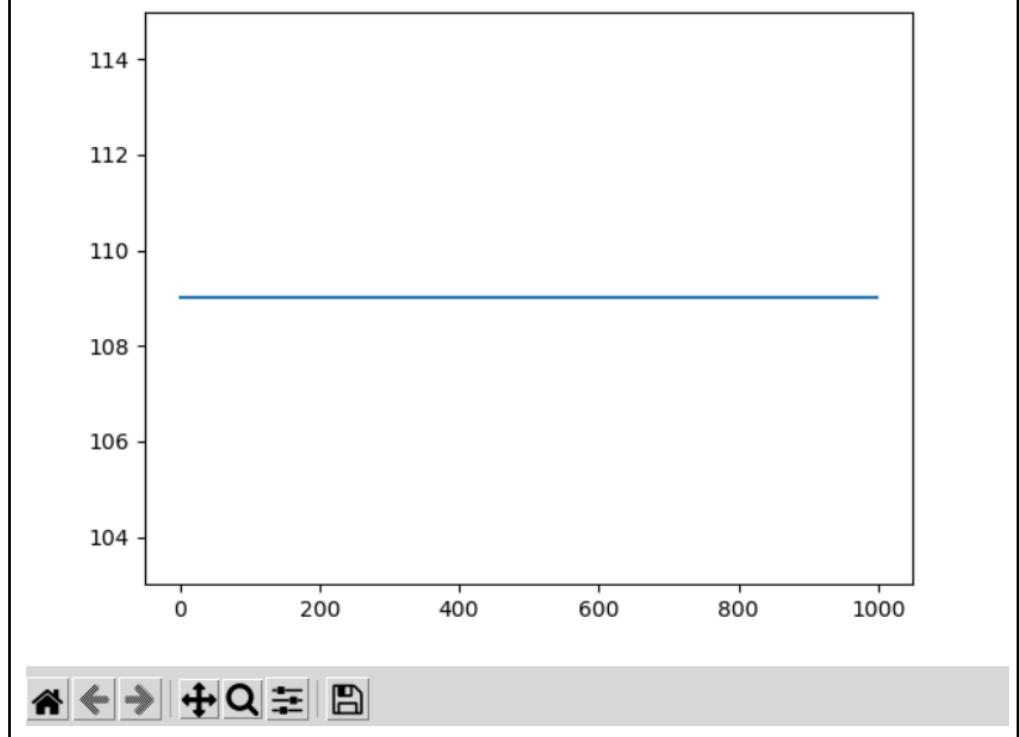
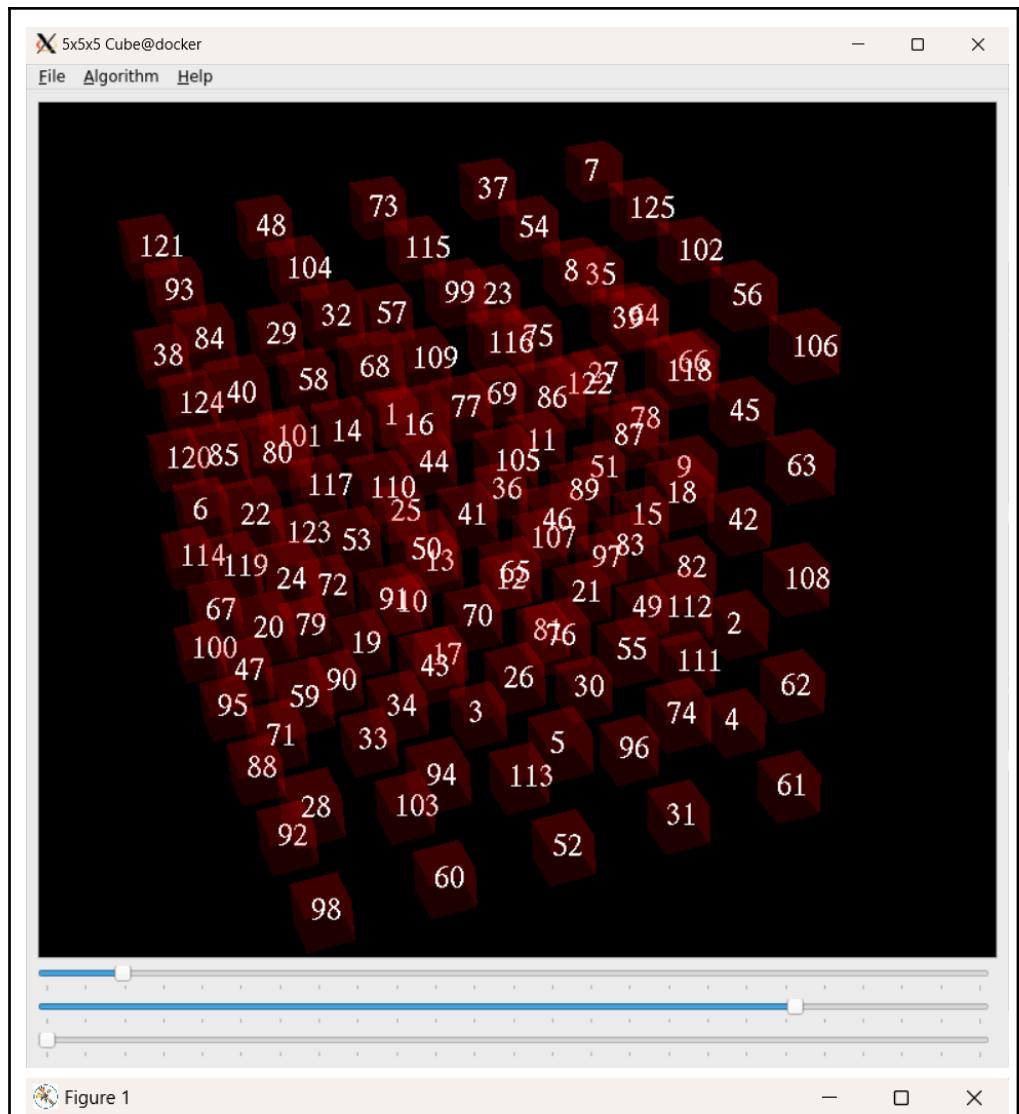
b. Populasi 8



