

IF3140 Dasar Inteligensi Artifisial
Implementasi Algoritma Local Search pada Magic Cube



Disusun oleh:

Jihan Aurelia / 18222001

Nasywaa Anggun Athiefa / 18222021

Ricky Wijaya / 18222043

Muhammad Adli Arindra / 18222089

Dosen Pengampu :

Ir. Windy Gambetta, M.B.A.

Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

Daftar Isi

Daftar Isi.....	2
Daftar Tabel.....	3
Daftar Gambar.....	4
Daftar Rumus.....	5
BAB I Deskripsi Persoalan.....	6
A. Magic Cube.....	6
B. Algoritma Local Search.....	7
BAB II Pembahasan.....	8
A. Pemilihan Objective Function.....	8
B. Penjelasan Implementasi Algoritma Local Search.....	9
1. Steepest Ascent Hill-Climbing with Sideways Move.....	9
2. Simulated Annealing.....	15
3. Genetic Algorithm.....	20
C. Hasil eksperimen dan analisis.....	33
1. Steepest Ascent Hill-Climbing with Sideways Move.....	34
2. Simulated Annealing.....	39
3. Genetic Algorithm.....	46
BAB III Analisis.....	51
A. Analisis Global Optima Setiap Algoritma.....	51
B. Perbandingan Setiap Algoritma.....	51
C. Durasi Pencarian Setiap Algoritma.....	52
D. Kekonsistenan Hasil Akhir.....	52
E. Iterasi dan Jumlah Populasi Akhir.....	53
BAB IV Spesifikasi Bonus.....	54
BAB IV Kesimpulan dan Saran.....	56
A. Kesimpulan.....	56
B. Saran.....	56
BAB VI Pembagian tugas.....	57
BAB V Referensi.....	58

Daftar Tabel

Tabel 2.2.1. Library untuk Sideways Move Hill-Climbing.....	9
Tabel 2.2.2. Static Variables untuk Sideways Move Hill-Climbing.....	10
Tabel 2.2.3. Fungsi SwapCost untuk Sideways Move Hill-Climbing.....	10
Tabel 2.2.4. Fungsi Run untuk Sideways Move Hill-Climbing.....	11
Tabel 2.2.5. Looping pada Sideways Move Hill-Climbing.....	13
Tabel 2.2.6. Mekanisme Swapping untuk Sideways Move Hill-Climbing.....	14
Tabel 2.2.7. Kondisional untuk Sideways Move Hill-Climbing.....	14
Tabel 2.2.8. Library untuk Simulated Annealing.....	15
Tabel 2.2.9. Static Variables untuk Simulated Annealing.....	16
Tabel 2.2.10. Fungsi GenerateRandomNeighbor untuk Simulated Annealing.....	17
Tabel 2.2.11. Fungsi Run untuk Simulated Annealing.....	18
Tabel 2.2.12. Looping untuk Simulated Annealing.....	18
Tabel 2.2.13. Kondisional untuk Simulated Annealing.....	19
Tabel 2.2.14. Library untuk Genetic Algorithm.....	20
Tabel 2.2.15. Static Variable untuk Genetic Algorithm.....	21
Tabel 2.2.16. Inisialisasi untuk Genetic Algorithm.....	22
Tabel 2.2.17. Fungsi ChooseCubeByRandom untuk Genetic Algorithm.....	25
Tabel 2.2.18. Fungsi Crossover untuk Genetic Algorithm.....	26
Tabel 2.2.19. Fungsi Mutation untuk Genetic Algorithm.....	27
Tabel 2.2.20. Fungsi Clone untuk Genetic Algorithm.....	28
Tabel 2.2.21. Fungsi Run untuk Genetic Algorithm.....	31
Tabel 3.1.1. Perbandingan algoritma Hill Climbing.....	52
Tabel 3.2.1. Code pembuktian algoritma Hill Climbing.....	52

Daftar Gambar

Gambar 2.3.1. Kondisi awal test case 1.....	33
Gambar 2.3.2. Kondisi awal test case 2.....	34
Gambar 2.3.3. Kondisi awal test case 3.....	34
Gambar 2.3.4. Hasil akhir case 1 Sideways Move Hill Climbing.....	35
Gambar 2.3.5. Plot nilai objective function terhadap banyak iterasi.....	36
Gambar 2.3.6. Hasil akhir case 2 Sideways Move Hill Climbing.....	36
Gambar 2.3.7. Plot nilai objective function terhadap banyak iterasi.....	37
Gambar 2.3.8. Hasil akhir case 3 Sideways Move Hill Climbing.....	38
Gambar 2.3.9. Plot nilai objective function terhadap banyak iterasi.....	38
Gambar 2.3.10. Hasil akhir case 1 Simulated Annealing.....	39
Gambar 2.3.11. Plot objective function terhadap iterasi Simulated Annealing.....	40
Gambar 2.3.12. Plot eE/t terhadap setiap iterasi.....	40
Gambar 2.3.13. Hasil akhir case 2 Simulated Annealing.....	41
Gambar 2.3.14. Plot objective function terhadap iterasi Simulated Annealing.....	42
Gambar 2.3.15. Plot eE/t terhadap setiap iterasi.....	42
Gambar 2.3.17. Plot objective function terhadap iterasi Simulated Annealing.....	44
Gambar 2.3.18. Plot eE/t terhadap setiap iterasi.....	44
Gambar 2.3.19. Hasil akhir case 1 Genetic Algorithm.....	46
Gambar 2.3.20. Plot hasil objective function terhadap iterations.....	47
Gambar 2.3.21. Hasil akhir case 2 Genetic Algorithm.....	48
Gambar 2.3.22. Plot hasil objective function terhadap iterations.....	48
Gambar 2.3.23. Hasil akhir case 3 Genetic Algorithm.....	49
Gambar 2.3.24. Plot hasil objective function terhadap iterations.....	50

