

# **LAPORAN TUGAS BESAR 1**

## **IF3170 Intelelegensi Artifisial**

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



Disusun oleh:

Ahmad Naufal Ramadan (13522005)

Erdianti Wiga Putri Andini (13522053)

Bagas Sambega Rosyada (13522071)

Ahmad Mudabbir Arif (13522072)

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**

**INSTITUT TEKNOLOGI BANDUNG**

**BANDUNG**

**2024**

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>1</b>
<b>BAB I</b>	
<b>DESKRIPSI PERSOALAN.....</b>	<b>3</b>
<b>BAB II</b>	
<b>LANDASAN TEORI.....</b>	<b>5</b>
2.1. Algoritma Hill-climbing.....	5
3.1.1. Steepest Ascent Hill-climbing.....	5
3.1.2. Hill-climbing with Sideways Move.....	5
3.1.3. Random Restart Hill-climbing.....	6
3.1.4. Stochastic Hill-climbing.....	7
2.2. Algoritma Simulated Annealing.....	7
2.3. Genetic Algorithm.....	8
<b>BAB III</b>	
<b>PEMBAHASAN.....</b>	<b>10</b>
3.1. Pemilihan Objective Function.....	10
3.2. Penjelasan Implementasi Algoritma Local Search.....	11
3.2.1. Algoritma Hill-climbing.....	11
3.2.1.1 Steepest Ascent Hill-climbing.....	11
3.2.1.2 Hill-climbing with Sideways Move.....	14
3.2.1.3 Random Restart Hill-climbing.....	18
3.2.1.4 Stochastic Hill-climbing.....	22
3.2.2. Algoritma Simulated Annealing.....	24
3.2.3. Genetic Algorithm.....	28
<b>BAB IV</b>	
<b>HASIL EKSPERIMENT DAN ANALISIS.....</b>	<b>33</b>
4.1. Algoritma Hill-climbing.....	33
4.1.1. Steepest Ascent Hill-climbing.....	33
4.1.2. Hill-climbing with Sideways Move.....	36
4.1.3. Random Restart Hill-climbing.....	39
4.1.4. Stochastic Hill-climbing.....	42
4.2. Algoritma Simulated Annealing.....	45

4.3. Algoritma Genetic Algorithm.....	48
4.4. Analisis Perbandingan Algoritma.....	67
<b>BAB V</b>	
<b>KESIMPULAN DAN SARAN.....</b>	<b>71</b>
5.1. Kesimpulan.....	71
5.2. Saran.....	71
<b>DAFTAR PUSTAKA.....</b>	<b>72</b>
<b>LAMPIRAN.....</b>	<b>73</b>
Pembagian Tugas.....	73

# **BAB I**

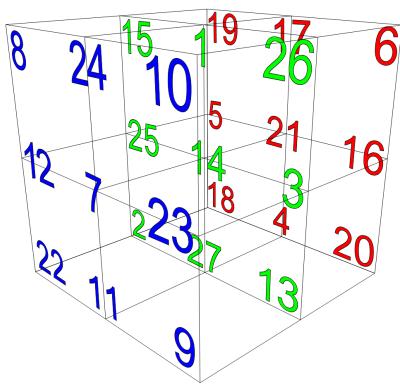
## **DESKRIPSI PERSOALAN**

Persoalan "Diagonal Magic Cube" mengharuskan kami untuk mengatur ulang angka pada sebuah kubus berukuran  $n \times n \times n$  dalam hal ini  $5 \times 5 \times 5$ , yang berisi angka dari 1 hingga  $n^3$  yang disusun secara acak, sehingga beberapa kriteria harus terpenuhi untuk mencapai definisi sebuah "magic cube". Kriteria-kriteria tersebut termasuk menjadikan jumlah angka di setiap baris, kolom, tiang, semua diagonal ruang pada kubus, dan diagonal pada setiap potongan bidang dari kubus, sama dengan sebuah nilai yang disebut *magic number*. *Magic number* ini adalah konstanta yang tidak harus merupakan angka dari 1 hingga  $n^3$ . *Magic number* ini didapatkan dari rumus berikut,

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

Untuk kubus berukuran 5, *magic number* yang dihasilkan adalah 315, yang berarti setiap baris, kolom, diagonal sisi, dan diagonal ruang harus memiliki jumlah sama dengan 315.

Berikut adalah ilustrasi dari potongan bidang yang ada pada suatu kubus berukuran 3 beserta 9 potongan bidangnya:



Gambar 1.1. Kubus 3 x 3 x 3

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

Gambar 1.2. Potongan bidang kubus  $3 \times 3 \times 3$

Pada tugas ini, kami akan menggunakan algoritma pencarian lokal (*local search algorithm*) untuk menemukan konfigurasi yang tepat dari kubus ini. Dalam setiap iterasi algoritma, kami diperbolehkan untuk menukar posisi dari dua angka dalam kubus. Dalam pendekatan menggunakan *genetic algorithm*, menukar posisi lebih dari dua angka dalam satu iterasi juga diperbolehkan meski tetap berfokus pada pertukaran dua angka.

Tujuan utama dari penyelesaian *diagonal magic cube* ini adalah mengoptimalkan posisi angka-angka tersebut agar semua kondisi yang didefinisikan untuk magic cube terpenuhi. Hal ini akan menghasilkan sebuah struktur yang setiap sumbu dan diagonalnya memiliki jumlah yang sama, yaitu magic number tersebut. Tugas ini menggabungkan konsep algoritma optimasi dengan pemahaman mendalam tentang struktur geometris dan matematis, menjadikannya sebuah studi kasus yang menarik dalam bidang matematika terapan dan ilmu komputer.

## **BAB II**

### **LANDASAN TEORI**

#### **2.1. Algoritma Hill-climbing**

##### **3.1.1. Steepest Ascent Hill-climbing**

Steepest Ascent Hill-climbing adalah metode pencarian solusi yang digunakan untuk mencapai solusi optimal secara global. Algoritma ini memulai pencariannya dari suatu kondisi awal yang dipilih secara acak dan melanjutkan iterasi sampai mencapai puncak global atau lokal. Menurut Tim Dosen Pengampu Mata Kuliah Intelelegensi Artifisial IF ITB pada tahun 2024, dalam proses algoritma ini, *successor* dihasilkan dengan menukarkan dua nilai dalam cube. *Successor* dengan biaya terendah dipilih sebagai tetangga berikutnya. Jika biaya tetangga ini lebih rendah dari biaya saat ini, maka ia menjadi kondisi baru. Proses ini terus berlanjut sampai tidak ditemukan lagi tetangga yang memiliki biaya lebih rendah, menandai berakhirnya pencarian dan menghasilkan kondisi terakhir sebagai optimum lokal, kecuali jika biayanya nol yang menandakan telah mencapai optimum global. Algoritma ini dapat mengalami kesulitan dalam mengatasi kondisi flat *local optimum* dan *shoulder*, di mana kondisi optimal global dengan biaya nol adalah tujuan utamanya.

##### **3.1.2. Hill-climbing with Sideways Move**

Hill-climbing with Sideways Move merupakan teknik pencarian solusi yang bertujuan untuk mencari solusi optimum global. Algoritma ini memulai dari suatu state awal yang secara acak ditentukan, dan proses iterasi terus berlangsung sampai ditemukan suatu *local optimum*. Dalam algoritma ini, semua *successor* dihasilkan dan *successor* dengan biaya paling rendah dipilih

sebagai tetangga. Jika biaya tetangga ini lebih rendah atau setara dengan biaya saat ini, maka tetangga tersebut akan diadopsi sebagai state baru. Proses ini berhenti ketika tidak ada lagi tetangga yang memiliki biaya lebih rendah atau sama, menghasilkan state terakhir sebagai *local optimum*, kecuali jika biayanya adalah nol yang menandakan tercapainya global optimum. Tujuan utama dari algoritma ini adalah mencapai kondisi di mana biaya mencapai nilai nol, yang menunjukkan pencapaian global optimum.

### 3.1.3. Random Restart Hill-climbing

Random Restart Hill Climbing adalah versi yang diperbaiki dari algoritma Hill Climbing, yang dirancang untuk menghindari permasalahan umum yang terjadi pada Steepest Ascent Hill Climbing yaitu terjebak pada optimum lokal. Algoritma ini memulai dengan menetapkan sebuah state awal secara acak. Dari situ, proses Hill Climbing dijalankan sesuai dengan langkah-langkah yang ada pada Steepest Ascent Hill Climbing. Bila algoritma mencapai titik optimum lokal di mana tidak ada peningkatan yang mungkin lagi, algoritma ini tidak berhenti; sebaliknya, Random Restart Hill Climbing memilih sebuah titik awal baru secara acak dan memulai lagi proses Steepest Ascent Hill Climbing dari awal. Proses ini diulang sesuai dengan parameter yang telah ditentukan, misalnya jumlah maksimal pengulangan restart. Jika algoritma menemukan optimum global sebelum batas restart tercapai, maka algoritma akan menghentikan pencarian dan mengembalikan nilai optimum global tersebut. Dengan memulai ulang dari berbagai titik awal secara acak, algoritma ini meningkatkan kesempatan untuk menemukan optimum global dan menghindari terjebak pada optimum lokal.

### 3.1.4. Stochastic Hill-climbing

Stochastic Hill-climbing adalah sebuah teknik dalam algoritma pencarian yang tidak memerlukan eksplorasi jalur lengkap untuk menemukan solusi. Algoritma ini merupakan variasi dari pencarian Hill-climbing, dengan pendekatan yang berbeda dari varian lainnya. Dalam konteks permasalahan diagonal magic cube, Stochastic Hill-climbing melaksanakan iterasi sejumlah  $n$  kali untuk menemukan solusi. Dalam setiap iterasi, algoritma ini secara acak menghasilkan sebuah state tetangga. Jika state tetangga tersebut menunjukkan peningkatan atau setidaknya keberlanjutan nilai (dalam hal ini, memiliki nilai cost yang lebih rendah atau sama dengan state saat ini), maka state tersebut akan menggantikan state saat ini sebagai state terbaik. Apabila state tetangga tidak menunjukkan peningkatan, iterasi berlanjut dan *successor* yang baru akan dibangkitkan secara acak. Proses ini akan terus berulang sampai jumlah iterasi yang ditentukan tercapai, dengan pembangkitan state tetangga yang dilakukan secara acak di setiap iterasi.

## 2.2. Algoritma Simulated Annealing

*Simulated annealing* adalah teknik pencarian yang menyertakan aspek stochastic hill-climbing dengan memperkenankan transisi ke state yang memiliki nilai objektif lebih rendah. Teknik ini dirancang untuk menghindari kemungkinan terjebak pada *local optimum* dengan memberi kemungkinan pada algoritma untuk menerima state dengan performa lebih buruk. Proses ini diizinkan asalkan probabilitas untuk menerima state yang lebih buruk masih tinggi. Sebagai mekanisme pengendali, probabilitas penerimaan state yang lebih buruk akan berkurang sejalan dengan bertambahnya jumlah langkah yang telah dilakukan. Probabilitas untuk memilih state yang lebih buruk dihitung menggunakan rumus:

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

di mana  $p$  adalah nilai peluang,  $e$  adalah basis logaritma natural,  $\Delta E$  adalah perbedaan nilai objektif antara state tetangga dan state saat ini, dan  $T$  adalah "temperatur" yang berkurang secara bertahap dengan setiap iterasi. Jika nilai  $p$  melebihi ambang batas tertentu, maka state dengan biaya lebih tinggi (performa lebih buruk) bisa diterima oleh program. Ini membantu menghindari kecenderungan algoritma untuk terjebak pada optimum lokal dan memungkinkan eksplorasi solusi yang lebih luas.

### 2.3. Genetic Algorithm

Genetic algorithm merupakan suatu metode yang terinspirasi oleh proses seleksi alam dalam biologi, seperti yang dijelaskan oleh Russell & Norvig pada tahun 2021. Algoritma ini menerapkan prinsip-prinsip acak dalam menghasilkan populasi berikutnya, dimana istilah 'populasi' digunakan untuk mendeskripsikan kumpulan *state* yang sedang dieksplorasi dalam satu siklus algoritma, dengan masing-masing state berjumlah  $k$ .

Dalam algoritma genetic, terdapat empat langkah utama yang diterapkan untuk menghasilkan populasi berikutnya dari populasi awal yang dianggap sebagai state saat ini:

1. Penghitungan *fitness function*. Fungsi ini mengukur seberapa 'fit' atau cocok setiap state dengan tujuan yang ingin dicapai. Dalam konteks diagonal magic cube, tujuan adalah untuk meminimalkan nilai objektif. Untuk memudahkan ini, nilai objektif dikalikan dengan nilai negatif sehingga nilai yang lebih kecil

(mendekati nol) menandakan state yang lebih diinginkan, sedangkan nilai yang lebih besar (lebih negatif) menunjukkan state yang lebih buruk.

2. Seleksi. Seleksi dilakukan menggunakan metode roulette wheel acak, di mana setiap state mengisi sebagian dari roda roulette. Bagian yang diisi oleh masing-masing state proporsional dengan 'kecocokannya', atau probabilitas terpilihnya berdasarkan fitness function, sehingga state dengan fitness yang lebih tinggi memiliki peluang lebih besar untuk terpilih untuk proses crossover.
3. Penyilangan (*Crossover*). Dari state yang terpilih, tiap state dibagi menjadi dua bagian pada titik crossover yang ditentukan secara acak. Misalnya, bagian A dari state pertama akan digabungkan dengan bagian B dari state lain. Proses ini diulang untuk semua pasangan state terpilih, menghasilkan dua state baru dari setiap pasangan.
4. Mutasi. Mutasi dilakukan untuk mengintroduksi variasi pada populasi dengan cara mengubah salah satu elemen dari state secara acak pada titik mutasi yang juga dipilih secara acak.

Proses-proses ini diulangi secara berkesinambungan hingga diperoleh state dengan nilai fitness function yang optimal atau mencapai nilai nol, yang mengindikasikan pencapaian tujuan yang diinginkan oleh algoritma.

## **BAB III**

### **PEMBAHASAN**

#### **3.1. Pemilihan Objective Function**

Untuk memahami kualitas dari sebuah susunan angka dalam Diagonal Magic Cube, dapat didefinisikan sebuah objective function. Objective function ini menggambarkan seberapa baik susunan angka dalam kubus dalam memenuhi aturan-aturan Diagonal Magic Cube. Nilai objective function didefinisikan sebagai nilai negatif dari jumlah pelanggaran terhadap aturan-aturan tersebut. Jika tidak ada pelanggaran, artinya semua aturan dipatuhi, nilai objective function akan mencapai 0. Sebaliknya, semakin banyak aturan yang dilanggar, semakin negatif nilainya. Aturan-aturan yang harus dipatuhi meliputi:

1. Jumlah angka dalam setiap baris harus sama dengan magic number.
2. Jumlah angka dalam setiap kolom harus sama dengan magic number.
3. Jumlah angka dalam setiap tiang harus sama dengan magic number.
4. Jumlah angka dalam setiap diagonal ruang harus sama dengan magic number.
5. Jumlah angka dalam setiap diagonal pada potongan ruang juga harus sama dengan magic number.

Sehingga, fungsi untuk menghitung objective function dari Diagonal Magic Cube dapat dituliskan sebagai berikut:

$$f(state) = - (f_{row}(state) + f_{col}(state) + f_{pillar}(state) + f_{sdiag}(state) + f_{pdiag}(state))$$

Pada Diagonal Magic Cube berukuran 5x5x5, terdapat 25 baris, 25 kolom, dan 25 tiang, sehingga totalnya menjadi 75 aturan. Selain itu, terdapat 4 diagonal utama di dalam kubus 3 dimensi yang juga harus mematuhi aturan. Setiap bidang dalam kubus

memiliki 2 diagonal. Terdapat 15 potongan bidang dalam magic cube berukuran 5x5x5, sehingga total diagonal pada potongan ruang adalah 30. Maka, total aturan yang harus dipenuhi dalam Diagonal Magic Cube berukuran 5x5x5 adalah 109 aturan.

Berdasarkan definisi dari objective function, nilai minimum dari objective function adalah -109. Kondisi ini terjadi ketika semua aturan dilanggar. Sementara itu, nilai maksimum dari objective function adalah 0. Kondisi ini terjadi ketika semua aturan dipatuhi dan kubus tersebut menjadi Diagonal Magic Cube.