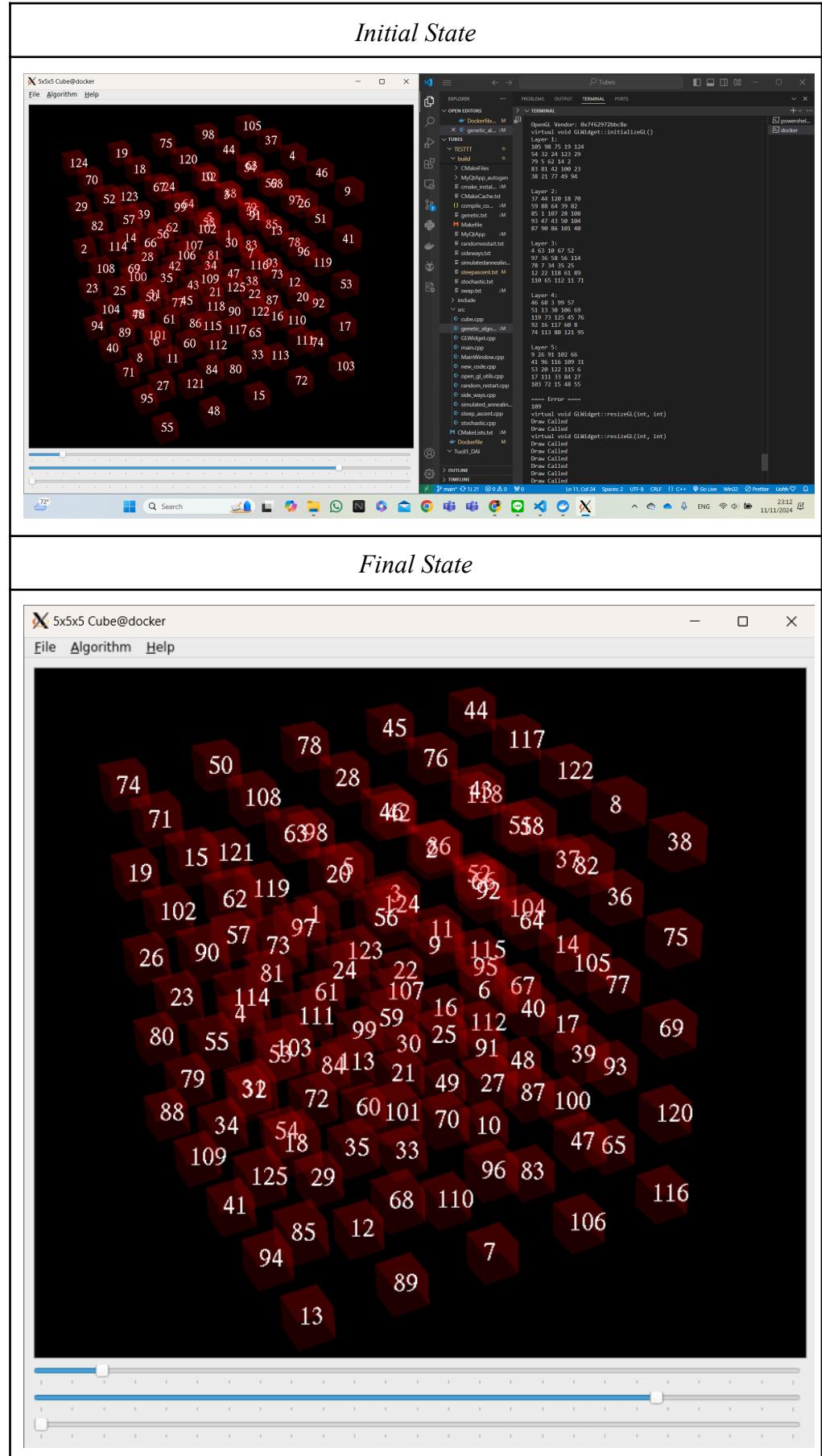


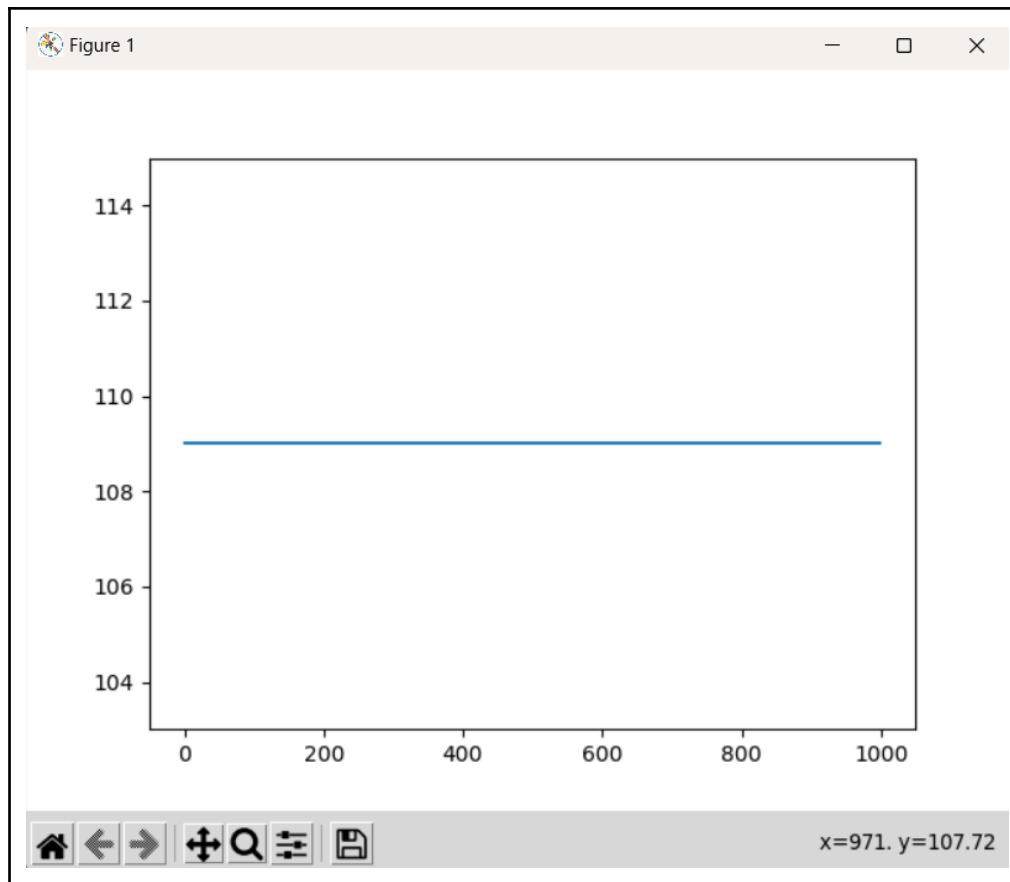
c. Populasi 10



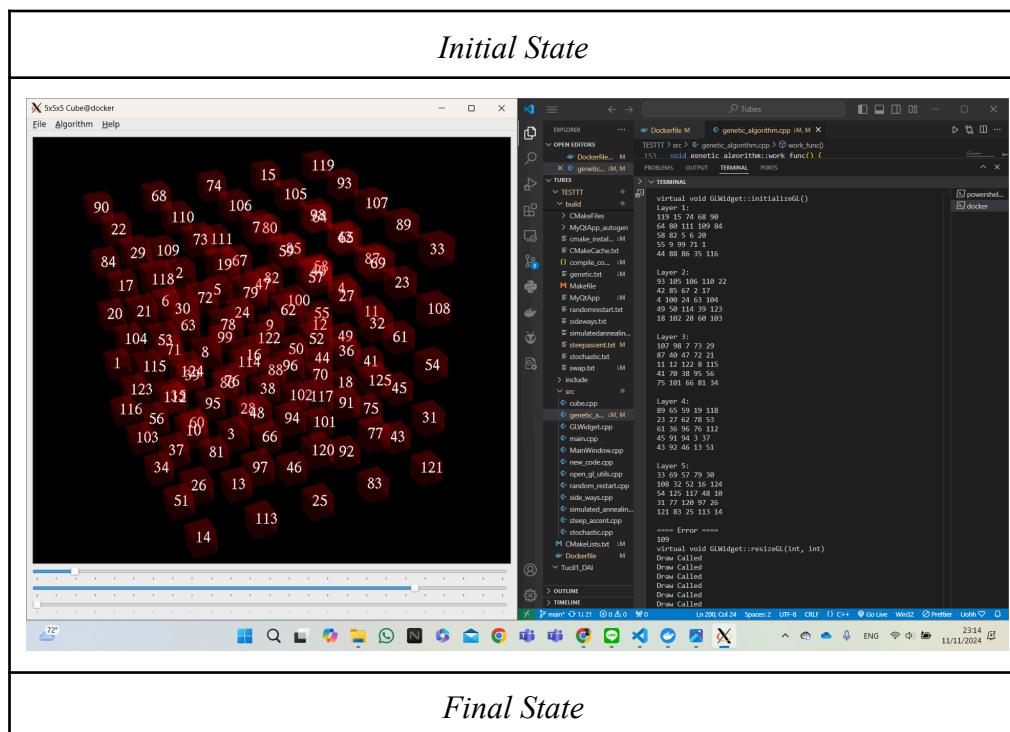


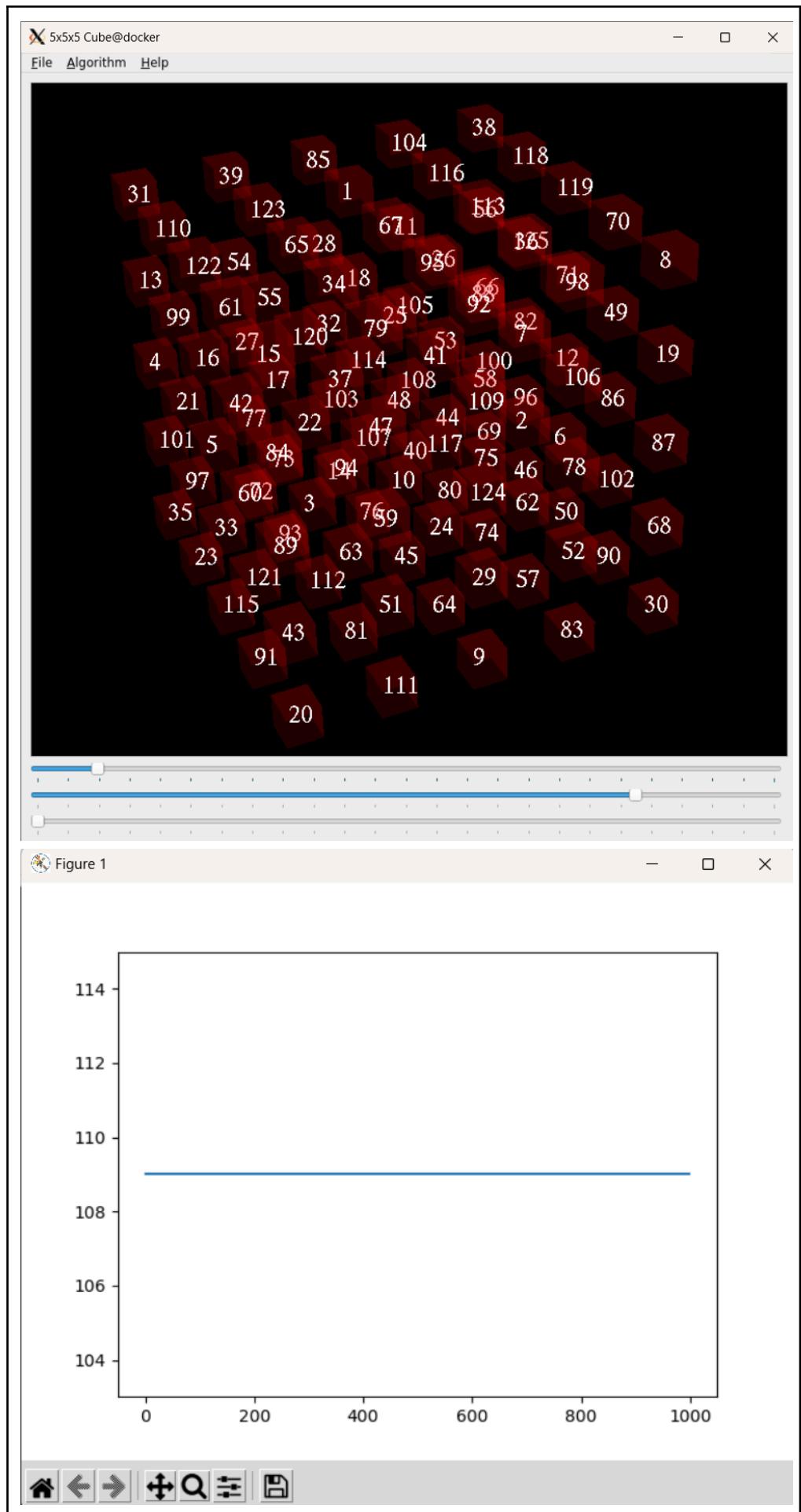
d. Iterasi 1000



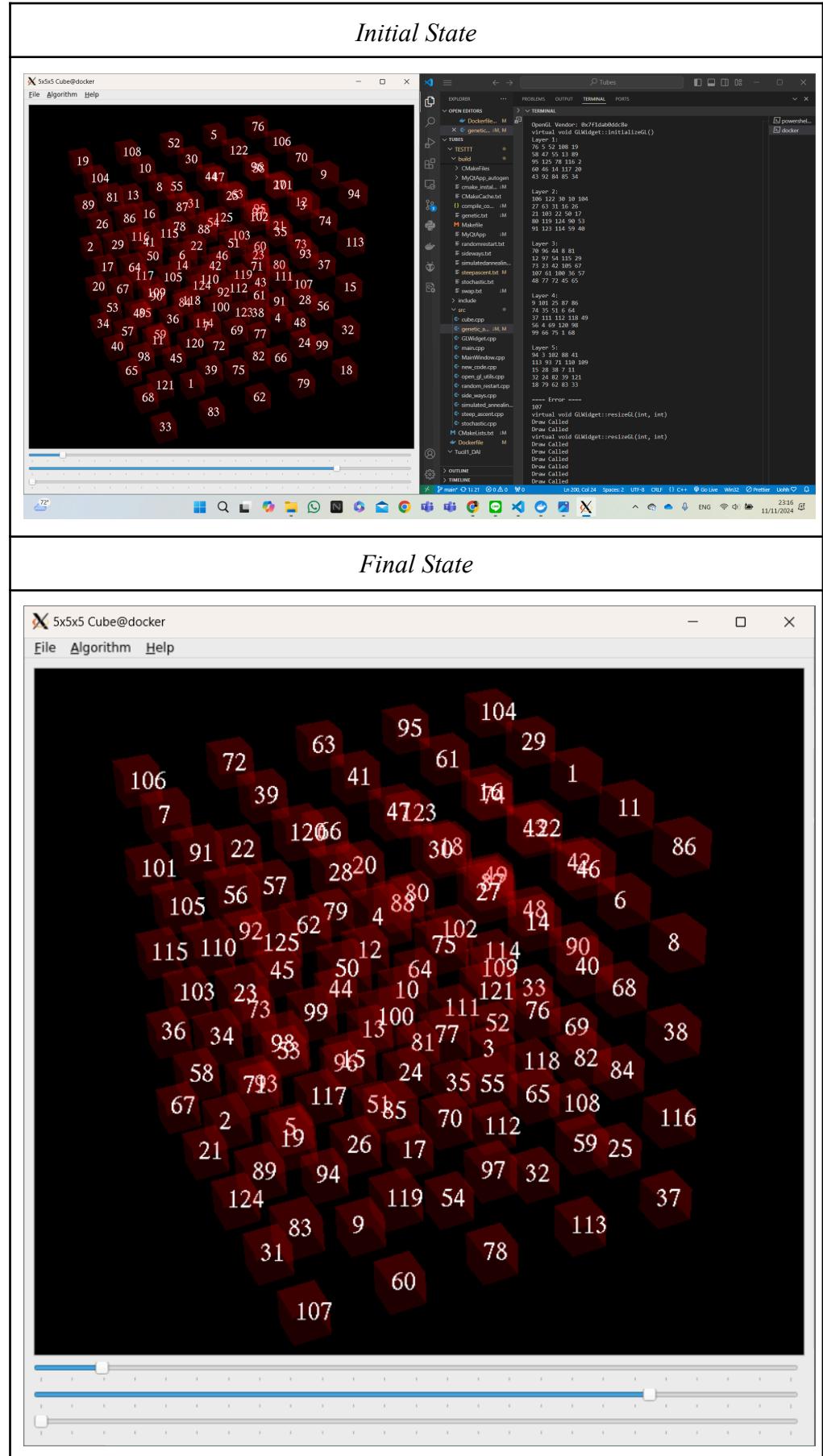


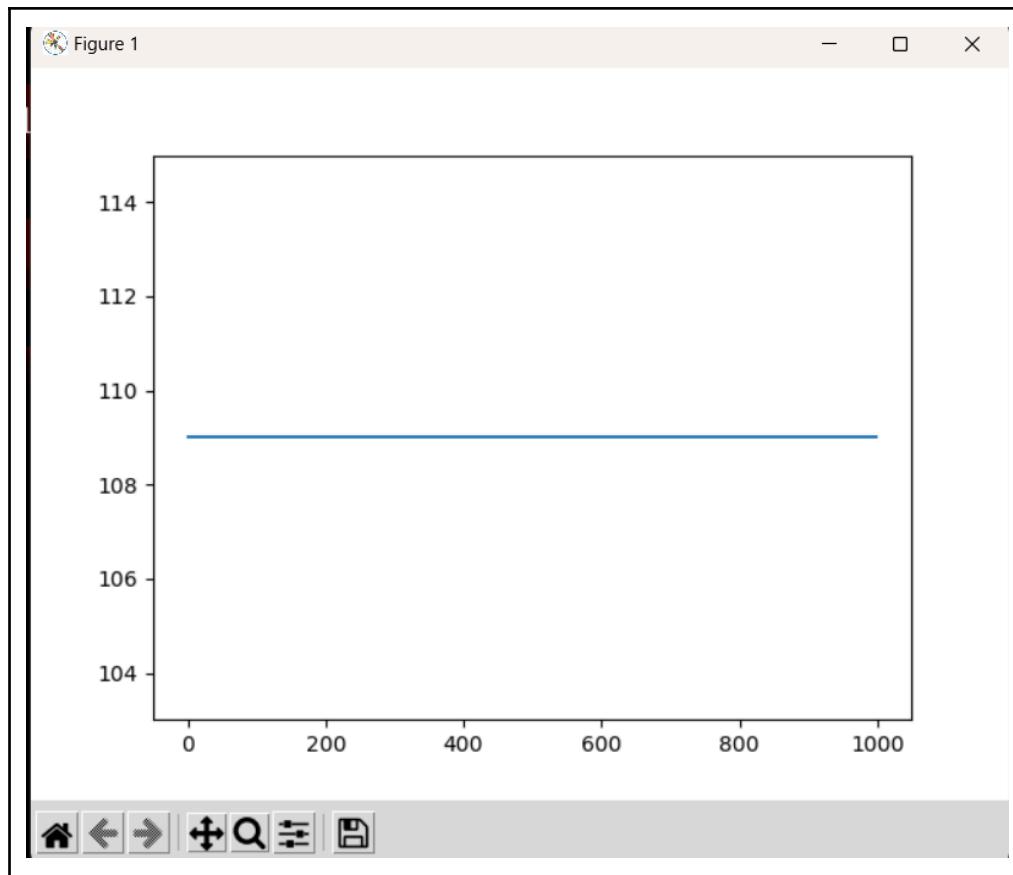
e. Iterasi 2000





f. Iterasi 3000





4. Analisis

4.1 Steepest Ascent Hill-Climbing

Setelah melakukan eksperimen pada algoritma Stochastic Hill Climbing, dapat diperoleh hasil analisa bahwa :

- 1) Seberapa dekat algoritma ini bisa mendekati global optima dan mengapa hasilnya demikian?**