Daftar Rumus

Rumus 1.1.1. Rumus untuk menghitung magic number.....	6
Rumus 2.1.1. Hasil perhitungan magic number.....	8
Rumus 2.1.2. Rumus objective function.....	8

BAB I

Deskripsi Persoalan

A. *Magic Cube*

Magic Cube dapat didefinisikan sebagai sekumpulan bilangan bulat yang tersusun menyerupai kubus berukuran $n \times n \times n$ dengan jumlah bilangan pada setiap baris, kolom, pilar, dan masing-masing empat diagonal ruang utama bernilai sama. Hasil penjumlahan ini disebut sebagai *magic value* yang dilambangkan dengan $M_3(n)$. Nilai tersebut bisa dicari menggunakan persamaan *magic number* berikut,

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

Rumus 1.1.1. Rumus untuk menghitung *magic number*

1. $M_3(n)$: *magic number*
2. N : ukuran/ ordo kubik

Angka-angka yang tersusun di dalam *magic cube* harus terdiri atas bilangan bulat di antara $1, \dots, n^3$. Nilai pada rentang tersebut tersebut hanya boleh muncul sekali dalam kubik tersebut. Berikut merupakan beberapa syarat utama yang harus dipenuhi oleh sebuah *Magic Cube*,

1. Setiap baris dalam kubus harus memiliki jumlah angka yang sama dengan *magic number*.
2. Setiap kolom di dalam kubus juga harus menghasilkan jumlah angka yang sama dengan *magic number*.
3. Setiap tiang yang terbentuk dari susunan vertikal angka dalam kubus harus memiliki jumlah yang sama dengan *magic number*.
4. Diagonal yang terbentang di seluruh kubus juga harus menghasilkan jumlah yang sama dengan *magic number*.
5. Diagonal yang terbentuk dalam setiap bidang pada kubus juga harus menghasilkan jumlah yang sama dengan *magic number*.

B. Algoritma *Local Search*

Pada tugas ini, mahasiswa diminta untuk menerapkan algoritma local search untuk memecahkan masalah *magic cube*. Terdapat 2 metode yang dapat digunakan oleh algoritma ini untuk penyusunan angka-angka di dalam kubus,

1. Penukaran posisi antara dua angka random. Cara ini memungkinkan eksplorasi angka-angka yang ada di dalam kubus.
2. Penukaran posisi antara lebih dari 2 angka pada satu iterasi. Metode ini hanya berlaku untuk algoritma local search dengan Genetic Algorithm melalui proses *crossover*. Penggunaan prinsip ini berguna untuk meningkatkan variasi dari potensi state yang lebih baik.

Berikut adalah beberapa istilah umum yang digunakan dalam penggunaan algoritma local search,

1. **Inisialisasi**, proses untuk menciptakan sebuah kubus 5x5x5 yang terdiri dari angka 1 sampai 125 dengan susunan yang acak.
2. **Neighbor**, kondisi selanjutnya yang memiliki memenuhi kriteria masing-masing algoritma.
3. **Current**, kondisi yang sedang ditinjau dan dibandingkan dengan state di sekitarnya.
4. **Value**, sebuah metode dari objek yang ditinjau untuk menghitung *objective value*
5. **Local maksimum**, kondisi ketika solusi yang ditemukan adalah yang terbaik untuk lingkungan yang terbatas sehingga ada solusi lain yang lebih baik namun tidak berhasil dideteksi.
6. **Global maksimum**, solusi terbaik secara keseluruhan dari *objective function*, yaitu ketika algoritma ini berhasil membentuk *magic cube*.