### 3.2. Penjelasan Implementasi Algoritma Local Search

#### 3.2.1. Algoritma Hill-climbing

##### 3.2.1.1 Steepest Ascent Hill-climbing

Kelas SteepestAscent merupakan turunan dari kelas HillClimb yang mengimplementasikan algoritma Steepest Ascent Hill Climbing dalam penyelesaian Magic Cube. Kelas ini memeliki atribut dan method yang diturunkan dari kelas HillClimb dengan method solve yang di-*override*. Dalam algoritma ini, setiap langkah pencarian dilakukan dengan memilih *successor* terbaik di antara semua *successor* yang dihasilkan, sehingga selalu bergerak menuju solusi yang paling optimal.

```
class SteepestAscent : public HillClimb {  
public:  
    SteepestAscent();  
    SteepestAscent(const MagicFive& other);  
    ~SteepestAscent();  
    void solve() override;  
};
```

Fungsi untuk menyelesaikan permasalahan dengan menggunakan algoritma Steepest Ascent Hill Climbing terdapat pada method solve()

```
void SteepestAscent::solve() {  
    int currentScore = cube.objectiveFunction(); // current  
    cube's objective score  
  
    appendObjectiveFunction(currentScore);  
  
    bool improved;  
  
    // buat hitung iterasi  
  
    int count = 0;  
  
    do {  
        improved = false;  
  
        vector<vector<int>> successors = generateSuccessors();  
  
        vector<int> bestSuccessorData =  
        cube.matrixToList(cube.getData());  
  
        int bestScore = currentScore;  
  
        for (const auto& successor : successors) {  
            cube.setData(MagicFive::listToMatrix(successor)); //  
            generate suksesor  
  
            int newScore = cube.objectiveFunction();  
  
            if (newScore > bestScore) {  
                bestSuccessorData = successor;  
            }  
        }  
        count++;  
    } while (improved);  
}
```

```
        bestScore = newScore;
        improved = true;
    }
}

if (improved) {

    cube.setData(MagicFive::listToMatrix(bestSuccessorData)); // ambil suksesor best value
    currentScore = bestScore;
    // cout << "Objective function: " << currentScore << endl;
    count++;
}

appendObjectiveFunction(currentScore);

} while (improved); // lanjut kalo ada tetangga yang better

// result objective function
std::cout << "Final objective function: " << currentScore << std::endl;
std::cout << "Total iterations: " << count << std::endl;

// optimal global or not
if (currentScore == 0) {
```

```
        std::cout << "Optimal solution found." << std::endl;
    }
else {
    std::cout << "Optimal solution not found." << std::endl;
}
}
```

Pada fungsi di atas, setiap iterasi akan dibangkitkan semua *successor* dari state saat ini kemudian akan dipilih *successor* dengan nilai yang lebih baik dengan nilai saat ini berdasarkan atribut *improved* yang menunjukkan apakah akan pindah ke neighbor *successor* atau tidak. Algoritma akan berhenti ketika tidak ada lagi *successor* yang lebih baik dari saat ini.

### 3.2.1.2 Hill-climbing with Sideways Move

Kelas *SidewaysMove* merupakan turunan dari kelas *HillClimb* yang mengimplementasikan algoritma Hill Climbing with Sideways Move dalam penyelesaian Magic Cube. Kelas ini memiliki atribut dan method yang diturunkan dari kelas *HillClimb* dengan tambahan atribut *maxSidewaysMove*, yaitu parameter untuk maksimum sideways move selama pencarian solusi dan method *solve* yang di-*override*.

```
class SidewaysMove : public HillClimb {
public:
    SidewaysMove(int maxSidewaysMoves);
    SidewaysMove(const      MagicFive&      other,      int
maxSidewaysMoves);
    ~SidewaysMove();
    void solve() override;
```

```
private:  
    int maxSidewaysMoves;  
};
```

Fungsi untuk menyelesaikan permasalahan dengan menggunakan algoritma Steepest Ascent Hill Climbing terdapat pada method solve()

```
void SidewaysMove::solve() {  
    int currentScore = cube.objectiveFunction();  
    appendObjectiveFunction(currentScore);  
    int sidewaysMoves = 0;  
    bool improved;  
  
    // buat hitung iterasi  
    int count = 0;  
  
    do {  
        improved = false;  
        std::vector<std::vector<int>> successors =  
        generateSuccessors();  
        std::vector<int> bestSuccessorData =  
        cube.matrixToList(cube.getData());  
        std::vector<int> bestSidewayData =  
        cube.matrixToList(cube.getData());  
        int bestScore = currentScore;  
        int bestSidewayScore = currentScore;
```

```
        for (const auto& successor : successors) {

            cube.setData(MagicFive::listToMatrix(successor)); // generate suksesor

            int newScore = cube.objectiveFunction();

            if (newScore > bestScore) {

                bestSuccessorData = successor;
                bestScore = newScore;

                sidewaysMoves = 0; // reset sideways move count
                kalau ada yang lebih baik

                improved = true;

            }

            else if (newScore == currentScore && sidewaysMoves < maxSidewaysMoves) {

                bestSidewayData = successor;
                bestSidewayScore = newScore;

            }

        }

        if (improved) {

            cube.setData(MagicFive::listToMatrix(bestSuccessorData)); // ambil suksesor best value

            currentScore = bestScore;

            // std::cout << "Objective function: " << currentScore << std::endl;

            count++;

        }

    }

}
```

```
        }

        else if (sidewaysMoves < maxSidewaysMoves) {

            improved = true;

            cube.setData(MagicFive::listToMatrix(bestSidewayData)); // ambil
            suksesor best value

            currentScore = bestSidewayScore;

            count++;

            sidewaysMoves++;

        }

        appendObjectiveFunction(currentScore);

        // cek apakah sudah mencapai batas sideways moves

        if (sidewaysMoves >= maxSidewaysMoves) {

            std::cout << "Reached maximum sideways moves limit of "
            << maxSidewaysMoves << std::endl;

            break;

        }

    } while (improved); // lanjut kalo ada tetangga yang better
    dan belum mencapai batas sideways moves

    // result objective function

    std::cout << "Final objective function: " << currentScore <<
    std::endl;

    std::cout << "Total iterations: " << count << std::endl;
```

```
// optimal global or not  
  
if (currentScore == 0) {  
  
    std::cout << "Optimal solution found." << std::endl;  
  
}  
  
else {  
  
    std::cout << "Optimal solution not found." << std::endl;  
  
}  
  
}
```

Pada fungsi di atas, setiap iterasi akan dibangkitkan semua *successor* dari state saat ini kemudian akan dipilih *successor* dengan nilai yang lebih baik dengan nilai saat ini berdasarkan atribut *improved* yang menunjukkan apakah akan pindah ke neighbor *successor* atau tidak. Jika nilai dari *successor* sama dengan nilai saat ini akan dilanjutkan pencarian ke iterasi selanjutnya dengan batas maksimum iterasi adalah *maxSidewaysMove*. Algoritma akan berhenti ketika mencapai batas maksimum sideways move.

### 3.2.1.3 Random Restart Hill-climbing

Kelas RandomRestart merupakan turunan dari kelas HillClimb yang mewakili algoritma Random Restart Hill-Climbing untuk menyelesaikan permasalahan Magic Cube. Kelas ini memiliki atribut yang sama pada kelas HillClimb, namun dengan atribut tambahan berupa *max\_restart* sebagai parameter maksimum restart untuk algoritma ini dan method *solve* yang merupakan *override* dari method *solve* di kelas HillClimb.

```
class RandomRestart : public HillClimb {  
  
    private:  
  
        int max_restart;  
  
    public:  
  
        RandomRestart();  
  
        RandomRestart(const MagicFive& other, int max_restart);  
  
        void solve() override;  
  
};
```

Fungsi untuk menyelesaikan permasalahan dengan menggunakan algoritma Random Restart Hill Climbing terdapat pada method solve()

```
void RandomRestart::solve() {  
  
    int best_objective = cube.objectiveFunction();  
  
    appendObjectiveFunction(best_objective);  
  
    vector<vector<int>> best_data = cube.getData();  
  
    int iterations = 0;  
  
    int restarts = 0;  
  
  
    while (restarts < this->max_restart) {  
  
        int currentScore = cube.objectiveFunction(); // current  
        cube's objective score  
  
        bool improved;  
  
  
        do {  
  
            improved = false;
```

```
vector<vector<int>> successors =  
generateSuccessors();  
  
vector<int> bestSuccessorData =  
cube.matrixToList(cube.getData());  
  
int bestScore = currentScore;  
  
for (const auto& successor : successors) {  
  
    cube.setData(MagicFive::listToMatrix(successor));  
    // generate suksesor  
  
    int newScore = cube.objectiveFunction();  
  
    if (newScore > bestScore) {  
  
        bestSuccessorData = successor;  
  
        bestScore = newScore;  
  
        improved = true;  
  
    }  
}  
  
if (improved) {  
  
    cube.setData(MagicFive::listToMatrix(bestSuccessorData));      //  
    ambil suksesor best value  
  
    currentScore = bestScore;  
}  
  
appendObjectiveFunction(currentScore);  
iterations++;
```

```
        } while (improved); // lanjut kalo ada tetangga yang
better

    cout << "Restart " << restarts << ":" << iterations << "
iterations with objective function: " << currentScore << endl;

    if (currentScore > best_objective) {

        best_objective = currentScore;
        best_data = cube.getData();
    }

    // Restart the cube with random values
    cube = MagicFive();
    restarts++;

    if (best_objective == 0) {

        cout << "Solution found after " << restarts << "
restarts." << endl;
        break;
    }
}

cube.setData(best_data);
if (best_objective != 0) {
```

```
        cout << "Best solution found after " << this->max_restart  
<< " restarts." << endl;  
  
        cout << "Objective function: " << best_objective << endl;  
    }  
}
```

Pada fungsi di atas, algoritma dilakukan dengan menggunakan loop yang dijalankan hingga jumlah maksimum restart (`max_restart`) tercapai atau solusi optimal ditemukan. Setiap kali restart akan dilakukan steepest ascent algorithm untuk mencari solusi, jika stuck di *local optimum* akan dilakukan restart dengan *initial state* baru secara acak.

#### 3.2.1.4 Stochastic Hill-climbing

Kelas `Stochastic` merupakan turunan dari `HillClimb` yang mengimplementasikan algoritma Stochastic Hill Climbing, di mana pembangkitan *successor* dilakukan secara random dan akan dipilih *successor* yang lebih baik. Atribut pada kelas ini diturunkan oleh kelas `HillClimb` dengan method `solve` yang merupakan *override* dari kelas `HillClimb`.

```
class Stochastic : public HillClimb {  
  
public:  
  
    Stochastic();  
  
    Stochastic(const MagicFive& other);  
  
    void solve() override;  
};
```

Fungsi untuk menyelesaikan permasalahan dengan menggunakan algoritma Stochastic Hill Climbing terdapat pada method `solve()`

```
void Stochastic::solve() {  
    int iterations = 0;  
  
    int current_obj = cube.objectiveFunction();  
  
    appendObjectiveFunction(current_obj);  
  
    while (iterations < 500000) {  
  
        vector<vector<int>> current_cube = cube.getData();  
  
        current_obj = cube.objectiveFunction();  
  
        vector<vector<int>> random_successor_data =  
        generateRandomSuccessor();  
  
        MagicFive random_successor =  
        MagicFive(random_successor_data);  
  
        int random_obj = random_successor.objectiveFunction();  
  
        if (random_obj > current_obj) {  
            // cout << "Objective function: " << random_obj <<  
            endl;  
  
            cube.setData(random_successor_data);  
        }  
  
        appendObjectiveFunction(current_obj);  
  
        iterations++;  
  
        if (current_obj == 0) {  
    }
```

```
        cout << "Solution found in " << iterations << "
iterations." << endl;

        break;

    }

}

cout << "Solution found in " << iterations << " iterations."
<< endl;

}
```

Pada fungsi di atas, algoritma diimplementasikan mirip dengan algoritma Hill-Climbing, namun dengan pembangkitan *successor* secara random dan dipilih *successor* yang lebih baik. Algoritma ini berlangsung hingga mencapai solusi optimal atau hingga batas iterasi yaitu 500.000 iterasi, dengan pemilihan angka iterasi ini berdasarkan eksperimen bahwa menambahkan jumlah iterasi tidak mempengaruhi hasil akhir.

### 3.2.2. Algoritma Simulated Annealing

Kelas SimulatedAnnealing merupakan kelas yang menerapkan algoritma Simulated Annealing untuk menyelesaikan permasalahan Magic Cube. Kelas ini merupakan turunan dari kelas Solver yang memiliki atribut kubus MagicCube itu sendiri, dan juga fungsi solve() yang dipanggil untuk memulai algoritma pemecahan memakai Simulated Annealing.

```
class SimulatedAnnealing : public Solver {

private:

    vector<float> acceptance_probabilities;

    float temperature;
```

```
    float cooling_rate;
    float acceptance_probability;
    double min_temperature;

public:
    SimulatedAnnealing();
    vector<float> getAcceptanceProbabilities();
    void appendAcceptanceProbability(float new_acceptance_probability);
    vector<vector<int>> generateRandomSuccessor();
    void solve() override;
    double decreaseTemperature(long iteration);
    double acceptanceProbability(int objCurrent, int objNeighbor,
        double currentTemperature);
};
```

Fungsi untuk menyelesaikan dengan menggunakan algoritma Simulated Annealing terdapat pada fungsi solve():

```
void SimulatedAnnealing::solve() {
    int objCurrent = cube.objectiveFunction();
    appendObjectiveFunction(objCurrent);
    double currentTemperature = this->temperature;
    int stuck = 0; // Definisi stuck adalah kalau objCurrent tidak berubah
    long iteration = 1;
    while (currentTemperature > this->min_temperature && objCurrent < 0)
{
```

```
vector<vector<int>> neighbor = generateRandomSuccessor();

int objNeighbor = MagicFive(neighbor).objectiveFunction();

double probability = acceptanceProbability(objCurrent,
objNeighbor, currentTemperature);

appendObjectiveFunction(objCurrent);

appendAcceptanceProbability((float)probability);

if (objNeighbor > objCurrent) {

    cube = MagicFive(neighbor);

    objCurrent = objNeighbor;

} else if (probability > acceptance_probability) {

    cube = MagicFive(neighbor);

    objCurrent = objNeighbor;

} else {

    stuck++;

}

currentTemperature = decreaseTemperature(iteration);

iteration++;

}

cout << "Objective function: " << objCurrent << " with steps: " <<
iteration << endl;

cout << "Stuck: " << stuck << endl;

if (objCurrent == 0) {

    cout << "Solution found." << endl;

} else {

    cout << "Solution not found." << endl;
```

```

    }
}

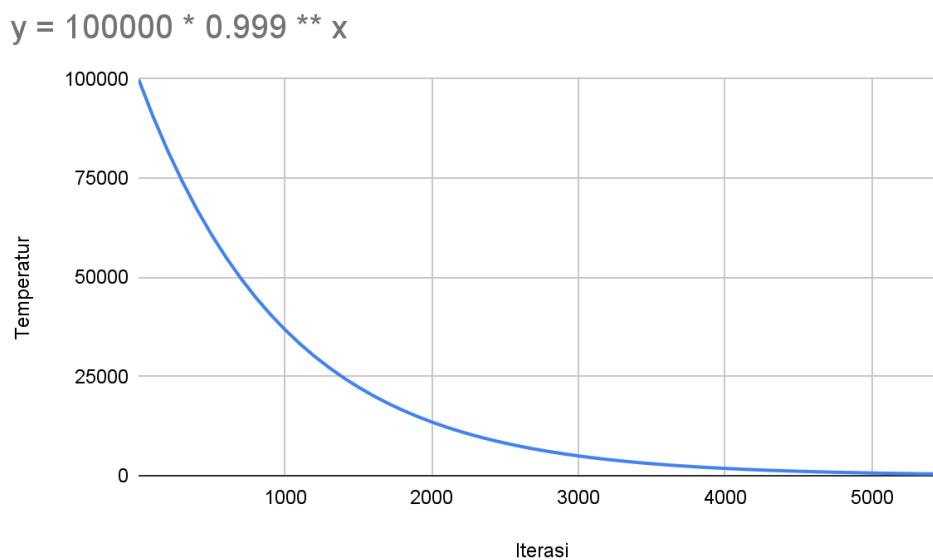
```

Pada fungsi di atas, fungsi `decreasingTemperature()` merupakan representasi dari fungsi *scheduling* pada temperatur. Pendefinisian fungsi penurunan nilai temperatur adalah,

$$T_{i+1} = T_0 \times \alpha^{i+1}$$

dengan  $i$  merupakan nilai iterasi saat ini,  $T_0$  merupakan nilai temperatur awal (sehingga nilainya konstan), dan  $T_i$  merupakan nilai temperatur saat ini. Penurunan nilai temperatur didasarkan pada fungsi eksponensial sehingga memungkinkan fungsi untuk menurun lebih lambat dibandingkan fungsi *schedule* temperatur secara linear, selama nilai  $0 < \alpha < 1$ .

Grafik penurunan nilai temperatur ditunjukkan oleh grafik berikut,



Grafik tersebut menunjukkan penurunan temperatur yang semakin landai seiring meningkatnya iterasi. Hal tersebut menyebabkan semakin banyak iterasi yang dapat dilakukan, terutama pada saat iterasi yang peluang untuk mengambil state yang nilai objektifnya lebih rendah sudah sangat kecil.

Pada program ini, nilai  $\alpha$  yang dipilih adalah 0.999 sehingga penurunan nilai temperatur lebih lambat dan lebih banyak iterasi yang dilakukan. Nilai temperatur awal ditentukan oleh rumus berikut,

$$T_0 = 10^{5 + (-1 \cdot \text{objective} / 20)}$$

Basis temperatur adalah  $10^5$ , dan semakin kecil (semakin jauh dari *goal state*) nilai objektif *state* awal kubus, maka nilai temperatur akan semakin besar, sehingga jumlah iterasi juga semakin tinggi. Untuk batas minimum nilai temperatur agar program berhenti adalah  $T_{min} = 10^{-325}$ . Nilai tersebut sangat kecil sehingga memungkinkan program tidak berhenti terlalu cepat.