Algoritma Steepest Ascent Hill Climbing seringkali mengalami kesulitan dalam mencapai global optima karena terjebak di optimum lokal.

- 2) Bagaimana perbandingan hasil pencarian algoritma ini dengan algoritma local search yang lain?**

Algoritma ini kurang efektif dalam mencapai solusi yang baik terkhususnya pada ruang pencarian yang kompleks. Hal ini dikarenakan algoritma ini tidak punya mekanisme untuk menghindari jebakan lokal.

- 3) Bagaimana perbandingan durasi proses pencarian algoritma ini relatif terhadap algoritma lainnya?**

Durasi dari algoritma ini jauh lebih cepat dibanding lainnya karena algoritmanya hanya memiliki fokus pada solusi terbaik dalam neighbor.

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

Hasil dari algoritma ini konsisten karena akan selalu memilih solusi yang terbaik dari setiap iterasi yang dilakukan.

4.2 Hill-Climbing with Sideways Move

Sideways Move Hill-Climbing adalah versi yang ditingkatkan dari metode hill climbing steepest ascent, dengan kemampuan ekstra untuk bergerak ke samping ketika tidak ada peningkatan yang bisa ditemukan. Pendekatan ini membantu algoritma mendekati atau mencapai hasil terbaik dengan mengatasi masalah area datar dalam ruang pencarian. Jika dibandingkan dengan Stochastic Hill Climbing yang biasanya lebih cepat namun sering kali hanya menemukan solusi yang cukup baik, Sideways Move Hill-Climbing lebih cermat dan memiliki kesempatan lebih besar untuk mencapai hasil yang optimal karena tidak terburu-buru dalam menyelesaikan pencarian.

Dalam hal durasi, Sideways Move Hill-Climbing mungkin lebih lambat daripada Stochastic Hill Climbing karena lebih teliti dalam mengevaluasi setiap langkah yang mungkin. Meskipun mungkin memakan waktu lebih lama, algoritma ini bisa lebih cepat dengan beberapa ketentuan yang diatur pengguna.