BAB II

Pembahasan

A. Pemilihan Objective Function

Untuk kasus **Diagonal Magic Cube** berukuran $5 \times 5 \times 5$, *objective function* yang dirancang harus mampu mengevaluasi seberapa dekat solusi saat ini dengan kondisi ideal, yaitu ketika semua baris, kolom, tiang, dan diagonal (baik bidang maupun ruang) memiliki jumlah yang sama dengan **magic number**. *Magic number* untuk kubus ini dapat dihitung menggunakan persamaan (1), dalam hal ini n adalah 5. Maka, *magic number* untuk kubus $5 \times 5 \times 5$ adalah:

$$\text{Magic Number} = \frac{5(5^3+1)}{2} = \frac{5(125+1)}{2} = \frac{5 \times 126}{2} = 315$$

Rumus 2.1.1. Hasil perhitungan *magic number*

Objective function harus mengukur seberapa jauh setiap aspek dari kubus dalam memenuhi *magic number*. Nilai dihitung melalui selisih *objective value* saat ini (penjumlahan *current state*) dengan *objective value* yang diharapkan, yaitu 315. Penghitungan *objective value* saat ini harus memenuhi syarat *magic cube*,

1. Jumlah angka pada setiap baris, kolom, dan tiang
2. Jumlah diagonal bidang, baik horizontal maupun vertikal
3. Jumlah diagonal ruang antar sudut pada kubus

Objective function akhir adalah jumlah dari semua selisih yang telah dijumlahkan. Selisih ditulis dengan simbol S . Jadi, total *objective function*-nya dapat dituliskan sebagai,

$$(S_{\text{baris}}) + (S_{\text{kolom}}) + (S_{\text{tiang}}) + (S_{\text{diagonal bidang}}) + (S_{\text{diagonal ruang}})$$

Rumus 2.1.2. Rumus *objective function*

Semakin kecil nilai *objective function*, semakin baik solusi tersebut, karena itu menunjukkan bahwa semua baris, kolom, tiang, dan diagonal semakin mendekati *magic number* (315).

Objective function untuk Diagonal Magic Cube 5x5x5 mengevaluasi solusi berdasarkan selisih antara jumlah angka pada setiap baris, kolom, tiang, diagonal bidang, dan diagonal ruang dengan magic number (315). Tujuan dari algoritma local search adalah **meminimalkan objective function** ini, sehingga **semua aspek kubus mendekati kondisi ideal**.

B. Penjelasan Implementasi Algoritma Local Search

1. Steepest Ascent Hill-Climbing with Sideways Move

Hill Climbing adalah algoritma local search yang melakukan perubahan-perubahan kecil untuk menyelesaikan permasalahan yang diberikan. Algoritma ini akan melakukan evaluasi dari kemungkinan terdekat (neighbor) yang menawarkan peningkatan tertinggi menuju kondisi tujuannya. Apabila tidak ada jalan yang lebih baik, maka akan terjadi proses terminasi sehingga berpotensi terjebak di maksimum lokal.

Library

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

Tabel 2.2.1. Library untuk Sideways Move Hill-Climbing

a. Library :

- i. System.Collections → Menyediakan koleksi non-generik di C#.
- ii. System.Collections.Generic → Menyediakan koleksi generik (seperti List<T>).
- iii. UnityEngine → Menyediakan fungsi dan kelas Unity, seperti MonoBehaviour sehingga script bisa digunakan sebagai komponen dalam Unity untuk visualisasi.

Static Variables

```
public class algo_sideway : MonoBehaviour {  
    private static int MN = main.MN;  
    private static int SIZE = main.SIZE;  
    private static int SIZE3 = main.SIZE3;
```

Tabel 2.2.2. Static Variables untuk Sideways Move Hill-Climbing

b. Variabel kelas:

- i. **MN, SIZE, DAN SIZE3 :** Mendefinisikan ukuran dari *magic cube*

SwapCost

```
public static int SwapCost(int[,] values, int i0,  
int j0, int k0, int i1, int j1, int k1) {  
    int[,] clone = (int[,])values.Clone();  
  
    // Swapping the element  
    int temp = clone[i0, j0, k0];  
    clone[i0, j0, k0] = values[i1, j1, k1];  
    clone[i1, j1, k1] = temp;  
  
    // Calculating the difference  
    return main.ObjectiveFunction(clone);  
}
```

Tabel 2.2.3. Fungsi SwapCost untuk Sideways Move Hill-Climbing

c. SwapCost

- i. **Tugas:** menghitung *objective value* setelah melakukan penukaran antara 2 elemen.

ii. **Parameter:**

1. values, keadaan *current state* dari *magic cube*.
2. i0, j0, k0, koordinat posisi pertama dari elemen yang akan ditukar.
3. i1, j1, k1, koordinat posisi kedua dari elemen yang akan ditukar.

iii. **Cara kerja:**

1. Membuat salinan dari objek yang ada ke variabel `clone`.
2. Menukar 2 elemen dari `clone` yang ditentukan oleh (i0, j0, k0) dan (i1, j1, k1).
3. Mengembalikan *objective value* dari `clone` sebagai evaluasi dari kualitas penukaran 2 angka tersebut.