### 3.2.3. Genetic Algorithm

Kelas Genetic merupakan turunan dari kelas Solver yang memiliki atribut dan method yang sama pada kelas Solver, dengan atribut tambahan berupa max\_objective\_function dan avg\_objective\_function, juga iterations, population\_size, dan threshold sebagai parameter.

```
class Genetic : public Solver {  
private:  
    vector<int> max_objective_functions;  
    vector<int> avg_objective_functions;  
    int iterations;
```

```
    int population_size;
    int threshold;

public:
    Genetic();
    Genetic(int iterations, int population_size, int threshold);
    Genetic(const MagicFive& other);
    Genetic(const MagicFive& other, int iterations, int population_size, int threshold);
    virtual ~Genetic();

    vector<int> getMaxObjectiveFunctions();
    vector<int> getAvgObjectiveFunctions();
    void setMaxObjectiveFunctions(const vector<int>& new_objective_functions);
    void setAvgObjectiveFunctions(const vector<int>& new_objective_functions);
    void appendMaxObjectiveFunction(int new_objective_function);
    void appendAvgObjectFunction(int new_objective_function);

    int getIterations();
    void setIterations(int iterations);
    int getPopulationSize();
    void setPopulationSize(int population_size);

    vector<MagicFive> generatePopulations();
```

```
    MagicFive selection(vector<MagicFive>& populations);

    MagicFive crossover(const MagicFive& parent1, const MagicFive&
parent2);

    MagicFive mutation(const MagicFive& child);

    void solve() override;

};
```

Fungsi untuk menyelesaikan permasalahan dengan menggunakan Genetic Algorithm terdapat pada method solve()

```
void Genetic::solve() {

    vector<MagicFive> populations = generatePopulations();

    bool found = false;

    for (int i = 0; i < iterations; i++) {
        vector<MagicFive> new_populations;

        int max_objective_function = -109;

        int avg_objective_function = 0;

        for (int j = 0; j < population_size; j++) {
            MagicFive parent1 = selection(populations);

            MagicFive parent2 = selection(populations);

            MagicFive child1 = crossover(parent1, parent2);

            MagicFive child2 = crossover(parent2, parent1);

            new_populations.push_back(mutation(child1));
            new_populations.push_back(mutation(child2));
        }

        populations = new_populations;
    }

    if (found) {
        cout << "Solution found!" << endl;
    } else {
        cout << "No solution found." << endl;
    }
}
```

```
        int child1_mutation = rand() % 100;

        if (child1_mutation < threshold) {

            child1 = mutation(child1);

        }

        new_populations.push_back(child1);

        int child2_mutation = rand() % 100;

        if (child2_mutation < threshold) {

            child2 = mutation(child2);

        }

        new_populations.push_back(child2);

        if      (child1.fitnessFunction()      ==      109      ||
child2.fitnessFunction() == 109) {

            found = true;

        }

        max_objective_function      =      max(max_objective_function,
max(child1.objectiveFunction(), child2.objectiveFunction()));

        avg_objective_function    +=   (child1.objectiveFunction()  +
child2.objectiveFunction());

    }

    avg_objective_function /= (population_size * 2);
```

```
appendMaxObjectiveFunction(max_objective_function);

appendAvgObjectFunction(avg_objective_function);

populations = new_populations;

if (found) {
    break;
}

}

MagicFive best_cube = *max_element(populations.begin(),
populations.end(), [](const MagicFive& a, const MagicFive& b) {
    return a.fitnessFunction() < b.fitnessFunction();
});

cube.setData(best_cube.getData());

cout << "Solution found in " << iterations << " iterations." <<
endl;

cout << "Objective function: " << best_cube.objectiveFunction()
<< endl;

}
```

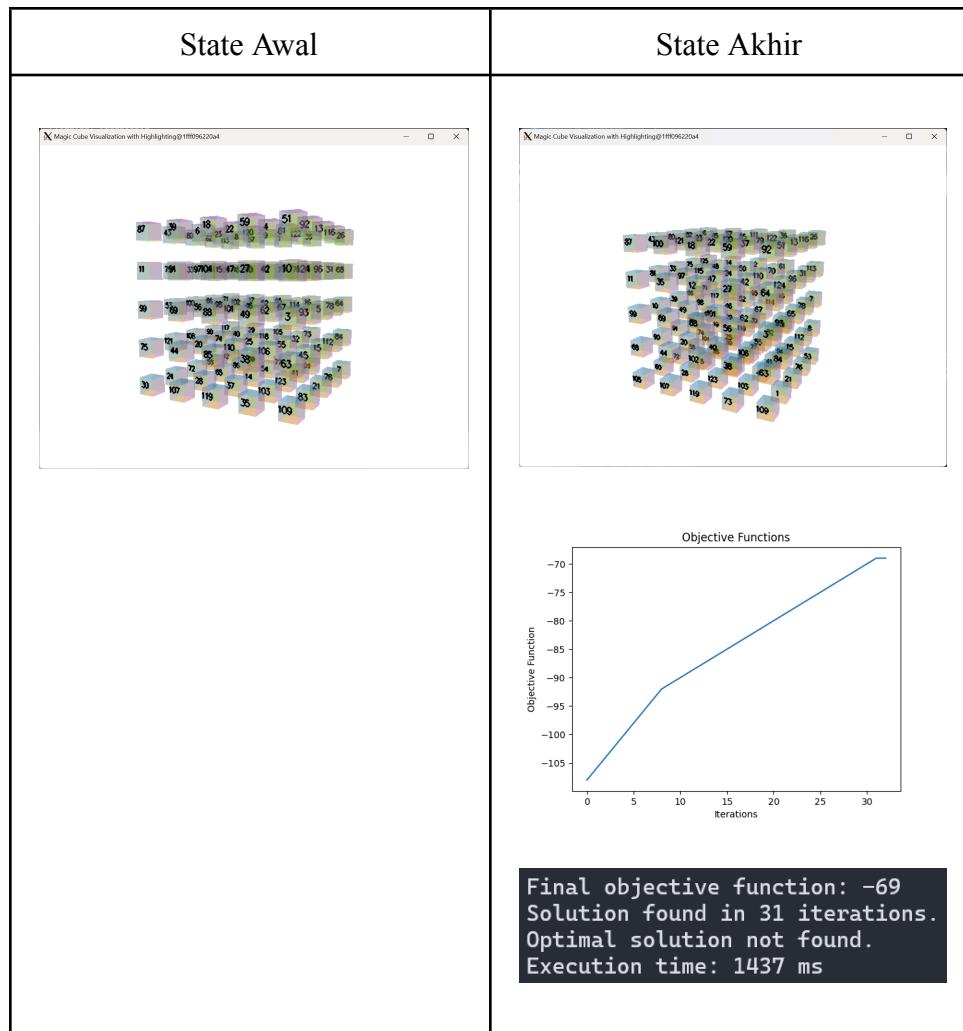
## BAB IV

### HASIL EKSPERIMENT DAN ANALISIS

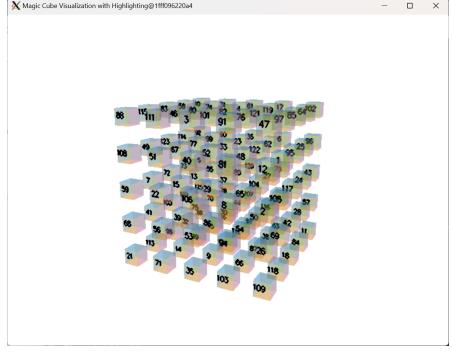
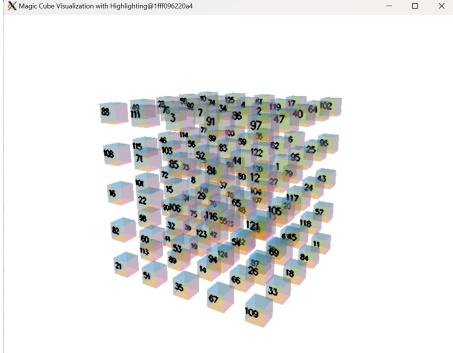
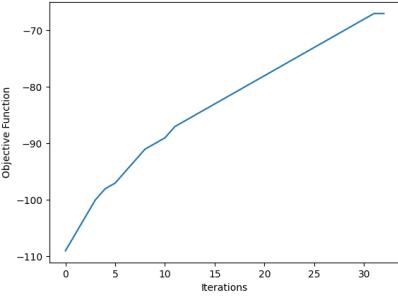
#### 4.1. Algoritma Hill-climbing

##### 4.1.1. Steepest Ascent Hill-climbing

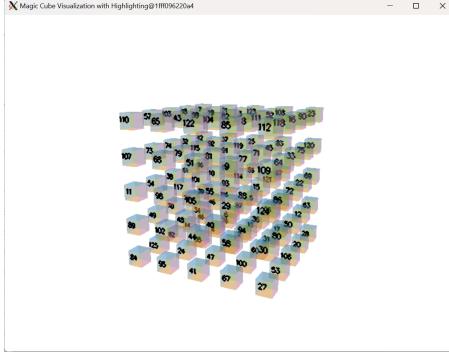
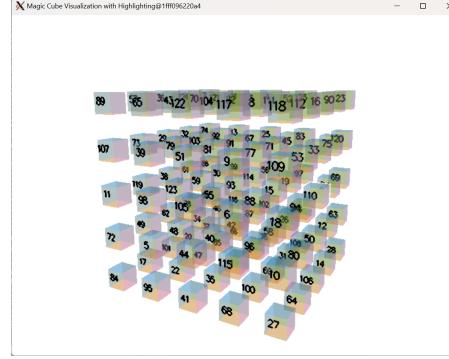
###### a. Percobaan 1



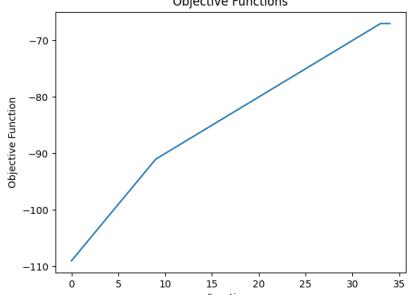
b. Percobaan 2

State Awal	State Akhir
	
	<p>Objective Functions</p>  <p>Final objective function: -67 Solution found in 31 iterations. Optimal solution not found. Execution time: 1468 ms</p>

c. Percobaan 3

State Awal	State Akhir
	

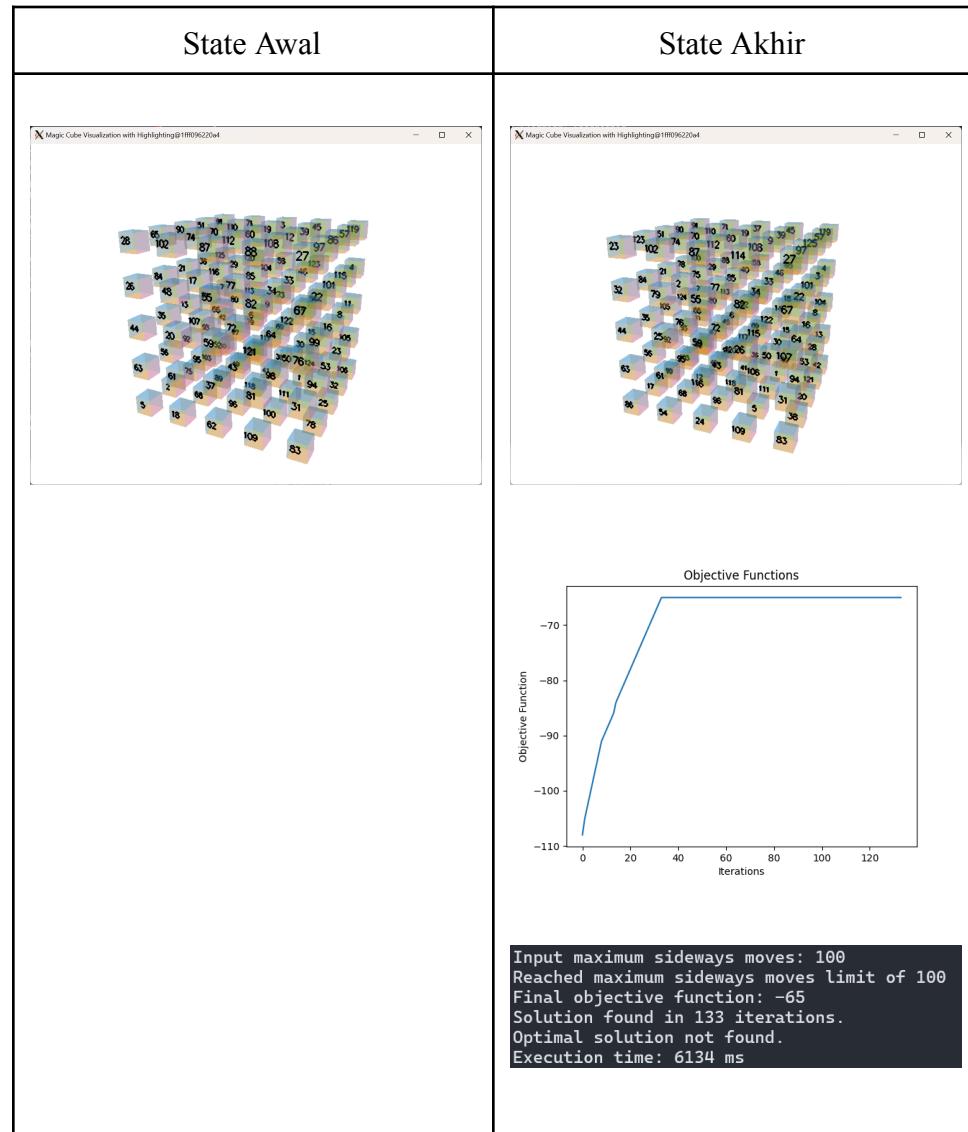
Objective Functions



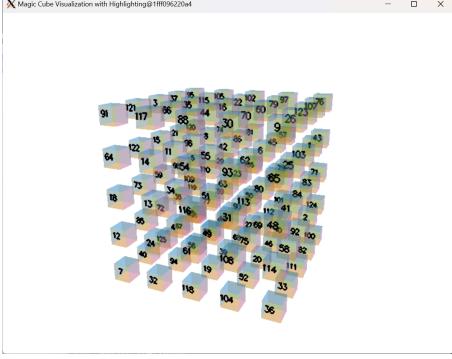
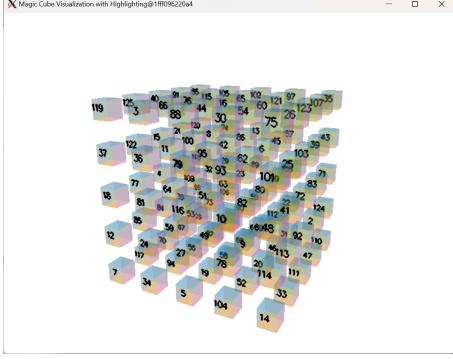
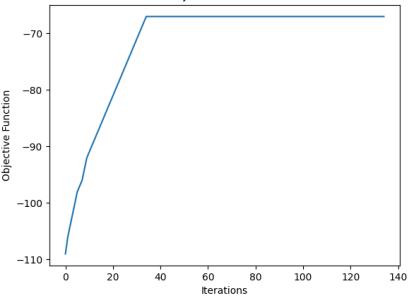
Final objective function: -67  
Solution found in 33 iterations.  
Optimal solution not found.  
Execution time: 1537 ms