Keuntungan lain dari Sideways Move Hill-Climbing adalah konsistensi hasilnya. Dibandingkan dengan algoritma yang menggunakan pendekatan acak, Sideways Move cenderung menghasilkan hasil yang lebih konsisten karena lebih mengandalkan analisis yang sistematis. Ini membuat Sideways Move Hill-Climbing menjadi pilihan yang lebih stabil.

Secara keseluruhan, Sideways Move Hill-Climbing menawarkan keseimbangan yang baik antara kecepatan dan kemampuan untuk mencapai solusi yang optimal. Dengan kemampuan untuk melanjutkan pencarian melalui pergerakan ke samping dan hasil yang lebih konsisten, algoritma ini adalah pilihan yang tepat dibanding hill-climbing lainnya.

4.3 Random Restart Hill-Climbing

Setelah melakukan eksperimen pada algoritma Random Restart Hill Climbing, dapat diperoleh hasil analisa bahwa : tentu varian dari hill climbing ini menambahkan suatu mekanisme untuk mengatasi jebakan pada optimum lokal. Dibandingkan dengan simulated annealing dan genetic algorithm, algoritma ini menawarkan cara yang efektif dalam meningkatkan hill climbing tanpa mengubah mekanisme dasar hill climbing. Namun, Random Restart ini dapat dikatakan masih kurang optimal dalam mengatasi permasalahan yang kompleks. Selain itu, random restart juga memakan durasi yang cukup besar karena harus beberapa kali restart. Namun, random restart ini tentu lebih cepat dibandingkan algoritma lain seperti genetic algorithm. Dari segi konsistensi, algoritma ini jauh lebih konsisten dibandingkan hill climbing biasa karena penggunaan restart memungkinkan untuk mencari poin start yang berbeda untuk ruang pencarian solusi yang berbeda juga.

4.4 Stochastic Hill Climbing

Setelah melakukan eksperimen pada algoritma Stochastic Hill Climbing, dapat diperoleh hasil analisa bahwa :

- 1) Seberapa dekat algoritma ini bisa mendekati global optima dan mengapa hasilnya demikian?**

Algoritma Stochastic Hill Climbing sulit atau memiliki keterbatasan untuk mendekati global optima dikarenakan algoritma ini hanya mempertimbangkan neighbor terbaik yang ditemukan pada setiap iterasi. Pada akhirnya, algoritma ini akan menemukan solusi yang baik namun tidak maksimal.

- 2) Bagaimana perbandingan hasil pencarian algoritma ini dengan algoritma local search yang lain?**

Apabila dibandingkan dengan algoritma varian hill climbing yang lain seperti Hill Climbing Steepest Ascent dan Hill Climbing Sideway Moves, Stochastic Hill Climbing lebih cepat karena tidak melakukan pengecekan ke segala kemungkinan successor Magic Cube.

- 3) Bagaimana perbandingan durasi proses pencarian algoritma ini relatif terhadap algoritma lainnya?**

Durasi dari algoritma Stochastic Hill Climbing akan lebih cepat dibandingkan algoritma Simulated Annealing dan Genetic Algorithm karena cara kerja algoritma ini yang lebih sederhana dan hanya perlu memikirkan neighbor dari setiap iterasi saja.

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

Hasil dari algoritma Stochastic Hill Climbing tidak konsisten apabila dilakukan berulang kali eksperimen. Hal ini dikarenakan algoritma ini mengambil nilai secara acak pada setiap iterasi, sehingga keakuratan pengambilan berdasarkan nilai awal dan nilai acak yang dipilih pada setiap iterasi.

4.5 Simulated Annealing

Setelah melakukan eksperimen pada algoritma Stochastic Hill Climbing, dapat diperoleh hasil analisa bahwa :

- 1) Seberapa dekat algoritma ini bisa mendekati global optima dan mengapa hasilnya demikian?**

Simulated annealing ini lebih efektif dalam mendekati global optima dikarenakan algoritma Simulated Annealing ini menggunakan sistem probabilitas yang memungkinkan untuk mengambil solusi yang lebih buruk. Efektivitas ini dipengaruhi oleh laju penurunan suhu (cooling schedule) yang digunakan.

- 2) Bagaimana perbandingan hasil pencarian algoritma ini dengan algoritma local search yang lain?**

Apabila dibandingkan dengan algoritma varian hill climbing, simulated annealing ini tentu akan memberikan hasil pencarian yang lebih baik, terkhususnya untuk yang memiliki banyak solusi lokal.

- 3) Bagaimana perbandingan durasi proses pencarian algoritma ini relatif terhadap algoritma lainnya?**

Tentu algoritma ini lebih lama saat melakukan pencarian karena membutuhkan proses tambahan saat terjadinya penurunan suhu dan ditemukannya suatu solusi yang buruk.

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

Algoritma Simulated Annealing cenderung akan memberikan hasil yang konsisten apabila jumlah iterasi tinggi dan penggunaan rumus cooling schedule yang tepat.

4.6 Genetic Algorithm

Setelah melakukan eksperimen pada algoritma Genetic Algorithm, dapat diperoleh hasil analisa bahwa :

- 5) Seberapa dekat algoritma ini bisa mendekati global optima dan mengapa hasilnya demikian?**

Genetic Algorithm mampu mendekati global optima karena bekerja dengan pendekatan population dan crossover yang menggunakan informasi dari solusi terbaik yang ada.

- 6) Bagaimana perbandingan hasil pencarian algoritma ini dengan algoritma local search yang lain?**

Genetic Algorithm mampu mencari solusi yang lebih baik dengan pendekatan populasi yang lebih luas.