Run

```
public static List<int[]> Run(int[, ,] values) {  
    List<int[]> sequences = new List<int[]>();  
    int previousScore = 0;  
    bool running = true;  
    int t = 0;  
    int stuck = 0;  
    int stuckLimit = 10;
```

Tabel 2.2.4. Fungsi Run untuk Sideways Move Hill-Climbing

d. Variabel kelas:

- i. **stuckLimit** : iterasi maksimum ketika algoritma ini *stuck* tanpa peningkatan nilai, setelah melewati batas ini maka algoritma akan berhenti.
- ii. **Tugas:** fungsi utama untuk mencari penukaran yang mampu memberikan penurunan terbesar dari *objective value*.

iii. **Parameter:**

- values, array 3 dimensi yang merepresentasikan *current state* dari *magic cube*.

e. Berikut adalah penjelasan detail di dalam kelas Run,

```
// Objective Function
    while (running) {
        t++;
        int i0 = 0, j0 = 0, k0 = 0;
        int i1 = 0, j1 = 0, k1 = 0;
        int minV = int.MaxValue;
        for (int di = 0; di < SIZE; ++di) {
            for (int dj = 0; dj < SIZE; ++dj) {
                for (int dk = 0; dk < SIZE; ++dk)
                {
                    for (int ni = 0; ni < SIZE;
+ni) {
                        for (int nj = 0; nj <
SIZE; ++nj) {
                            for (int nk = 0; nk <
SIZE; ++nk) {
                                if (ni != di || nj
!= dj || nk != dk) {
                                    int v =
SwapCost(values, di, dj, dk, ni, nj, nk);
                                    if (v <= minV)
{
                                        minV = v;
                                        i0 = di;
                                        j0 = dj;
                                        k0 = dk;
                                        i1 = ni;
```

Tabel 2.2.5. Looping pada Sideways Move Hill-Climbing

i. Cara kerja:

- a. Melakukan looping selama kondisi running bernilai True,
 - 1. Melakukan iterasi terhadap seluruh pasangan elemen di dalam kubus untuk menemukan pertukaran yang menghasilkan *objective value* terkcel ($\min V$)
 - 2. Koordinat dari pertukaran yang menghasilkan $\min V$ akan disimpan dalam variabel i_0, j_0, k_0, i_1, j_1 , dan k_1 .

```
// Swapping the elements
        (values[i0, j0, k0], values[i1, j1, k1]) =
(values[i1, j1, k1], values[i0, j0, k0]);
        sequences.Add(new int[] {i0, j0, k0, i1,
j1, k1, minV});
```

Tabel 2.2.6. Mekanisme *Swapping* untuk Sideways Move Hill-Climbing

3. Kedua posisi tersebut akan ditukar dan urutannya serta nilai *minV* akan disimpan di variabel *sequences*.

```
        if      (previousScore ==  
main.ObjectiveFunction(values)) {  
    stuck++;  
}  
else {  
    stuck = 0;  
}  
previousScore =  
main.ObjectiveFunction(values);  
if (stuck > stuckLimit) {  
    running = false;  
}  
}  
return sequences;  
}
```

Tabel 2.2.7. Kondisional untuk Sideways Move Hill-Climbing

4. Jika *objective value* tidak menunjukkan perbaikan, maka variabel *stuck* akan bertambah 1.
5. Jika *objective value* menunjukkan perbaikan, maka variabel *stuck* akan diubah menjadi 0.
6. Jika *stuck* melebihi *stuckLimit* maka algoritma akan berhenti.
7. Nilai yang dikembalikan adalah *sequence* yang menyimpan urutan penukaran posisi elemen di dalam kubus dan *objective value* dari setiap langkah.

2. Simulated Annealing

Simulated annealing adalah algoritma yang menggabungkan pendekatan *hill climbing* dengan *random walk* untuk menemukan solusi dari masalah kompleks, terutama ketika terdapat banyak lokal maksimum. Algoritma ini memungkinkan langkah buruk (*bad move*) untuk keluar dari lokal maksimum, tetapi mengurangi frekuensi langkah buruk tersebut seiring berjalaninya waktu.

Berikut adalah fungsi/ kelas yang kami gunakan untuk menyelesaikan permasalahan *magic cube* dengan menggunakan bahasa pemrograman C#,

<i>Library</i>
using System; using System.Collections; using System.Collections.Generic; using System.Diagnostics; using System.Linq; using UnityEngine; using Random = System.Random;

Tabel 2.2.8. Library untuk Simulated Annealing

a. Library :

- i. System → Mendukung fungsi dasar .NET, seperti Math untuk perhitungan eksponensial (Math.Exp).
- ii. System.Collections → Menyediakan koleksi non-generik.
- iii. System.Collections.Generic → Menyediakan koleksi generik seperti List<T>
- iv. System.Diagnostics → Berguna untuk debugging
- v. System.Linq → Menyediakan metode pemrosesan koleksi, seperti Where dan Select.
- vi. UnityEngine → Mendukung kelas pada Unity seperti MonoBehaviour untuk visualisasi.