#### 4.1.2. Hill-climbing with Sideways Move

##### a. Percobaan 1



b. Percobaan 2

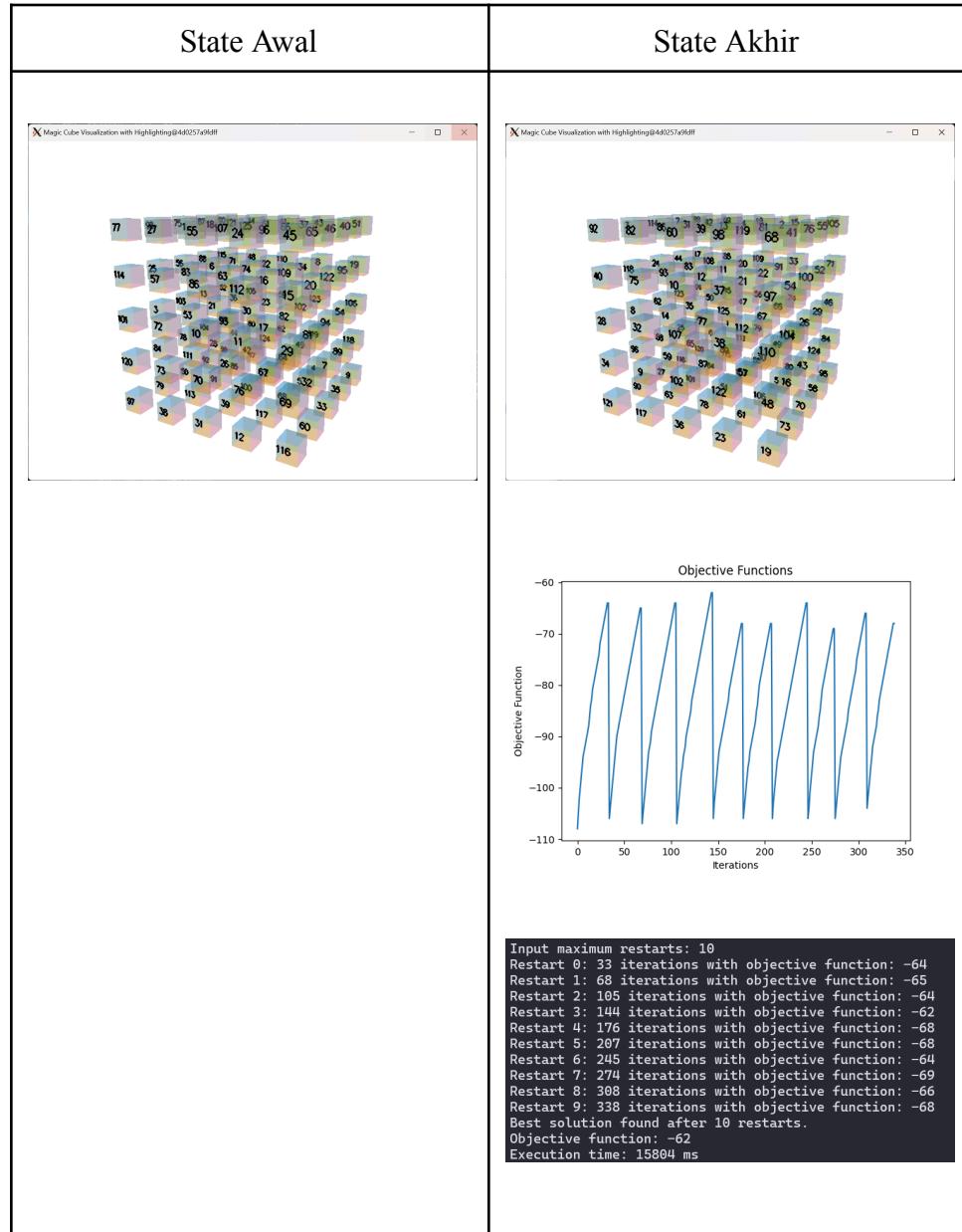
State Awal	State Akhir
	
	<p>Objective Functions</p>  <p>Iterations</p> <p>Object Function</p> <pre>Input maximum sideways moves: 100 Reached maximum sideways moves limit of 100 Final objective function: -67 Solution found in 134 iterations. Optimal solution not found. Execution time: 6129 ms</pre>

c. Percobaan 3

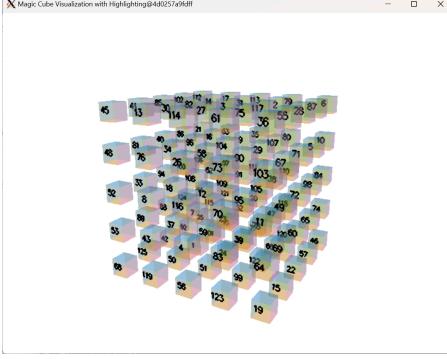
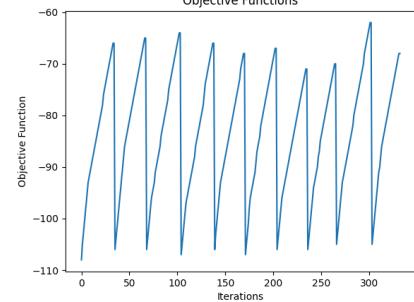
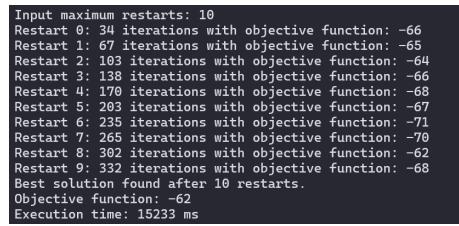
State Awal	State Akhir
	<p>Objective Functions</p> <p>Iterations</p> <p>Objective Function</p> <p>Input maximum sideways moves: 100 Reached maximum sideways moves limit of 100 Final objective function: -58 Solution found in 140 iterations. Optimal solution not found. Execution time: 6454 ms</p>

#### 4.1.3. Random Restart Hill-climbing

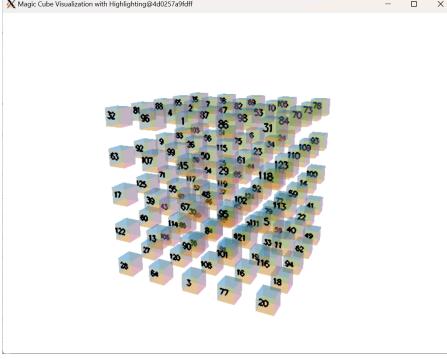
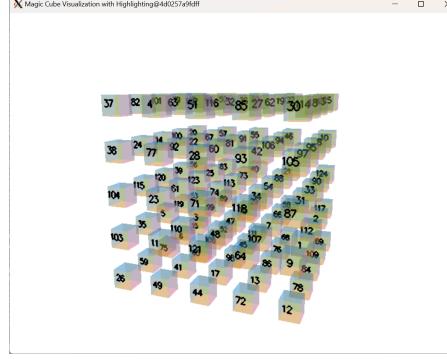
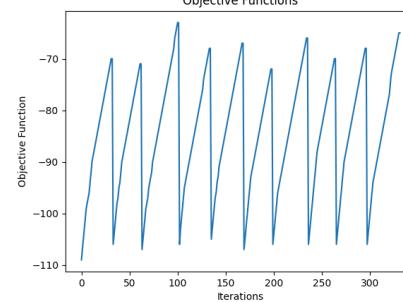
##### a. Percobaan 1



b. Percobaan 2

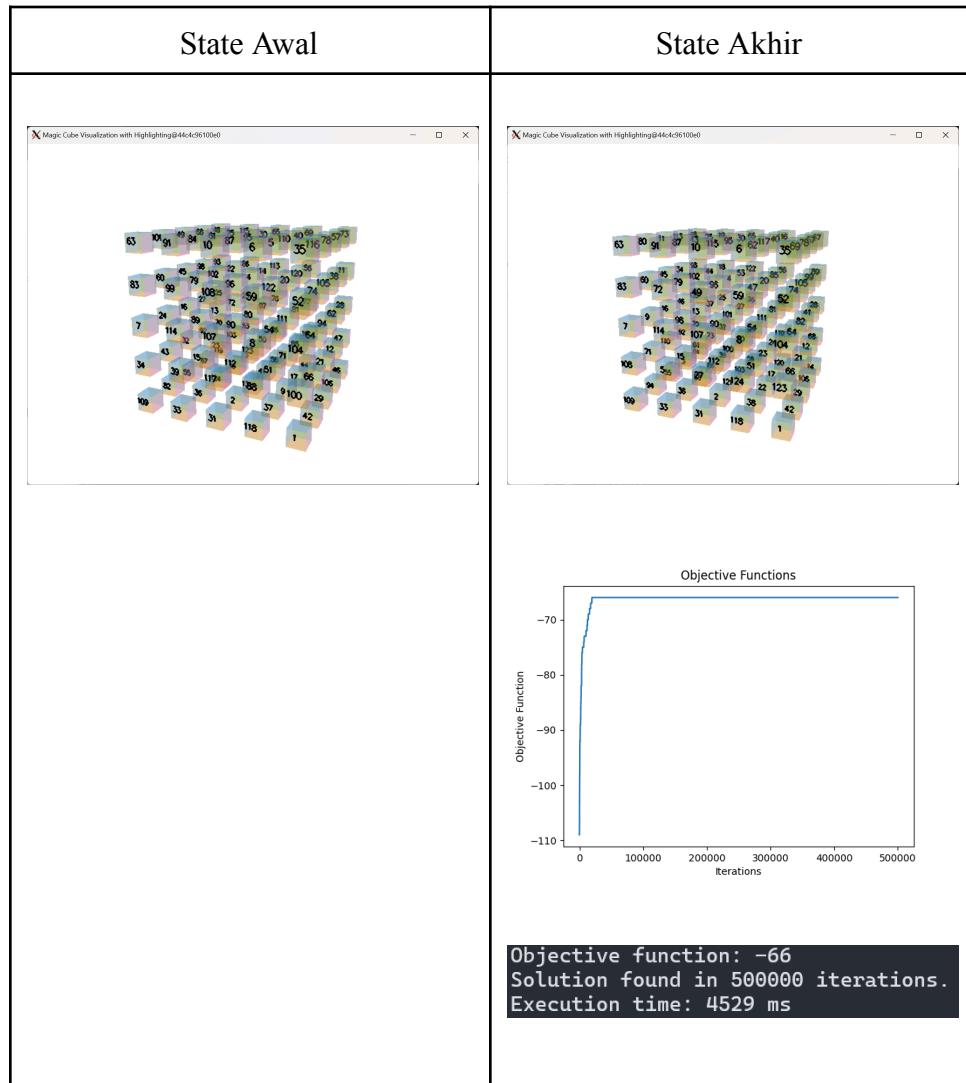
State Awal	State Akhir
	
	 <pre> Input maximum restarts: 10 Restart 0: 34 iterations with objective function: -66 Restart 1: 67 iterations with objective function: -65 Restart 2: 103 iterations with objective function: -64 Restart 3: 138 iterations with objective function: -66 Restart 4: 170 iterations with objective function: -68 Restart 5: 203 iterations with objective function: -67 Restart 6: 235 iterations with objective function: -71 Restart 7: 265 iterations with objective function: -70 Restart 8: 302 iterations with objective function: -62 Restart 9: 332 iterations with objective function: -68 Best solution found after 10 restarts. Objective function: -62 Execution time: 15233 ms </pre>

c. Percobaan 3

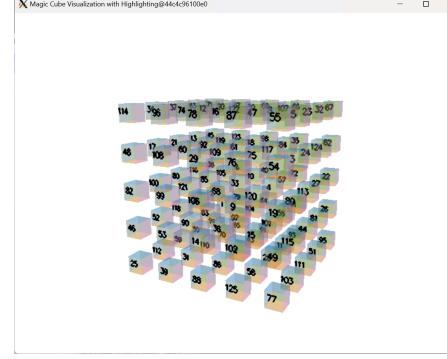
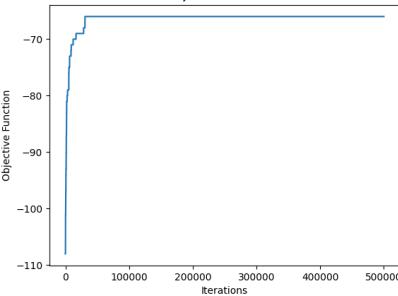
State Awal	State Akhir
	
 <pre>Input maximum restarts: 10 Restart 0: 32 iterations with objective function: -70 Restart 1: 62 iterations with objective function: -71 Restart 2: 101 iterations with objective function: -63 Restart 3: 134 iterations with objective function: -68 Restart 4: 168 iterations with objective function: -67 Restart 5: 198 iterations with objective function: -72 Restart 6: 235 iterations with objective function: -66 Restart 7: 264 iterations with objective function: -70 Restart 8: 296 iterations with objective function: -68 Restart 9: 331 iterations with objective function: -65 Best solution found after 10 restarts. Objective function: -63 Execution time: 15099 ms</pre>	

#### 4.1.4. Stochastic Hill-climbing

##### a. Percobaan 1



b. Percobaan 2

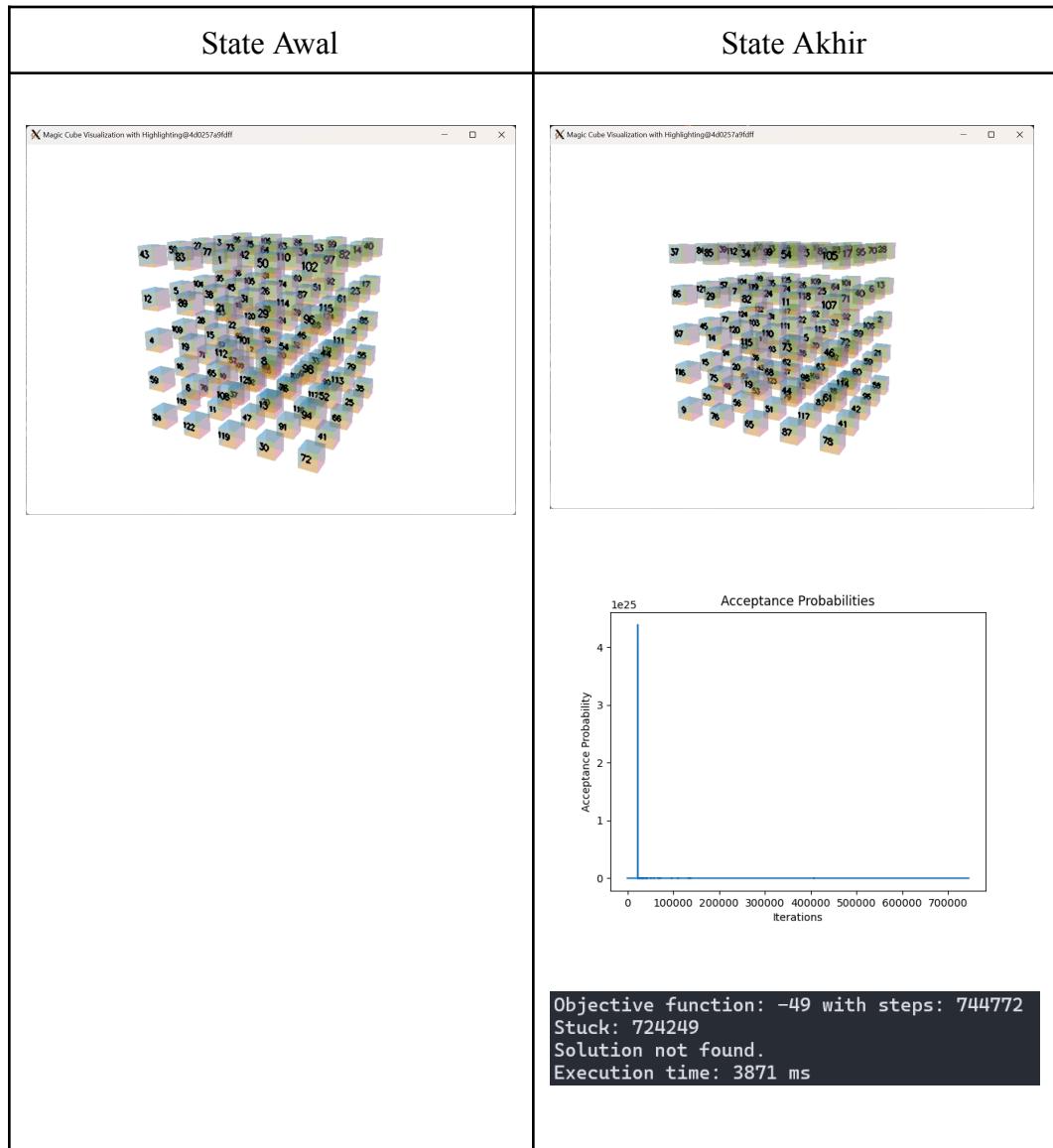
State Awal	State Akhir
	
	<p>Objective Functions</p>  <p>Objective Function</p> <p>Iterations</p> <p>Objective function: -66 Solution found in 500000 iterations. Execution time: 4806 ms</p>

c. Percobaan 3

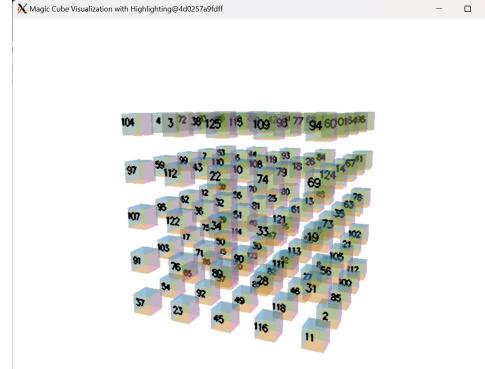
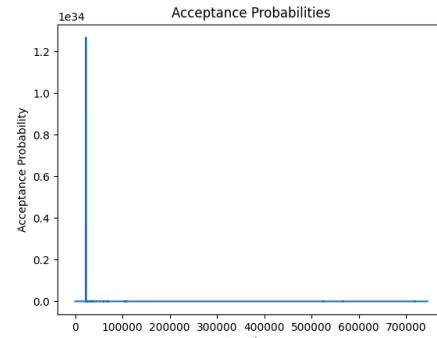
State Awal	State Akhir
	<p data-bbox="980 910 1432 1227">Objective Functions</p> <p data-bbox="980 1290 1432 1374">Objective function: -68 Solution found in 500000 iterations. Execution time: 4474 ms</p>