- 7) Bagaimana perbandingan durasi proses pencarian algoritma ini relatif terhadap algoritma lainnya?**

Durasi pencarian genetic algorithm lebih lama dibandingkan varian hill climbing dan simulated annealing, karena memerlukan cukup banyak operasi atau tahapan dari fitness function, selection, crossover, hingga mutation.

- 8) Seberapa konsisten hasil akhir yang didapatkan dari tiap-tiap eksperimen yang dilakukan?**

Konsistensi cenderung beragam bergantung pada probabilitas crossover dan mutasi.

III. KESIMPULAN DAN SARAN

A. Kesimpulan

Dalam dokumen tugas ini, telah diimplementasikan enam jenis algoritma *local search* dalam mencari solusi untuk *magic cube 5x5*. Dapat dilihat seberapa efektif algoritma Steepest Ascent Hill Climbing dalam menemukan puncak lokal dengan cepat. Algoritma ini mengevaluasi semua tetangga yang mungkin secara sistematis dan kemudian memilih *state* yang memiliki nilai evaluasi terbaik. Keunggulan utamanya adalah kemampuan untuk secara konsisten menemukan kondisi yang lebih baik di setiap langkah, yang menjadikannya pilihan yang baik dalam situasi di mana puncak lokal adalah tujuan yang cukup. Kelemahan dari kemampuan ini, bagaimanapun, adalah bahwa algoritma ini sering terhenti di puncak lokal dan tidak dapat mengeksplorasi di luar ketika semua tetangga memiliki nilai yang tidak optimal.

Untuk memperluas cakupan pencarian, Sideways Move Hill Climbing memungkinkan pergerakan netral dengan bergerak ke tetangga dengan nilai yang sama untuk menghindari jebakan pada kondisi datar. Meskipun ada batasan ketika berhadapan dengan puncak yang lebih tinggi dalam topologi fungsi yang kompleks, modifikasi ini meningkatkan kemampuan algoritma untuk analisis yang lebih luas.

Dengan menggunakan elemen acak, Random Restart Hill-Climbing mengatasi beberapa keterbatasan *Steepest Ascent* dengan mulai kembali pencarian dari titik acak setiap kali menemui kebuntuan. Strategi ini sangat meningkatkan kemungkinan algoritma untuk mencapai maksimum global, memberikan jalan baru setiap kali kondisi stagnan muncul. Metode ini lebih fleksibel, tetapi membutuhkan lebih banyak waktu dan iterasi, yang mungkin tidak efektif untuk semua jenis masalah.

Dibandingkan dengan Steepest Ascent Hill Climbing, Simulated Annealing menawarkan pendekatan yang lebih fleksibel karena konsep "suhu" yang menurun secara bertahap, memungkinkan pencarian solusi yang lebih buruk untuk sementara waktu. Ini menunjukkan bahwa algoritma ini memiliki kemampuan untuk keluar dari puncak lokal dan bergerak ke solusi yang pada awalnya tampaknya tidak menguntungkan. Seiring dengan penurunan suhu, algoritma menjadi lebih konservatif, mengurangi kemungkinan menerima solusi yang lebih buruk, dan tetap stabil dalam pencarian maksimum global. Dengan fitur ini, Simulated Annealing sangat berguna untuk situasi dengan lanskap yang rumit dan berbukit-bukit di mana risiko tinggi.

Genetic Algorithm menggunakan prinsip evolusi biologis untuk mengeksplorasi ruang solusi dengan menggunakan operasi seperti seleksi, crossing, dan mutasi. Dengan memulai dari populasi solusi, algoritma ini secara iteratif mencoba menghasilkan generasi solusi yang lebih baik. Operasi crossing memungkinkan kombinasi fitur dari dua solusi, dan mutasi memungkinkan variasi acak yang dapat membawa solusi ke area ruang pencarian yang belum dijelajahi sebelumnya. Kemampuannya untuk menjelajah ruang pencarian yang luas dan menemukan solusi yang kuat terhadap berbagai kondisi masukan adalah keunggulan utama metode ini. Genetic Algorithm, di sisi lain, membutuhkan banyak waktu dan sumber daya komputasi, terutama untuk populasi dan fungsi objektif yang kompleks.

Dalam memilih metode pencarian yang tepat untuk masalah *diagonal magic cube 5x5*, harus dipahami secara mendalam. Algoritma Simulated Annealing dan Genetic Algorithm bagus untuk mengatasi masalah yang rumit dan sering kali bisa menemukan solusi terbaik yang lebih luas dibandingkan metode pencarian sederhana. Namun, kedua algoritma ini juga membutuhkan banyak waktu dan daya komputer

yang besar. Lagipun, pada percobaan, Genetic Algorithm belum mendapatkan hasil yang baik. Agar bisa menggunakannya dengan efektif, kita perlu mengatur setting atau parameter algoritma dengan tepat dan harus benar-benar mengerti bagaimana algoritma tersebut bekerja dengan masalah yang akan kita selesaikan.

B. Saran

Dari penjelasan dan analisis dilakukan, dapat dilihat bahwa algoritma pencarian lokal seperti Steepest Ascent Hill Climbing, Sideways Move Hill Climbing, Random Restart Hill Climbing, Simulated Annealing, dan Genetic Algorithm masing-masing memiliki karakteristik dalam menyelesaikan masalah *diagonal magic cube*. Steepest Ascent Hill Climbing sangat efisien dalam mendekati solusi lokal terdekat tetapi seringkali terjebak di puncak lokal. Sideways Move Hill Climbing menambahkan kemampuan untuk melanjutkan pencarian di dataran datar, sedangkan Random Restart Hill Climbing meningkatkan peluang menemukan solusi global dengan memulai ulang pencarian dari titik yang berbeda.