- vii. Random = System.Random → Membuat instance Random yang digunakan untuk menghasilkan bilangan acak dalam kelas GenerateRandomNeighbor.

Static Variables

```
public class algo_annealing : MonoBehaviour {
    private static int MN = main.MN;
    private static int SIZE = main.SIZE;
    private static int SIZE3 = main.SIZE3;

    private static int[] _currentSeq;
```

Tabel 2.2.9. Static Variables untuk Simulated Annealing

b. Variabel kelas:

- i. **MN, SIZE, DAN SIZE3** : Mendefinisikan ukuran dari *magic cube*
- ii. **_currentSeq** : Menyimpan urutan posisi yang ditukar dan *objective value* dari *state* tersebut

GenerateRandomNeighbor

```
private static int[, , ]
GenerateRandomNeighbor(int[, , ] state)
{
    int[, , ] neighbor = (int[, , ]) state.Clone();
    Random random = new Random();
    int i0 = random.Next(SIZE); int j0 = random.Next(SIZE); int k0 = random.Next(SIZE);
    int i1 = random.Next(SIZE); int j1 = random.Next(SIZE); int k1 = random.Next(SIZE);
    (neighbor[i0, j0, k0], neighbor[i1, j1, k1]) =
    (neighbor[i1, j1, k1], neighbor[i0, j0, k0]);
```

```

        _currentSeq = new[] { i0, j0, k0, i1, j1, k1,
main.ObjectiveFunction(neighbor) };

        return neighbor;
}

```

Tabel 2.2.10. Fungsi GenerateRandomNeighbor untuk Simulated Annealing

c. **GenerateRandomNeighbor:**

- i. **Tugas:** menghasilkan tetangga secara acak dari *current state* dengan cara melakukan *swap* pada 2 angka acak dalam kubus.
- ii. **Parameter:**
 1. state, kondisi kubus saat ini
- iii. **Cara kerja:**
 1. Menyalin *current state* ke variabel *neighbor*
 2. Memiliki 2 posisi acak dari *neighbor* untuk ditukar
 3. Melakukan mekanisme *swap* pada pada kedua elemen yang terpilih
 4. Memperbarui variabel *_currentSeq* dengan posisi yang di-*swap* beserta *objective value* untuk *neighbor* tersebut.

Run

```

public static List<int[]> Run(int[,] values, double
initialTemp = 1000, double coolingRate = 0.999,
double minTemp = 0.01) {
    List<int[]> sequences = new List<int[]>();
    int[,] currentState =
(int[,])values.Clone();
    double currentValue =
main.ObjectiveFunction(currentState);
    double temperature = initialTemp;
}

```

```
var random = new Random();
```

Tabel 2.2.11. Fungsi Run untuk Simulated Annealing

d. **Run:**

i. **Tugas:** Menjalankan algoritma Simulated Annealing

ii. **Parameter:**

1. values, keadaan awal
2. initialTemp, temperatur awal
3. coolingRate, laju penurunan suhu
4. minTemp, temperatur minimum

iii. **Cara kerja:**

1. Melakukan inisialisasi dari keadaan awal menyalin nilai values ke currentState dan menghitung *objective value*.
2. Melakukan looping selama temperatur masih lebih besar dari minTemp.

```
while (temperature > minTemp) {  
    int[,] neighbor =  
        GenerateRandomNeighbor(currentState);  
    double neighborValue =  
        main.ObjectiveFunction(neighbor);  
  
    double deltaE = currentValue -  
        neighborValue;
```

Tabel 2.2.12. Looping untuk Simulated Annealing

3. Menghasilkan *neighbor* secara acak dengan GenerateRandomNeighbor.

4. Kemudian, dihitung selisih antara *objective value* dari *neighbor* dan *current* (*deltaE*).

```
if (deltaE > 0 || Math.Exp(Math.Min(deltaE / temperature, 700)) > random.NextDouble()) {  
    currentValue = neighborValue;  
    sequences.Add(_currentSeq);  
}  
  
temperature *= coolingRate;  
}  
  
return sequences;  
}  
}
```

Tabel 2.2.13. Kondisional untuk Simulated Annealing

5. Mengubah *currentState* menjadi *neighbor*, memperbarui *currentValue*, dan menyimpan langkah ini di *sequences* apabila,
 - a. *deltaE* yang dihasilkan lebih optimal (*deltaE* > 0)
 - b. nilai probabilitas yang dihasilkan lebih besar dari angka *random* antara 0 sampai 1 ($\text{Math.Exp}(\text{deltaE} / \text{temperature}) > \text{random.NextDouble}()$)
 - c. Mengalikan temperatur dengan *coolingRate* agar temperaturnya turun secara bertahap.
6. Nilai yang dikembalikan adalah *sequence* yang menyimpan urutan perubahan posisi elemen dalam kubus.