## 4.2. Algoritma Simulated Annealing

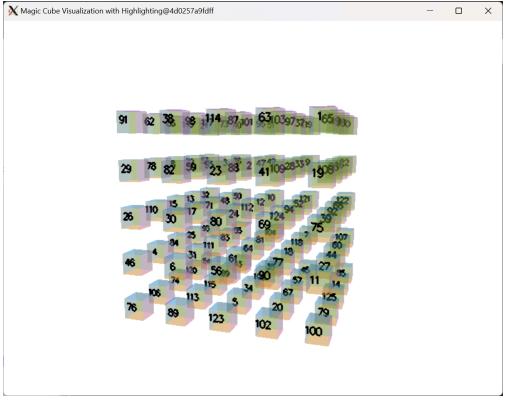
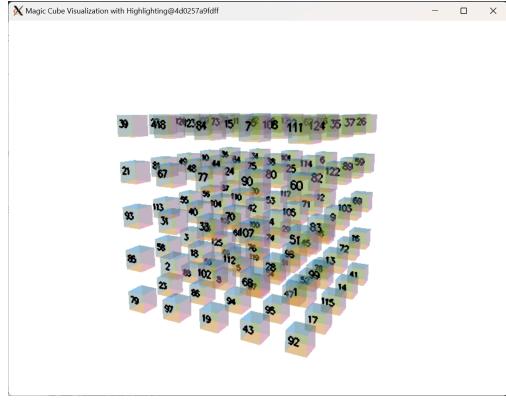
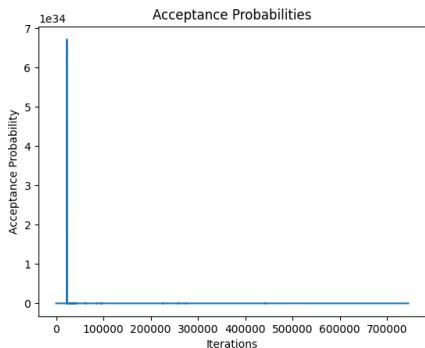
### a. Percobaan 1



b. Percobaan 2

State Awal	State Akhir
	
	 <div data-bbox="918 1364 1432 1459" style="background-color: black; color: white; padding: 5px;"> <p>Objective function: -49 with steps: 744772          Stuck: 724171          Solution not found.          Execution time: 3941 ms</p> </div>

c. Percobaan 3

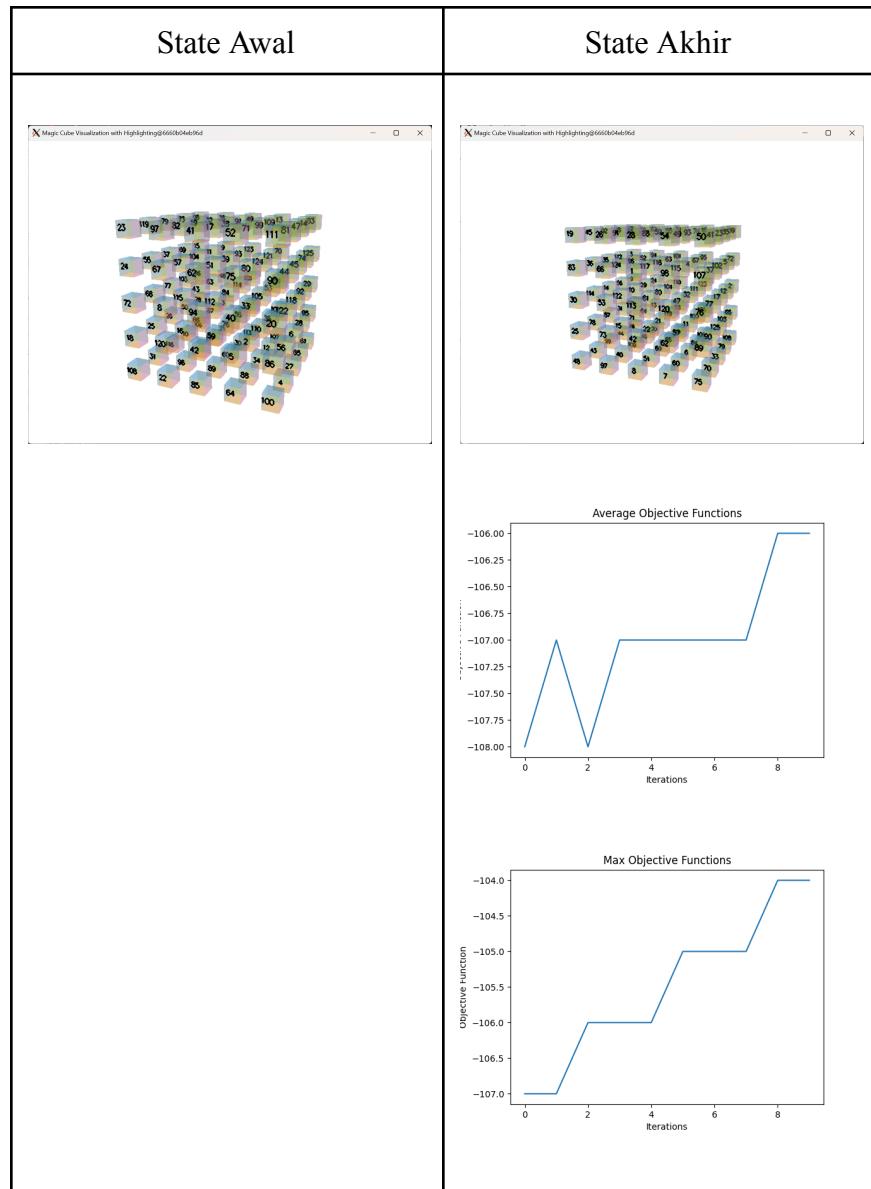
State Awal	State Akhir
	
	<pre>Objective function: -49 with steps: 744772 Stuck: 723076 Solution not found. Execution time: 3850 ms</pre>

### 4.3. Algoritma Genetic Algorithm

a. Populasi Sebagai Kontrol (16 populasi)

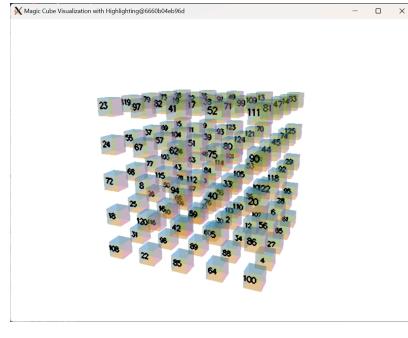
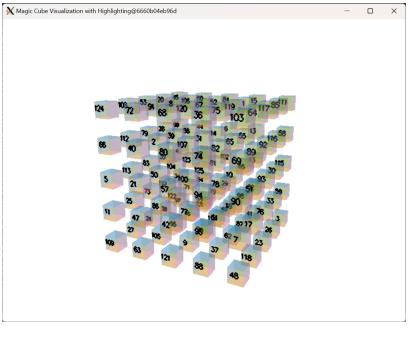
i. Variasi 1 (10 iterasi)

1. Percobaan 1



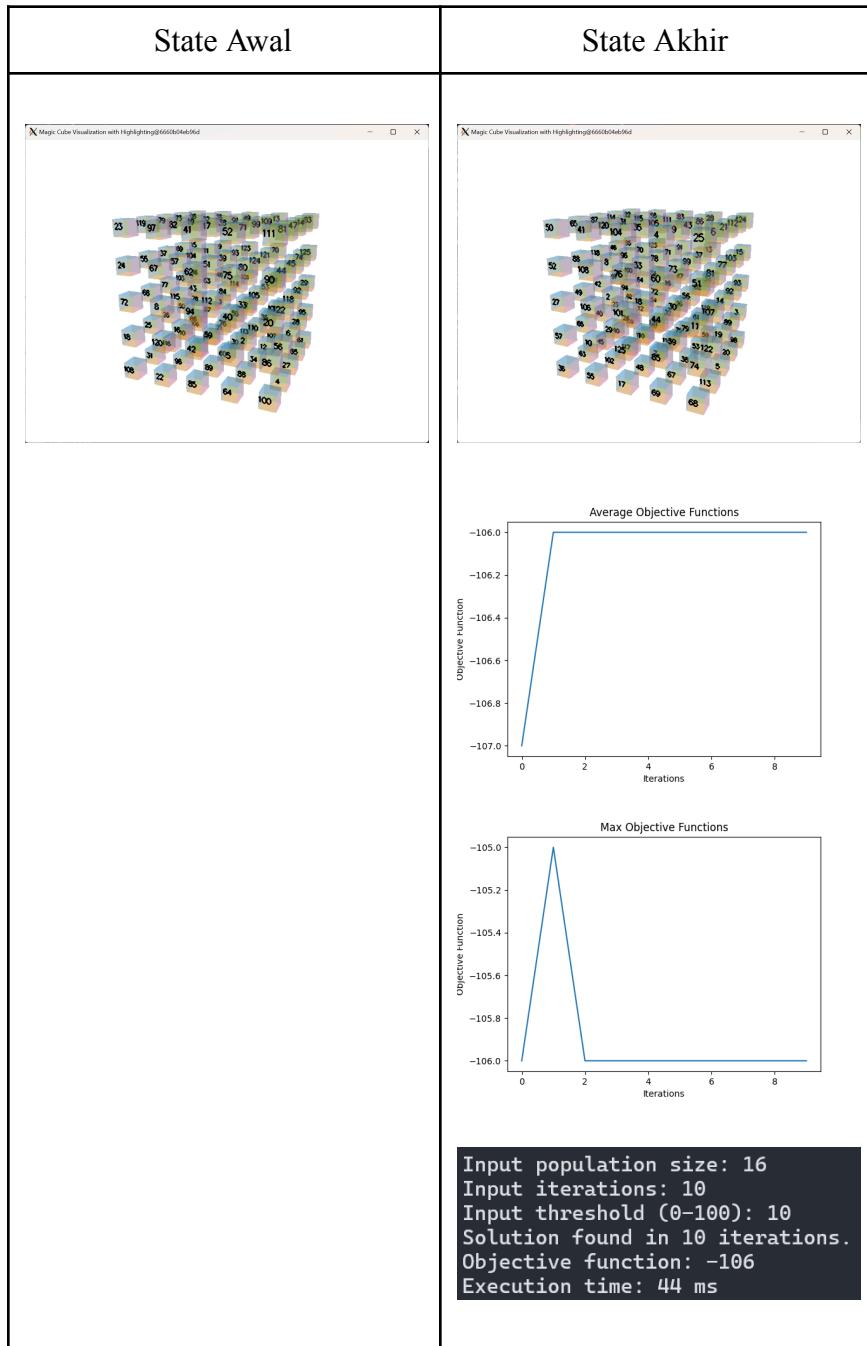
	<pre>Input population size: 16 Input iterations: 10 Input threshold (0-100): 10 Solution found in 10 iterations. Objective function: -104 Execution time: 45 ms</pre>
--	---

## 2. Percobaan 2

State Awal	State Akhir
	

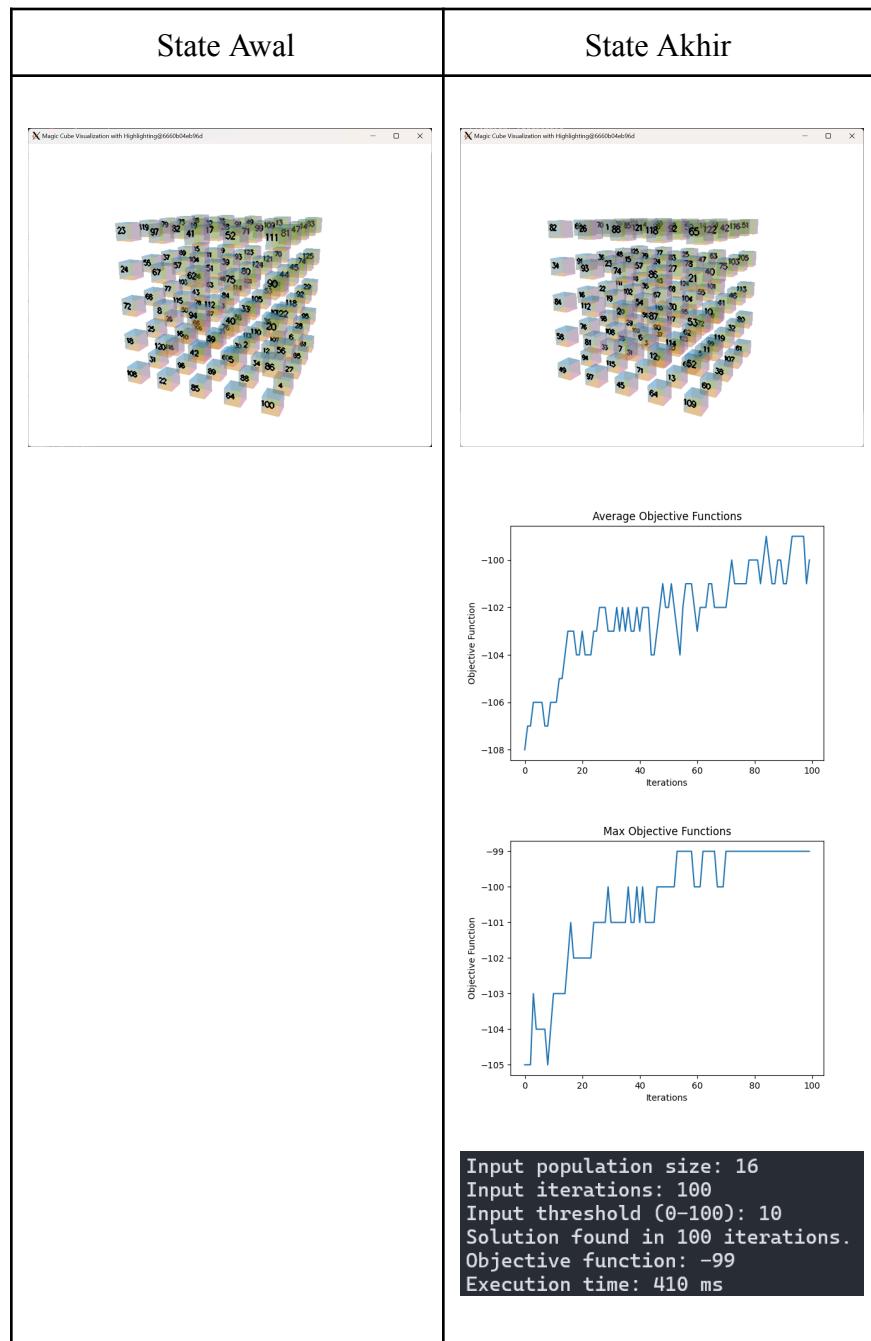


### 3. Percobaan 3

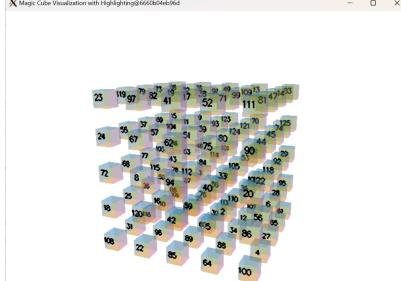
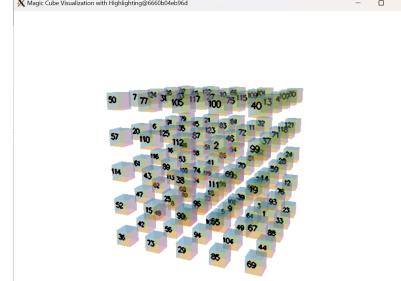
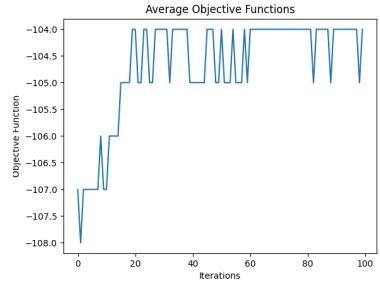
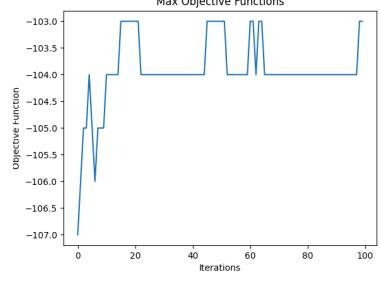


ii. Variasi 2 (100 iterasi)