Simulated Annealing dan Genetic Algorithm menawarkan strategi yang lebih canggih dan fleksibel dengan mengeksplorasi ruang solusi yang lebih luas dan memiliki potensi lebih besar untuk mencapai solusi global. Simulated Annealing memungkinkan eksplorasi ke solusi yang lebih buruk untuk menghindari terjebak pada maksimum lokal, sementara Genetic Algorithm menggunakan mekanisme seleksi alam untuk terus menghasilkan solusi yang semakin mendekati kondisi optimal melalui generasi.

Setiap algoritma memiliki kelebihan dan kekurangannya sendiri, termasuk dalam hal kebutuhan komputasi dan sumber daya. Dalam memilih algoritma yang tepat, penting untuk mempertimbangkan karakteristik masalah yang dihadapi dan sumber daya yang tersedia. Pemahaman yang baik tentang bagaimana setiap algoritma bekerja akan membantu dalam menyesuaikan pendekatan untuk mendapatkan hasil yang paling efektif. Selain itu, dalam beberapa kasus, kombinasi beberapa algoritma bisa menjadi strategi yang efektif untuk mendapatkan solusi yang optimal.

IV. LINK SOURCE CODE

Berikut ini merupakan tautan atau link source code (dalam bentuk github repository) :
https://github.com/Flame25/Tubes1_DAI

V. PEMBAGIAN TUGAS TIAP ANGGOTA KELOMPOK

Berikut ini merupakan pembagian tugas setiap anggota tim untuk Tugas Besar 1 Dasar Artificial Intelligence :

Nama Anggota Tim	Tugas yang Dilakukan Anggota
------------------	------------------------------

Anthony Bryant Gouw (18222033)	Merancang dan membuat Laporan bagian <i>objective function</i>
	Membuat dan merevisi kode Steepest Ascent, Sideways Move, dan Stochastic Hill-Climbing
	Membuat laporan bagian Analisis
	Membuat Laporan Penjelasan implementasi kode bagian Stochastic Hill-Climbing, Simulated Annealing dan Genetic Algorithm.
Christopher Richard Chandra (18222057)	Merancang dan membuat Laporan bagian <i>objective function</i>
	Membuat kode bagian <i>cube</i> dan visualisasi
	Membuat dan merevisi kode Simulated Annealing
	Membuat kesimpulan
Richie Leonardo (18222071)	Membuat dan merevisi kode Steepest Ascent, Genetic Algorithm
	Merancang dan membuat Laporan bagian <i>objective function</i>
	Membuat kode bagian <i>cube</i> dan visualisasi
	Merancang dan membuat Laporan bagian <i>objective function</i>
Josia Ryan Juliandy Silalahi (18222075)	Membuat dan merevisi kode Steepest Ascent, Sideways Move, dan Random Restart
	Membuat Laporan Penjelasan implementasi kode bagian Steepest Ascent, Sideways Move, dan Random Restart
	Membuat laporan bagian Hasil Visualisasi Eksperimen
	Membuat Laporan bagian deskripsi persoalan
	Membuat Saran
	Merancang dan membuat Laporan bagian <i>objective function</i>

VI. REFERENSI

Magic cube. (n.d.). Wikipedia. Retrieved October 1, 2024, from https://en.wikipedia.org/wiki/Magic_cube

Perfect Magic Cube -- from Wolfram MathWorld. (n.d.). Wolfram MathWorld. Retrieved November 2, 2024, from <https://mathworld.wolfram.com/PerfectMagicCube.html>

de Fermat, P. (n.d.). *MULTIMAGIE.COM - Perfect cubes.* Multimagie.com. Retrieved October 2, 2024, from <http://www.multimagie.com/English/Perfectcubes.htm>

Jain, S. (2024, August 22). *Local Search Algorithm in Artificial Intelligence.* GeeksforGeeks. Retrieved October 2, 2024, from <https://www.geeksforgeeks.org/local-search-algorithm-in-artificial-intelligence/>

Magic Cube -- from Wolfram MathWorld. (n.d.). Wolfram MathWorld. Retrieved October 2, 2024, from <https://mathworld.wolfram.com/MagicCube.html>

Objective Function - What Is Objective Function in LPP? , Applications, Examples. (n.d.). Cuemath. Retrieved October 2, 2024, from <https://www.cuemath.com/algebra/objective-function/>

Smolyakov, V. (2018, June 10). *Algorithms in C++.* Complete Search, Greedy, Divide and... | by Vadim Smolyakov. Towards Data Science. Retrieved October 2, 2024, from <https://towardsdatascience.com/algorithms-in-c-62b607a6131d>

Kosta. (2020). *Local Search Algorithms Magic Square Problem.* Github. Retrieved October 2, 2024, from <https://github.com/Artificial-Intelligence-kosta/Local-search-algorithms-Magic-Square-problem>

(n.d.). Local Search. Retrieved October 3, 2024, from https://www.cs.iusb.edu/~danav/teach/c463/6_local_search.html

An Optimal Cooling Schedule Using a Simulated Annealing Based Approach. (n.d.). Scientific Research Publishing. Retrieved October 3, 2024, from <https://www.scirp.org/journal/paperinformation?paperid=78834>

(n.d.). 781f16-3 [Autosaved]. Retrieved October 3, 2024, from https://procaccia.info/courses/15381f16/c_slides/781f16-2a.pdf

(n.d.). Order Crossover (OX): proposed by Davis[99] A kind of variation of PMX with a different repairing procedure Procedure: OX 1. Se. Retrieved October 3, 2024, from <https://mat.uab.cat/~alseda/MasterOpt/GeneticOperations.pdf>

(n.d.). Coding and minimizing a fitness function using the Genetic Algorithm. Retrieved October 3, 2024, from <https://nature.berkeley.edu/getz/dominance/GA/gads/gadsdemos/html/gafitness.html#5>