3. Genetic Algorithm

Algoritma ini bekerja berdasarkan prinsip seleksi alam yang dikenal sebagai evolusi. Algoritma ini dapat disamakan dengan organisme yang melestarikan spesiesnya melalui reproduksi, perubahan genetik (mutasi), dan evolusi untuk beradaptasi dengan lingkungan sekitarnya guna bertahan hidup (*survival of the fittest*).

Berikut adalah fungsi/ kelas yang kami gunakan untuk menyelesaikan permasalahan *magic cube* dengan menggunakan bahasa pemrograman C#,

Library

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using Unity.VisualScripting;
using Random = System.Random;
using UnityEngine;
```

Tabel 2.2.14. Library untuk Genetic Algorithm

a. **Library :**

- i. **System** : mendukung fungsi dasar .NET, seperti Math untuk perhitungan eksponensial (Math.Exp)
- ii. **System.Collections** : menyediakan kelas dan antarmuka untuk struktur data yang berguna, seperti ArrayList, Hashtable, Queue, Stack, dan lainnya
- iii. **System.Collections.Generic** : berisi implementasi struktur data yang sama dengan System.Collections namun dengan tipe data generic (contoh : List<T>, Dictionary< TKey, TValue>, dan Queue<T>)
- iv. **System.Linq** : menyediakan fitur Language-Integrated Query (LINQ) sehingga pengguna dapat menggunakan query yang mirip

dengan SQL pada koleksi data dalam C#.

- v. **Unity.VisualScripting** : library yang disediakan oleh Unity untuk fitur Visual Scripting sehingga pembuatan logika tidak perlu menulis manual **Random** : **System.Random** untuk menghasilkan angka random
- vi. **UnityEngine** : library utama Unity yang menyediakan fungsi untuk manipulasi elemen-elemen seperti transformasi objek, audio, rendering, dan fisika

Static Variables

```
private static int MN = main.MN;
private static int SIZE = main.SIZE;
private static int SIZE3 = main.SIZE3;
```

Tabel 2.2.15. *Static Variable* untuk Genetic Algorithm

b. Static variable :

- i. **MN, SIZE, DAN SIZE3** : Mendefinisikan ukuran dari *magic cube*
- ii. **_currentSeq** : Menyimpan urutan posisi yang ditukar dan *objective value* dari *state* tersebut

AlgoGenetic

```
public class algo_genetic : MonoBehaviour {
    private class child {
        public int[,] State { get; set; }
        public List<int[]> Sequence { get; set; }
        public int Heuristic;
        public child(int[,] values) {
            this.State = values;
            this.Sequence = new List<int[]>();
            this.Heuristic =
```

```

        main.ObjectiveFunction(this.State);
    }

    public child(child copy) {
        this.State = copy.State;
        this.Sequence = CopySeq(copy.Sequence);
        this.Heuristic = copy.Heuristic;
    }

    public child(child before, int[] seq) {
        this.State = before.State;
        (this.State[seq[0]], seq[1], seq[2]),
        this.State[seq[3], seq[4], seq[5]]) =
            (this.State[seq[3], seq[4], seq[5]],
        this.State[seq[0], seq[1], seq[2]]);
        this.Heuristic =
        main.ObjectiveFunction(this.State);

        this.Sequence = CopySeq(before.Sequence);
        this.Sequence.Add(new int[] {seq[0],
        seq[1], seq[2], seq[3], seq[4], seq[5],
        this.Heuristic});
    }

    private List<int[]> CopySeq(List<int[]> seq) {
        List<int[]> ret = new List<int[]>();
        for (int i = 0; i < seq.Count; ++i) {
            ret.Add((int[])seq[i].Clone());
        }
        return ret;
    }
}

```

Tabel 2.2.16. Inisialisasi untuk Genetic Algorithm

a. **algo_genetic :**

i. **Tugas** : mendefinisikan kelas algo_genetic dengan sebuah nested class bernama child. Kelas child bertanggung jawab untuk mewakili satu entitas atau solusi dalam algoritma Genetic Algorithm dengan atribut yang menyimpan kondisi saat ini (state), urutan tindakan yang dilakukan (sequence), dan nilai heuristik (heuristic) yang mengukur kualitas solusi

ii. **Parameter :**

1. int[,,] State : array tiga dimensi yang menyimpan kondisi atau status dari entitas saat ini
2. List<int[]> Sequence : daftar tindakan atau urutan operasi yang diterapkan pada state untuk mencapai kondisi saat ini
3. int Heuristic : nilai yang dihasilkan oleh fungsi objectiveFunction dari kelas main, digunakan untuk menilai efektivitas kondisi entitas (state)

iii. **Cara kerja :**

1. Constructor child(int[, ,] values) : konstruktor untuk menginisialisasi child baru dengan kondisi awal State yang diberikan sebagai parameter values. Selain itu juga berguna untuk mengatur sequence sebagai list kosong dan menghitung nilai heuristic berdasarkan fungsi objectiveFunction dari kelas main untuk memberikan penilaian awal dari state
2. Constructor child(child copy) : konstruktor ini membuat salinan dari child yang ada. Melalui pengaturan state dan heuristic agar sama dengan copy, konstruktor menyalin sequence dari copy menggunakan metode CopySeq, sehingga baru memiliki urutan yang identik dengan asli tanpa perubahan referensi
3. Constructor child(child before, int[] seq) : konstruktor ini membuat child baru dengan menyalin State dari child lain

(before) dan menerapkan perubahan dengan array seq, yang berisi koordinat elemen yang akan dipertukarkan dalam State. Setelah perubahan diterapkan maka nilai Heuristic akan diperbarui dan Sequence disalin dari before dan seq ditambahkan ke Sequence baru untuk mencatat perubahan tersebut

4. Metode CopySeq(List<int[]> seq) : untuk menghasilkan salinan mendalam dari list seq yang berisi array integer. Setiap elemen dalam seq disalin menggunakan Clone() dan ditambahkan ke list baru ret. Metode berikut memastikan bahwa Sequence yang dihasilkan benar-benar terpisah dari aslinya, sehingga perubahan pada salah satu list tidak memengaruhi list yang lain.

ChooseCubeByRandom