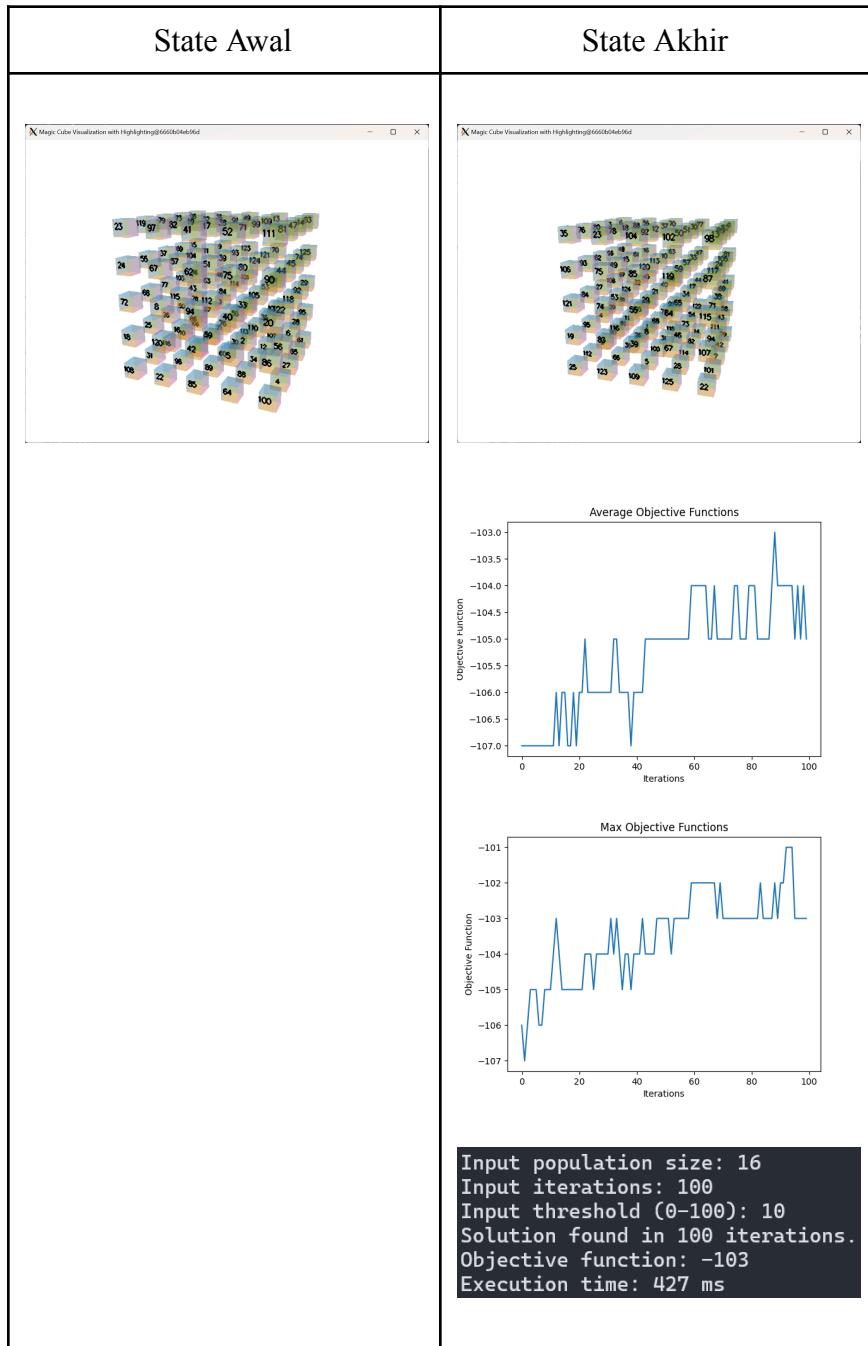
1. Percobaan 1



## 2. Percobaan 2

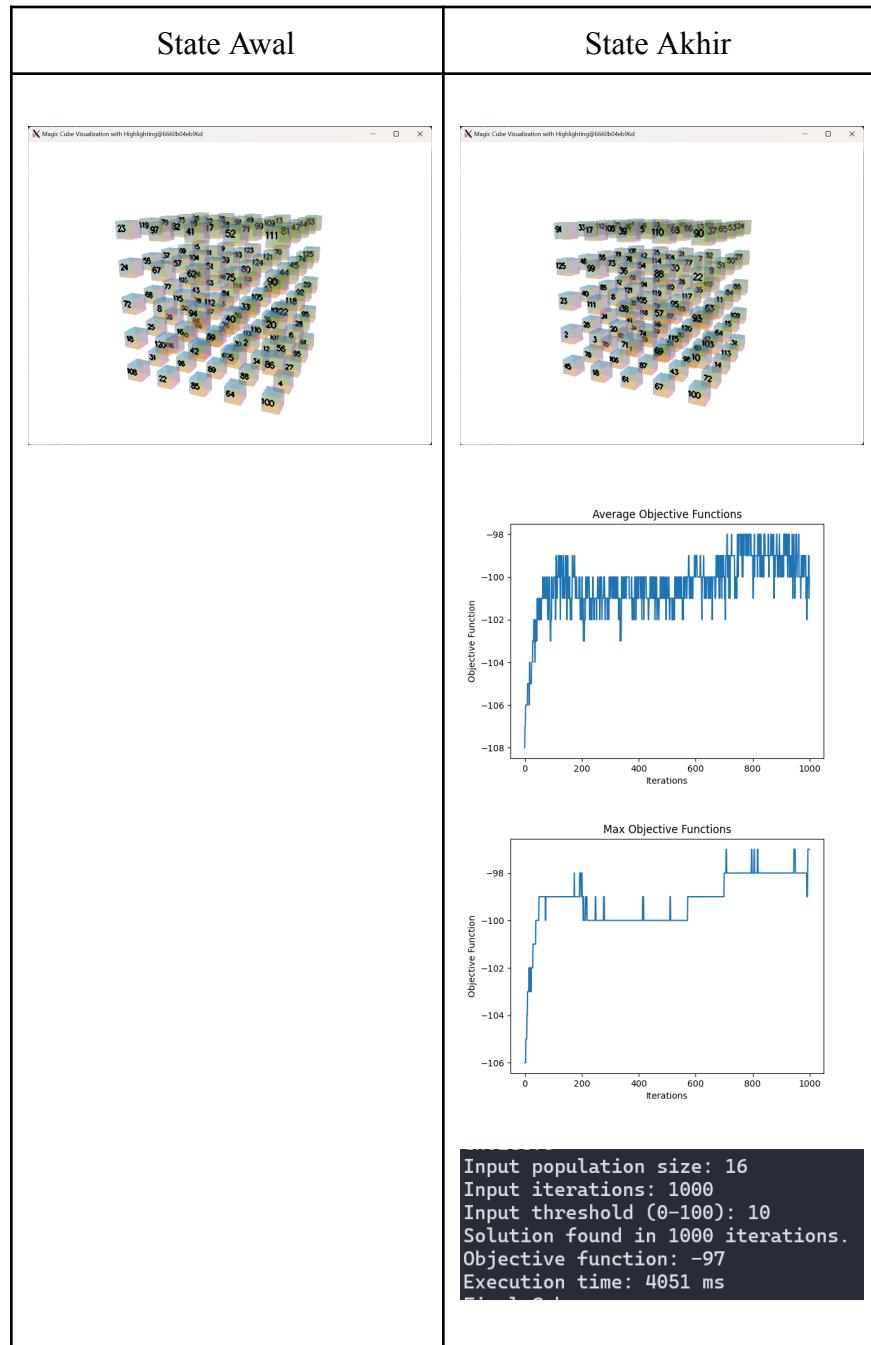
State Awal	State Akhir
	
	
	 <pre data-bbox="1013 1501 1432 1655">Input population size: 16 Input iterations: 100 Input threshold (0-100): 10 Solution found in 100 iterations. Objective function: -103 Execution time: 413 ms</pre>

### 3. Percobaan 3

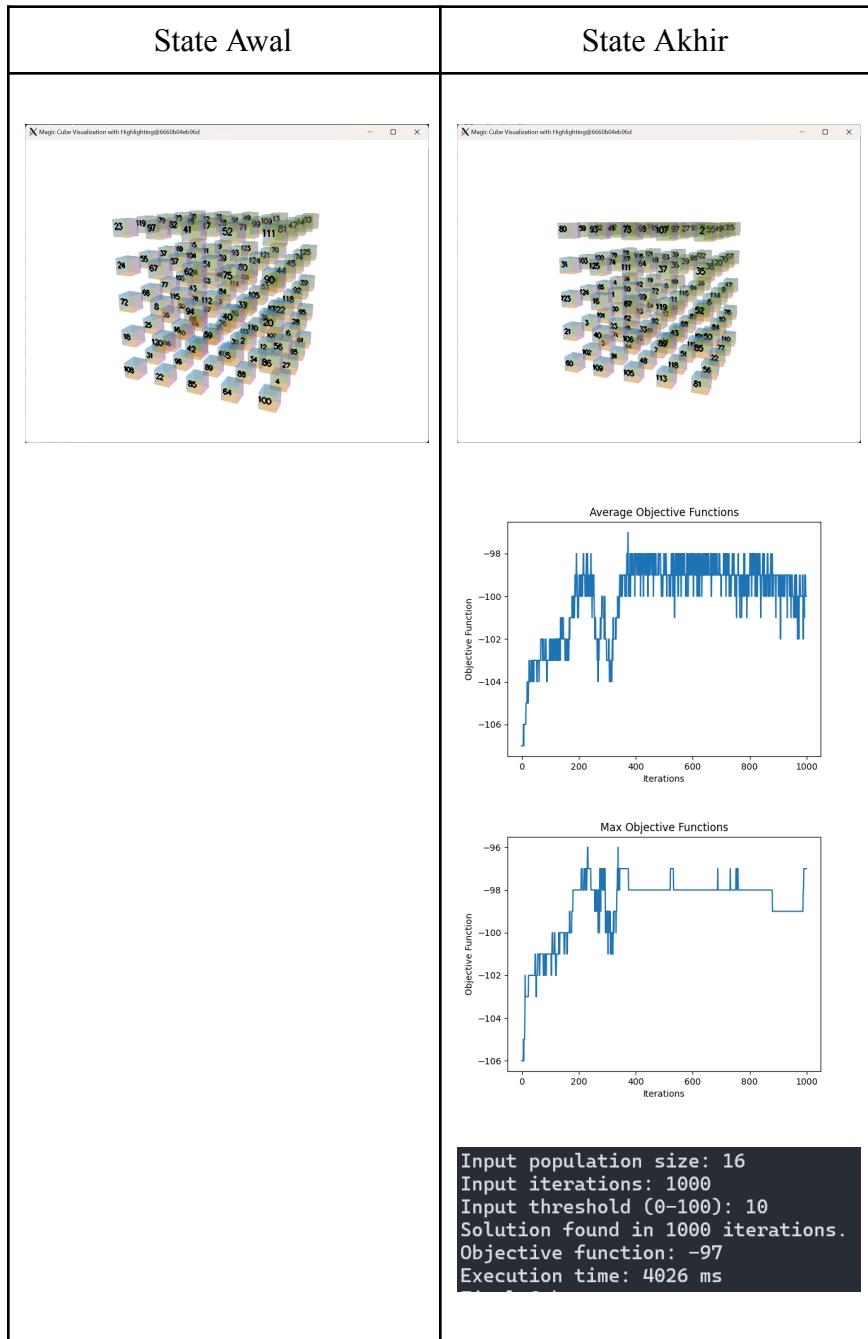


iii. Variasi 3 (1000 iterasi)

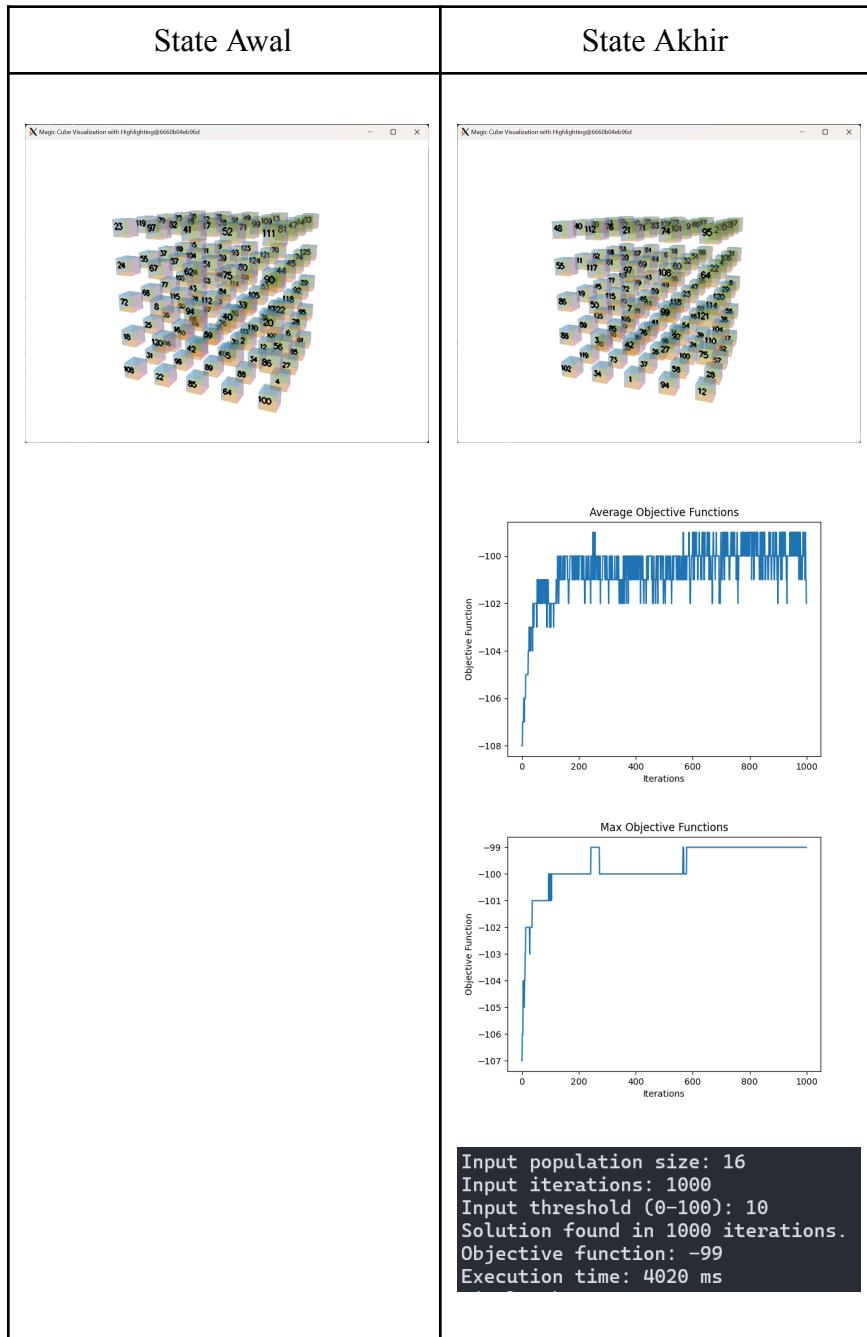
1. Percobaan 1



## 2. Percobaan 2



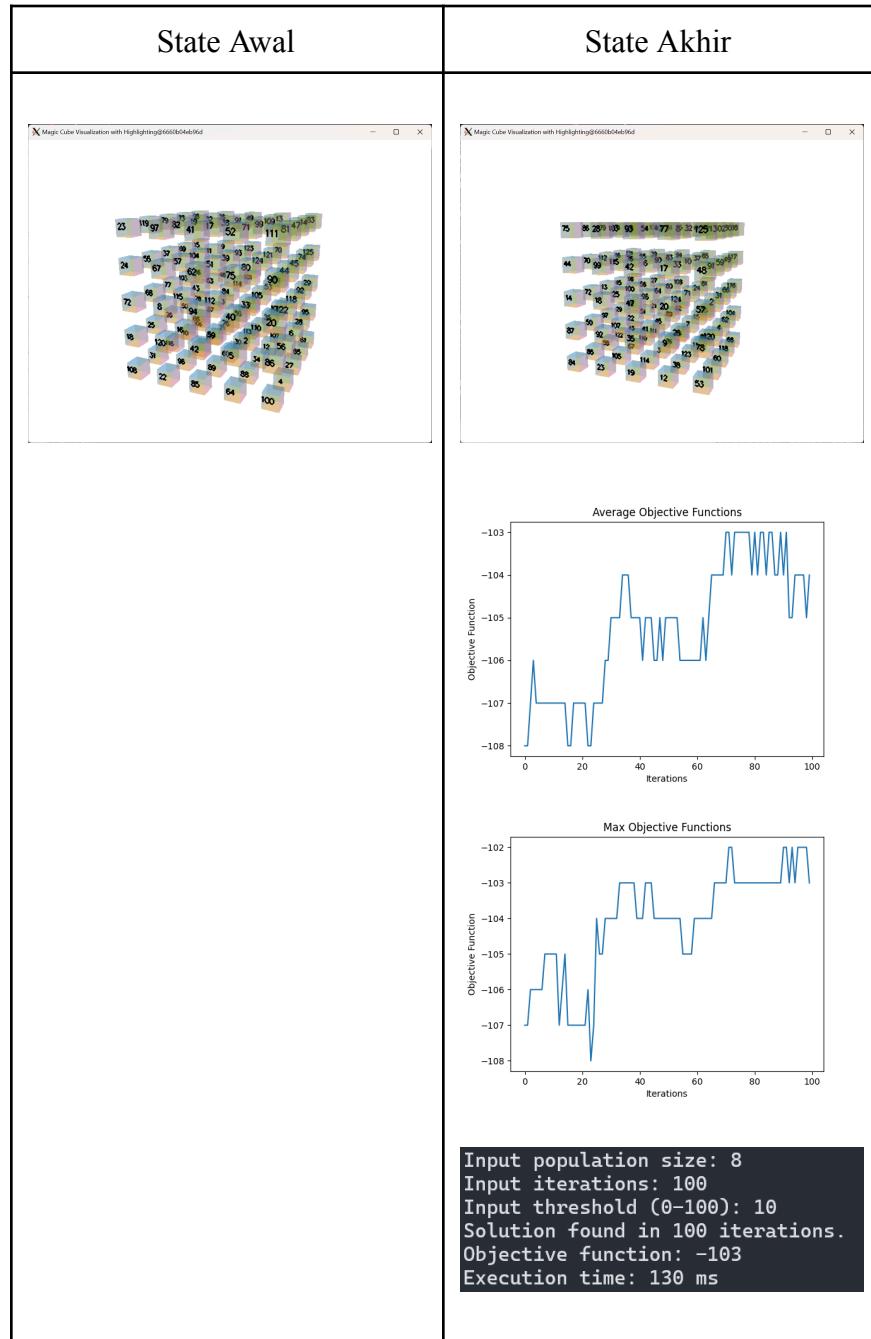
### 3. Percobaan 3



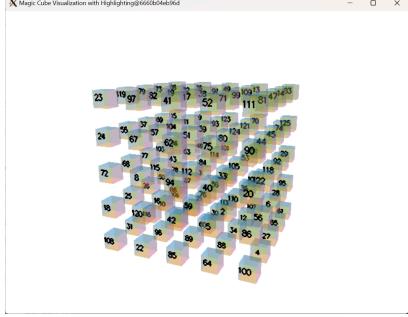
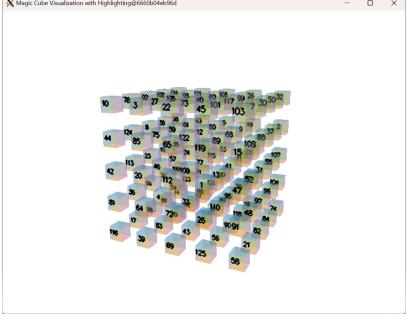
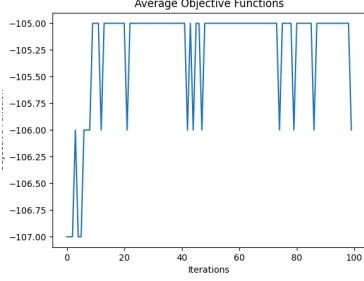
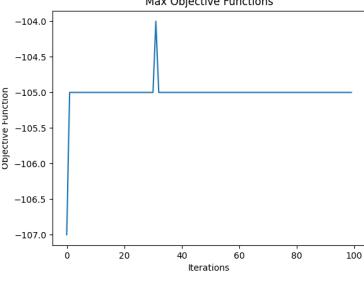
b. Iterasi Sebagai Kontrol (100 iterasi)

i. Variasi 1 (8 populasi)