```
static child ChooseCubeByRandom(List<child> cubes)
{
    List<int> ranges = new List<int>();
    int maxH = 0;
    for (int i = 0; i < cubes.Count; ++i) if
(cubes[i].Heuristic > maxH) maxH = cubes[i].Heuristic;

    ranges.Add(maxH - cubes[0].Heuristic);
    for (int i = 1; i < cubes.Count; ++i) {
        ranges.Add(ranges[i - 1] + (maxH -
cubes[0].Heuristic));
    }
    int maxval = ranges[cubes.Count - 1];
    Random rand = new Random();
    double val = rand.Next(maxval);
```

```

        for (int i = 0; i < cubes.Count(); ++i) {
            if (val <= ranges[i]) return cubes[i];
        }
        return cubes[0];
    }
}

```

Tabel 2.2.17. Fungsi ChooseCubeByRandom untuk Genetic Algorithm

b. **ChooseCubeByRandom :**

- i. **Tugas :** digunakan untuk memilih child dari list cubes secara acak dengan menggunakan sistem probabilitas berdasarkan nilai heuristik. Cube dengan nilai heuristik lebih rendah memiliki peluang lebih tinggi untuk dipilih, sehingga proses seleksi cenderung memilih child yang lebih optimal

ii. **Parameter :**

1. List<child> cubes : list dari child yang masing-masing memiliki nilai heuristik. Fungsi ini akan memilih salah satu child dari list ini secara acak dengan peluang yang dipengaruhi oleh nilai heuristik masing-masing child

iii. **Cara kerja :**

1. Fungsi dimulai dengan mengidentifikasi nilai heuristik tertinggi (maxH) dari seluruh objek dalam list cubes
2. Setelah itu fungsi membuat list ranges yang menyimpan perbedaan antara maxH dan nilai heuristik setiap objek. List menciptakan rentang kumulatif yang mencerminkan peluang pemilihan setiap objek
3. Selanjutnya, fungsi membuat nilai acak val antara 0 dan maxval (nilai terakhir dalam ranges) menggunakan Random
4. Fungsi kemudian membandingkan val dengan nilai-nilai dalam ranges dan mengembalikan child pertama yang sesuai dengan rentang yang ditentukan

5. Jika val tidak sesuai dengan rentang mana pun fungsi akan mengembalikan objek pertama dalam list cubes sebagai default

Crossover

```
static (child, child) Crossover(child parent1,  
child parent2) {  
    child child1 = new child(parent1,  
parent2.Sequence.Last());  
    child child2 = new child(parent2,  
parent1.Sequence.Last());  
  
    return (child1, child2);  
}
```

Tabel 2.2.18. Fungsi Crossover untuk Genetic Algorithm

c. **Crossover :**

- i. **Tugas :** menghasilkan dua child baru dari dua child induk (parent1 dan parent2) dalam algoritma genetika. Hal tersebut merupakan bagian dari proses reproduksi yang kedua individu induk dikombinasikan untuk menciptakan dua keturunan yang mungkin mewarisi fitur dari masing-masing induk

ii. **Parameter :**

1. child parent1 : child pertama yang berfungsi sebagai salah satu induk dalam proses crossover
2. child parent2 : child kedua yang berfungsi sebagai induk kedua dalam proses crossover

iii. **Cara kerja :**

1. Fungsi pertama-tama membuat child1 sebagai salinan dari parent1 tetapi elemen terakhir dari Sequence diambil dari

- parent2. Hal tersebut dapat memberikan variasi dengan sedikit modifikasi dari karakteristik induk
2. Kemudian child2 dibuat sebagai salinan dari parent2 dengan elemen terakhir dari Sequence diambil dari parent1
 3. Terakhir, fungsi mengembalikan tuple (child1, child2), yang merupakan dua child baru yang dihasilkan dari proses crossover antara parent1 dan parent2

Mutation