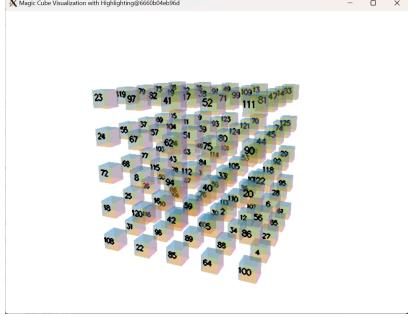
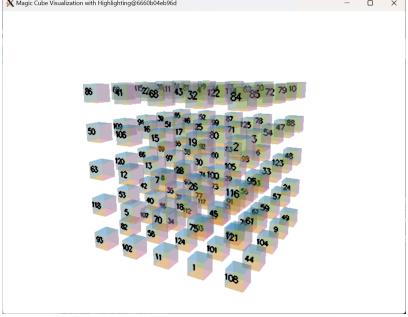
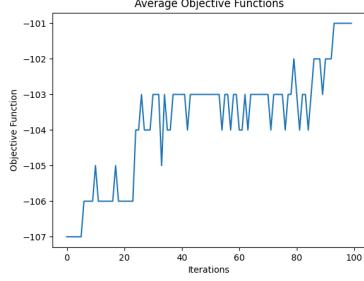
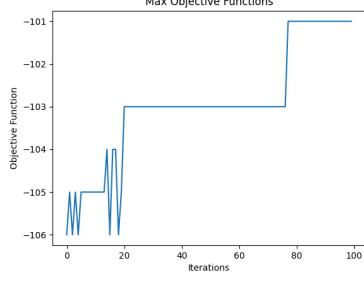
1. Percobaan 1



## 2. Percobaan 2

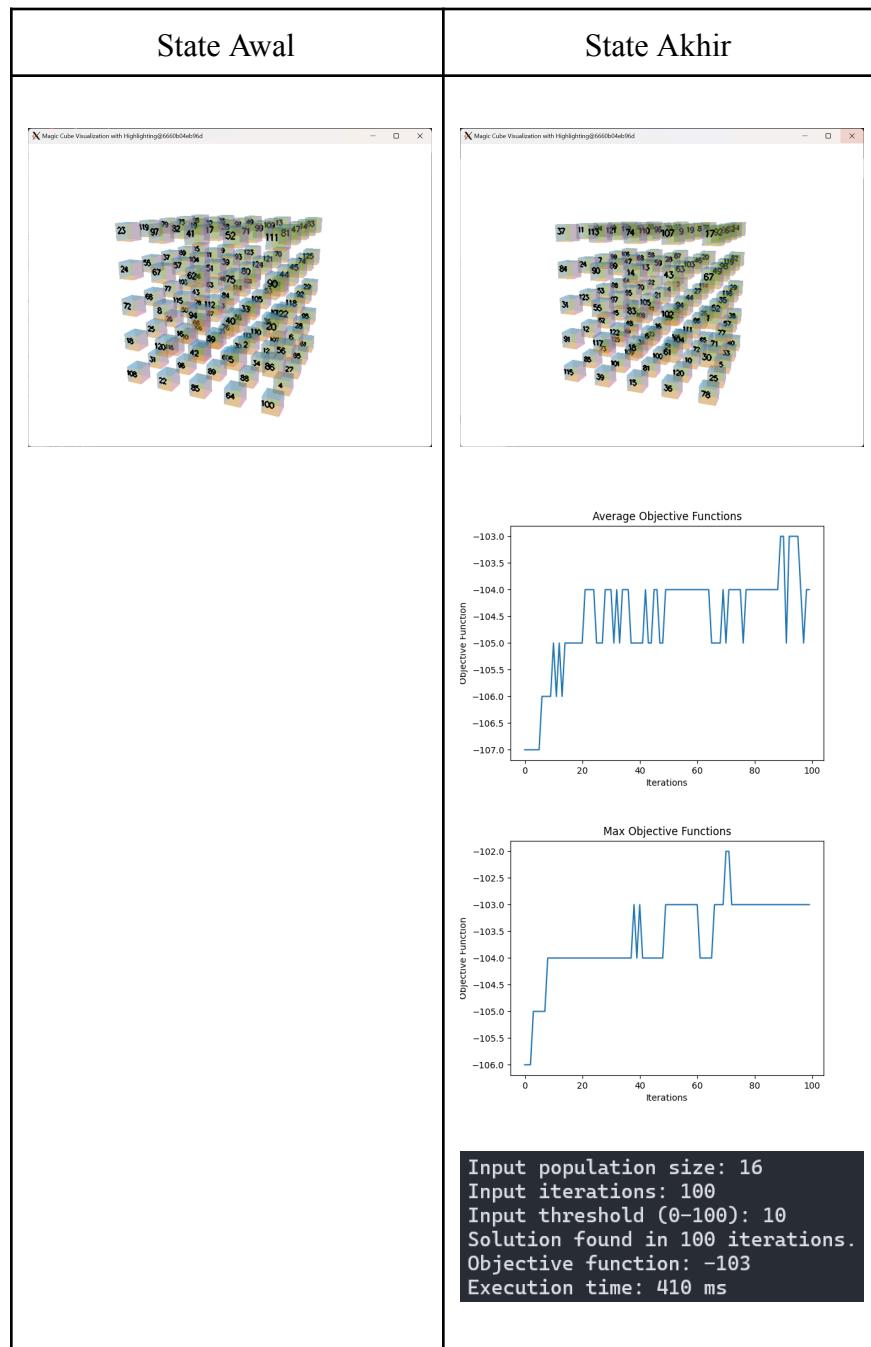
State Awal	State Akhir
	
	
	
	<pre>Input population size: 8 Input iterations: 100 Input threshold (0-100): 10 Solution found in 100 iterations. Objective function: -105 Execution time: 126 ms</pre>

### 3. Percobaan 3

State Awal	State Akhir
	
	
	
	<pre>Input population size: 8 Input iterations: 100 Input threshold (0-100): 10 Solution found in 100 iterations. Objective function: -101 Execution time: 121 ms</pre>

ii. Variasi 2 (16 populasi)

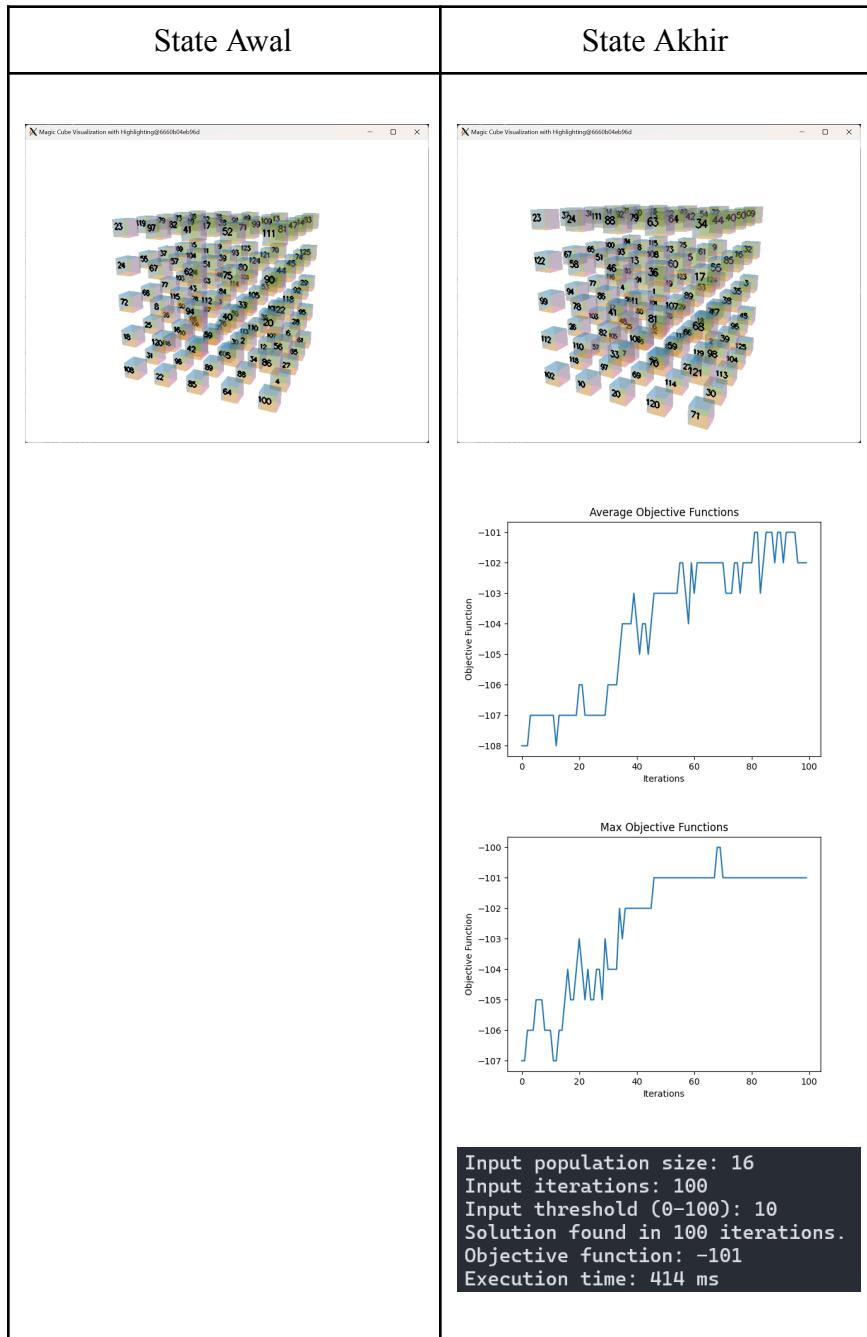
1. Percobaan 1



## 2. Percobaan 2

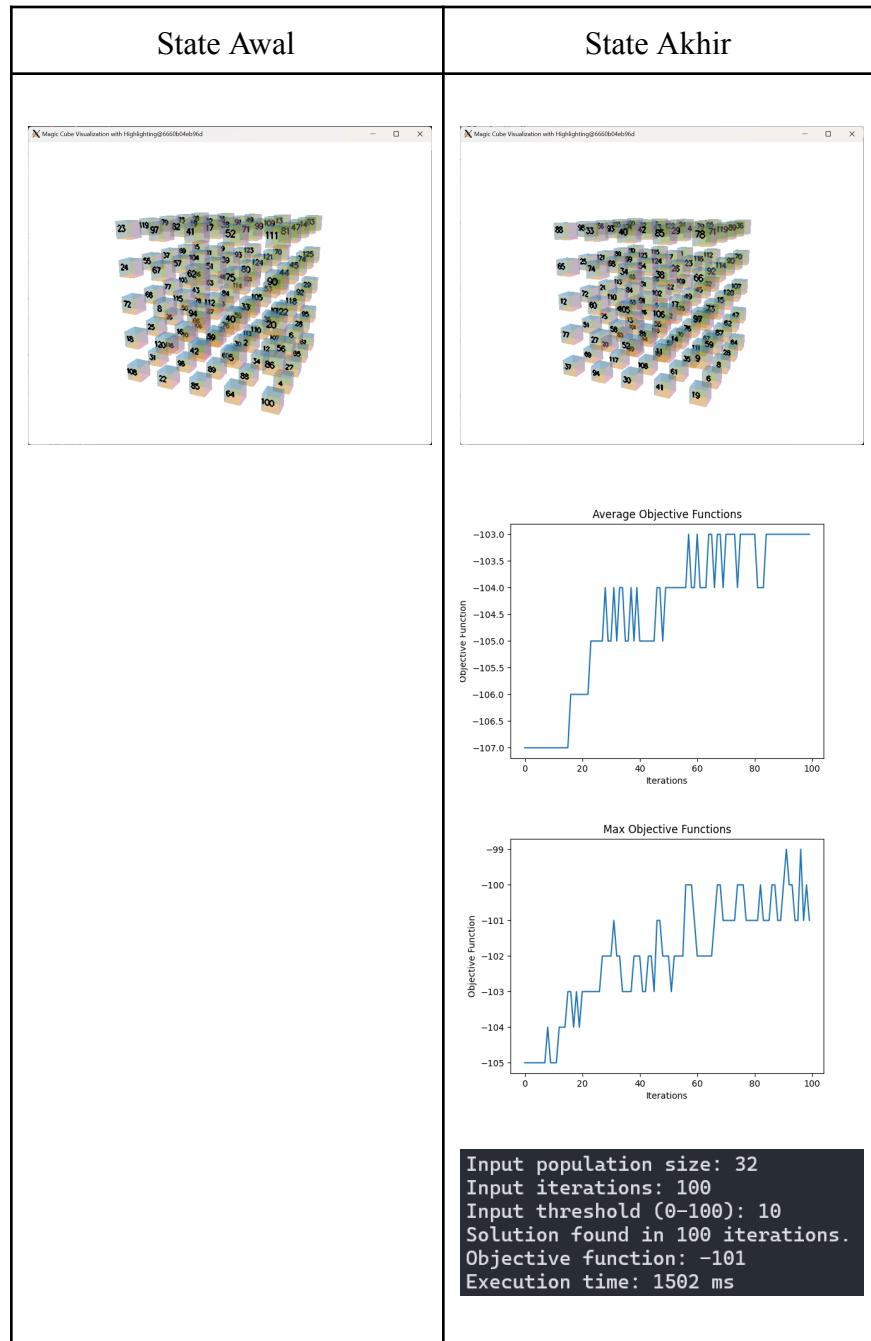
State Awal	State Akhir
	<pre>Input population size: 16 Input iterations: 100 Input threshold (0-100): 10 Solution found in 100 iterations. Objective function: -101 Execution time: 413 ms</pre>

### 3. Percobaan 3

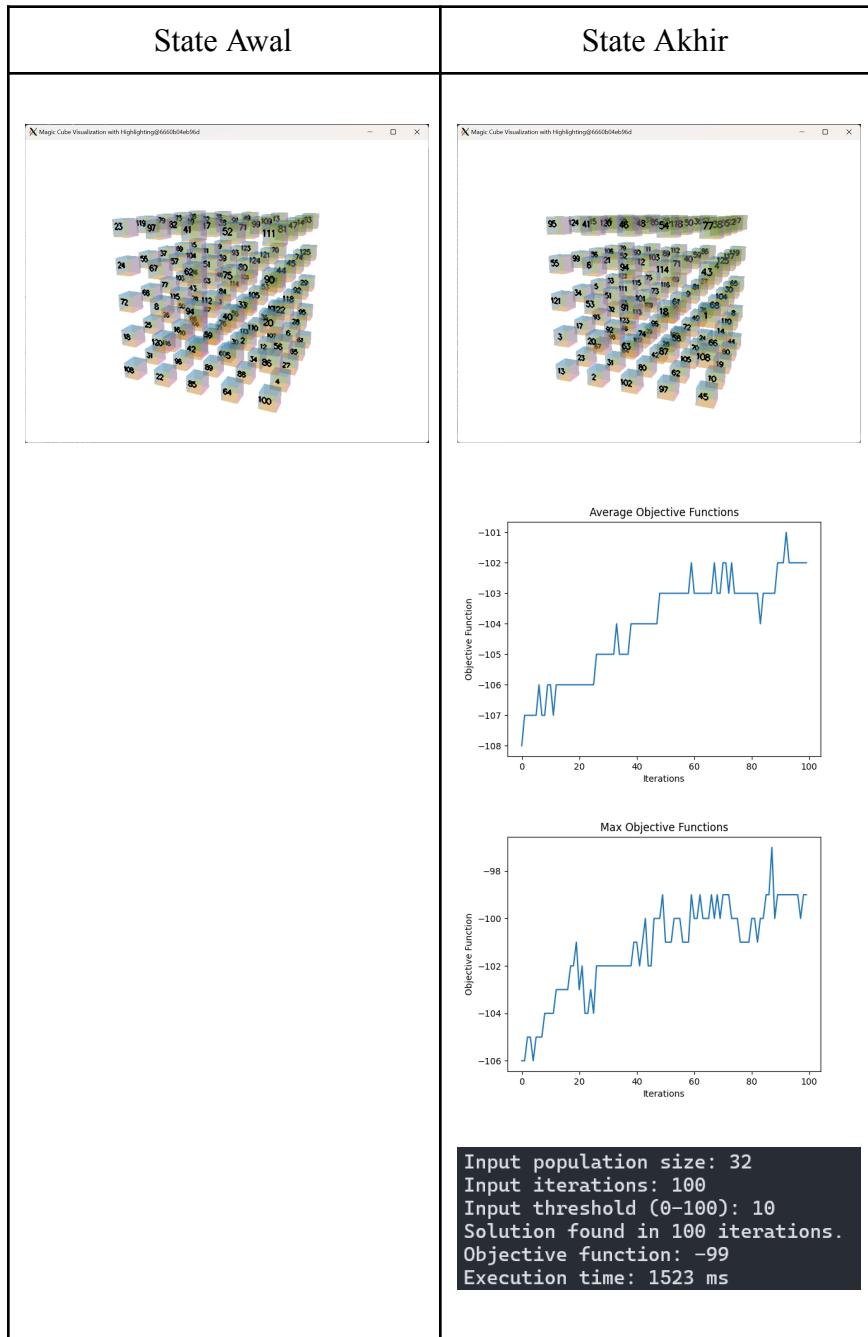


iii. Variasi 3 (32 populasi)

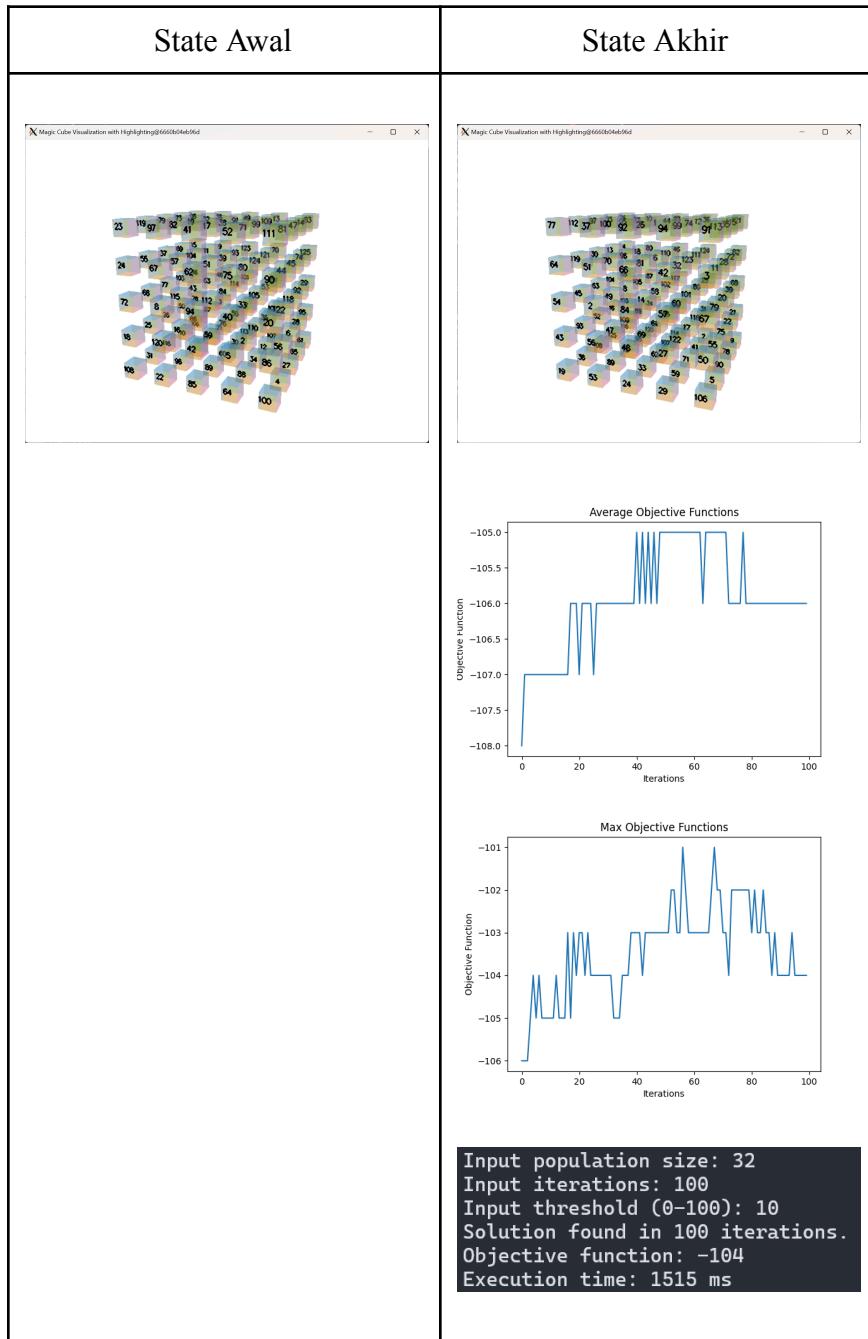
1. Percobaan 1



## 2. Percobaan 2



### 3. Percobaan 3



#### 4.4. Analisis Perbandingan Algoritma

Kubus ke-*i* pada percobaan ke-*i* pada algoritma-algoritma di atas adalah sama sehingga mudah untuk membandingkan hasil dari setiap algoritma. Tabel hasil penyelesaian *local search* menggunakan algoritma-algoritma di atas dan *objective value* yang dihasilkan di akhir pencarian terlampir sebagai berikut (semakin tinggi/mendekati 0, semakin baik),

Kategori	Percobaan 1			Percobaan 2			Percobaan 3		
	Iterasi	Value	Waktu	Iterasi	Value	Waktu	Iterasi	Value	Waktu
Steepest Ascent	31	-69	1437 ms	31	-67	1468 ms	33	-67	1537 ms
Sideways Move	100	-65	6134 ms	100	-67	6129 ms	100	-58	6454 ms
Random Restart	10 (restart)	-62	15804 ms	10 (restart)	-62	15233 ms	10 (restart)	-63	15099 ms
Stochastic	500.000	-66	4529 ms	500.000	-66	4806 ms	500.000	-68	4474 ms
Simulated Annealing	744.772	-49	3871 ms	744.772	-49	3941 ms	744.772	-49	3850 ms
Genetic (populasi)	V1 (10) V2 (100) V3 (1000)	V1: -104 V2: -99 V3: -97	V1: 45 ms V2: 410 ms V3: 4051 ms	V1 (10) V2 (100) V3 (1000)	V1: -104 V2: -103 V3: -97	V1: 48 ms V2: 413 ms V3: 4026 ms	V1 (10) V2 (100) V3 (1000)	V1: -106 V2: -103 V3: -99	V1: 44 ms V2: 427 ms V3: 4020 ms
Genetic	V1 (8)	V1:	V1:	V1 (8)	V1:	V1:	V1 (8)	V1:	V1:

c (iterasi )	V2 (16) V3 (32)	-103 V2: -103 V3: -101	130 ms V2: 410 ms V3: 1502 ms	V2 (16) V3 (32)	-105 V2: -101 V3: -99	126 ms V2: 413 ms V3: 1523 ms	V2 (16) V3 (32)	-101 V2: -101 V3: -104	121 ms V2: 413 ms V3: 1515 ms
--------------------	--------------------------	------------------------------------	--	--------------------------	-----------------------------------	--	--------------------------	------------------------------------	--

Sebagai catatan, khusus pada Random Restart, dihitung berdasarkan jumlah pengulangan (*restart*) dan untuk *Genetic Algorithm*, terdapat tiga (3) variasi percobaan, yaitu dengan tiga jumlah populasi yang berbeda dan tiga variasi percobaan lainnya berdasarkan jumlah iterasi yang berbeda namun populasi sama.

Berdasarkan percobaan yang kami lakukan, **Simulated Annealing** mendapatkan hasil yang paling mendekati global optima. Hal ini dikarenakan algoritma Simulated Annealing memiliki kemampuan untuk mengambil *state* yang lebih kecil nilai objektifnya jika peluangnya masih besar, sehingga mampu menghindari kemungkinan *stuck* di *local maxima*. Selain itu, karena penurunan nilai temperatur yang sangat lambat, menyebabkan algoritma Simulated Annealing memiliki jumlah iterasi jauh lebih banyak dibandingkan algoritma lainnya, sehingga probabilitas untuk mencapai global optima jauh lebih besar dibandingkan algoritma lainnya.

Algoritma lainnya yang lebih mendekati global optima selain *Simulated Annealing* adalah *Sideways Move*, diikuti oleh *Random Restart*, *Stochastic*, *Steepest Ascent*, kemudian terakhir *Genetic*. Algoritma *Sideways Move* dan *Steepest Ascent* memungkinkan program untuk mengeksplorasi seluruh *neighboring state* di *state* saat itu, dan mencari *state* dengan nilai terbaik di sana. Namun algoritma *Sideways Move* memungkinkan program untuk mencari *state* dengan nilai objektif yang sama dengan *state* saat ini, memungkinkan program untuk mencari kemungkinan *state* lain di fase *shoulder* dan masih mungkin untuk melanjutkan pencarian ke *state* yang lebih baik, dibandingkan *Steepest Ascent* yang akan langsung berhenti saat tidak menemukan *state* yang lebih baik. Namun keduanya tidak dapat menghindari kondisi *local maxima*, sehingga ketika masuk ke kondisi *local maxima*, program tidak bisa memilih *state*

dengan nilai objektif lebih rendah. Oleh karenanya, rata-rata nilai objektif *Sideways Move* lebih baik dibandingkan nilai di algoritma *Steepest Ascent*.

Algoritma *Random Restart* dan *Stochastic* mengambil setiap *neighboring state* secara acak. Pada algoritma *Random Restart*, memanfaatkan restart dengan *random initial state* ketika pencarian mencapai *local optimum*. Algoritma ini menggunakan dasar algoritma *steepest ascent* pada pencarian solusi setiap restart, jika stuck di local akan dilakukan *random initial state* dan dari setiap restart akan dipilih state dengan *objective function* yang lebih baik, sehingga algoritma ini akan mendekati ke global maksimum. Pada algoritma *Stochastic*, pemilihan *successor* dilakukan secara random dan setiap iterasi akan dipilih random *successor* dengan nilai yang baik, maka algoritma ini akan menghasilkan hasil yang lebih mendekati global maksimum.

Di sisi lain, *Genetic Algorithm* terlalu banyak memanfaatkan pembangkit bilangan acak sehingga anak yang dihasilkan dapat lebih baik atau lebih buruk dari parent yang dipilih. Hal ini mengakibatkan perpindahan *state* yang tidak stabil sehingga sulit untuk algoritma ini untuk menuju maksimum global.

Dari segi perbandingan waktu, sangat dipengaruhi oleh jumlah *state* yang dibangkitkan pada setiap iterasi dan juga banyak iterasi yang dilakukan. Algoritma yang membangkitkan seluruh *neighboring state* pada *Sideways Move*, *Steepest Ascent*, dan *Genetic Algorithm* memerlukan waktu dan *resource* yang jauh lebih banyak dibanding algoritma yang membangkitkan satu *state* pada setiap iterasinya seperti pada *Random Restart*, *Stochastic* dan *Simulated Annealing*. Namun, pada *Random Restart*, setiap restart akan dilakukan pencarian dengan *steepest ascent* dan akan berjalan sebanyak jumlah maksimal restart atau hingga solusi ditemukan, sehingga akan butuh waktu yang lebih lama juga.

Selain dari segi jumlah *state* yang dibangkitkan, proses untuk membangkitkan *state* juga berpengaruh besar. *Random restart* pada setiap *restart*-nya melakukan iterasi juga, sehingga jika ditotalkan jumlah iterasi pada setiap percobaan *random restart* dapat mencapai 3000 iterasi.

Selain itu, faktor kompleksitas algoritma seperti pada *Genetic Algorithm* juga berpengaruh besar ke waktu eksekusi. *Genetic algorithm* membangkitkan setiap populasi dan kemudian menghitung *fitness function* setiap populasi, melakukan proses *selection* dan *crossover*, dan kemudian *mutation*, sehingga waktu yang dibutuhkan sangat panjang.

Berdasarkan urutan rata-rata waktu eksekusi per iterasi, algoritma terlama adalah *Genetic Algorithm*, diikuti *Steepest Ascent* dan *Sideways Move*, kemudian *Random Restart, Stochastic*, dan kemudian *Simulated Annealing*.

Dari percobaan-percobaan yang dilakukan, rata-rata *objective values* pada setiap algoritma menunjukkan hasil yang relatif konsisten, tanpa perbedaan yang signifikan di antara masing-masing percobaan. Hal ini mengindikasikan bahwa setiap algoritma memiliki tingkat stabilitas yang serupa. Konsistensi ini juga menunjukkan bahwa algoritma-algoritma yang digunakan memiliki kemampuan yang mirip dalam mendekati solusi yang diinginkan walaupun algoritma-algoritma tersebut belum optimal.

Pada *Genetic Algorithm*, jumlah iterasi tidak lebih mempengaruhi hasil yang didapat dibanding dengan jumlah populasi. Berdasarkan hasil yang didapat, *objective value* maksimum yang didapat dari percobaan populasi sebagai kontrol adalah sebesar -97, yaitu dengan 1000 iterasi dan dengan waktu 4026 ms. Sedangkan *objective value* maksimum yang didapat dari percobaan iterasi sebagai kontrol adalah sebesar -99, yaitu dengan jumlah populasi sebanyak 32 dan dengan waktu 1523 ms. Dari kedua data tersebut, tidak cukup untuk mengambil kesimpulan parameter apa yang lebih baik untuk diutamakan pada *Genetic Algorithm*. Nilai dari *objective value* yang didapatkan juga berdasarkan hasil pembangkitan bilangan acak, sehingga sampel yang diambil harus lebih banyak untuk mengambil kesimpulan.

## **BAB V**

### **KESIMPULAN DAN SARAN**

#### **5.1.Kesimpulan**

Dalam tugas besar ini, kami berhasil mengimplementasikan algoritma pencarian lokal seperti hill-climbing, simulated annealing, dan genetic algorithm untuk mengoptimalkan susunan angka dalam kubus  $5 \times 5 \times 5$  agar mendekati konfigurasi magic cube dengan magic number sebesar 315. Meskipun algoritma yang kami gunakan belum sepenuhnya mencapai solusi ideal di mana setiap baris, kolom, tiang, dan semua diagonal memiliki jumlah yang sama dengan magic number, pendekatan ini menunjukkan bahwa optimasi lokal dapat mendekati solusi yang diharapkan.

Hasil eksperimen ini memperlihatkan potensi dari algoritma optimasi lokal untuk mendekati konfigurasi yang memenuhi syarat magic cube meskipun tidak mencapai solusi sempurna. Kesulitan mencapai magic number yang akurat di setiap sumbu menunjukkan adanya keterbatasan dalam pendekatan yang kami gunakan. Tugas besar ini memberikan wawasan penting tentang penerapan teknik optimasi dalam pemecahan masalah berbasis konfigurasi spasial yang dapat menjadi dasar untuk eksplorasi lebih lanjut dalam penggunaan algoritma yang lebih canggih atau variasi pendekatan lain untuk mencapai solusi yang lebih optimal.

#### **5.2.Saran**

Saran untuk perbaikan di masa yang akan datang adalah memilih objective function yang lebih efektif. Selain itu, penggunaan algoritma yang lebih canggih atau kombinasi beberapa metode dengan menggunakan heuristik tertentu dapat membantu mempercepat dan meningkatkan kemungkinan mencapai solusi optimal sehingga hasil pencarian lebih efektif dan efisien.

## **DAFTAR PUSTAKA**

ITB. (n.d.). *Beyond Classical Search - Hill Climbing*. Diambil dari

[https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804849395\\_IF3170\\_Materi3\\_Seg03\\_BeyondClassicalSearch\\_HillClimbing.pdf](https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804849395_IF3170_Materi3_Seg03_BeyondClassicalSearch_HillClimbing.pdf)

ITB. (n.d.). *Beyond Classical Search - Simulated Annealing*. Diambil dari

[https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804872404\\_IF3170\\_Materi03\\_Seg04\\_BeyondClassicalSearch\\_SimulatedAnnealing.pdf](https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804872404_IF3170_Materi03_Seg04_BeyondClassicalSearch_SimulatedAnnealing.pdf)

ITB. (n.d.). *Beyond Classical Search*. Diambil dari

[https://cdn-edunex.itb.ac.id/storages/files/1727405202098\\_IF3170\\_Materi03\\_Seg05\\_BeyondClassicalSearch.pdf](https://cdn-edunex.itb.ac.id/storages/files/1727405202098_IF3170_Materi03_Seg05_BeyondClassicalSearch.pdf)

Russell, S. J., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (Global ed.).

Pearson.

## LAMPIRAN

### Pembagian Tugas

No.	NIM	Pembagian Tugas
1.	13522005	Laporan, Stochastic, Genetic Algorithm, Plot, Cube (MagicFive)
2.	13522053	Laporan, Steepest Ascent, Sideways Move
3.	13522071	Laporan, Simulated Annealing, Cube (MagicFive)
4.	13522072	Laporan, Random Restart, Visualisasi