```

static Child Mutation(Child cube) {
    Random rand = new Random();
    int i1 = rand.Next(5), j1 = rand.Next(5), k1 =
rand.Next(5);
    int i2 = rand.Next(5), j2 = rand.Next(5), k2 =
rand.Next(5);

    Child ret = new Child(cube, new[] { i1, j1, k1,
i2, j2, k2 });
    return ret;
}

```

Tabel 2.2.19. Fungsi Mutation untuk Genetic Algorithm

d. Mutation :

- i. **Tugas** : menghasilkan versi mutasi dari child yang diberikan. Mutasi ini dilakukan dengan mengganti posisi elemen dalam State dari child tersebut secara acak yang bertujuan untuk memberikan variasi dan meningkatkan eksplorasi solusi dalam algoritma genetika
- ii. **Parameter** :

1. child cube : child yang akan dimutasi dari populasi algoritma genetika yang akan mengalami perubahan pada atribut State

iii. **Cara kerja :**

1. Fungsi ini menggunakan Random untuk menentukan dua set koordinat acak (i_1, j_1, k_1) dan (i_2, j_2, k_2), yang mewakili posisi elemen dalam State dari cube
2. Kemudian, fungsi membuat child baru dengan konstruksi new child(cube, new[] { $i_1, j_1, k_1, i_2, j_2, k_2$ }). Konstruktor menggunakan kedua set koordinat tersebut untuk menukar posisi elemen dalam State sehingga menghasilkan child baru dengan elemen-elemen yang berbeda posisi dari cube asli
3. Fungsi akhirnya mengembalikan child hasil mutasi (ret)

Clone

```
static List<child> Clone(List<child> source) {
    List<child> target = new List<child>();
    for (int i = 0; i < source.Count; ++i) {
        child childCopy = new child(source[i]);
        target.Add(childCopy);
    }
    return target;
}
```

Tabel 2.2.20. Fungsi Clone untuk Genetic Algorithm

e. **Clone :**

- i. **Tugas :** membuat salinan (deep copy) dari list child yang diberikan (source). Sehingga setiap child dalam list baru (target) adalah salinan terpisah sehingga perubahan pada list hasil Clone tidak akan memengaruhi list asli

ii. **Parameter :**

1. List<child> source : list yang berisi child yang akan disalin.
Setiap child dalam list ini akan dibuat salinannya untuk ditempatkan di list baru

iii. **Cara kerja :**

1. Fungsi membuat terlebih dahulu list kosong bernama target untuk menyimpan salinan child
2. Untuk setiap elemen dalam source fungsi akan membuat salinan dari child tersebut menggunakan konstruktor copy (new child(source[i])) dan menambahkan salinan ini ke dalam target
3. Setelah semua elemen dalam source disalin, fungsi mengembalikan target yang berisi semua child hasil salinan dari source

Run

```
public static List<int[]> Run(int[, ,] values, int population = 100, int generation = 500) {  
    List<child> lastPopulation = new List<child>();  
    List<child> currentPopulation = new List<child>();  
    child initial = new child(values);  
    for (int i = 0; i < population; ++i) {  
        lastPopulation.Add(Mutation(initial));  
    }  
    for (int g = 0; g < generation; ++g) {  
        // Outputs 55% of the children  
        for (int i = 0; i < Mathf.RoundToInt(population * 0.55f); ++i) {  
            child current =
```

```

        Mutation(ChooseCubeByRandom(lastPopulation));
                    currentPopulation.Add(current);
    }

    // Outputs 40% of the children
    for (int i = 0; i <
Mathf.RoundToInt(population * 0.2f); ++i) {
    child parent1 =
ChooseCubeByRandom(lastPopulation);
    child parent2 =
ChooseCubeByRandom(lastPopulation);
    (child, child) children =
Crossover(parent1, parent2);
    currentPopulation.Add(children.Item1);
    currentPopulation.Add(children.Item2);
}

// Outputs 5% of the children
for (int i = 0; i <
Mathf.RoundToInt(population * 0.05f); ++i) {
    child current =
ChooseCubeByRandom(lastPopulation);
    currentPopulation.Add(current);
}

lastPopulation.Clear();
lastPopulation = Clone(currentPopulation);
currentPopulation.Clear();
}

int bestHeuristic = Int32.MaxValue;
int bestIdx = 0;
for (int i = 0; i < lastPopulation.Count; ++i)

```

```

{
    if (lastPopulation[i].Heuristic <
bestHeuristic) {
        bestHeuristic = lastPopulation[i].Heuristic;
        bestIdx = i;
    }
    return lastPopulation[bestIdx].Sequence;
}

```

Tabel 2.2.21. Fungsi Run untuk Genetic Algorithm

f. Run :

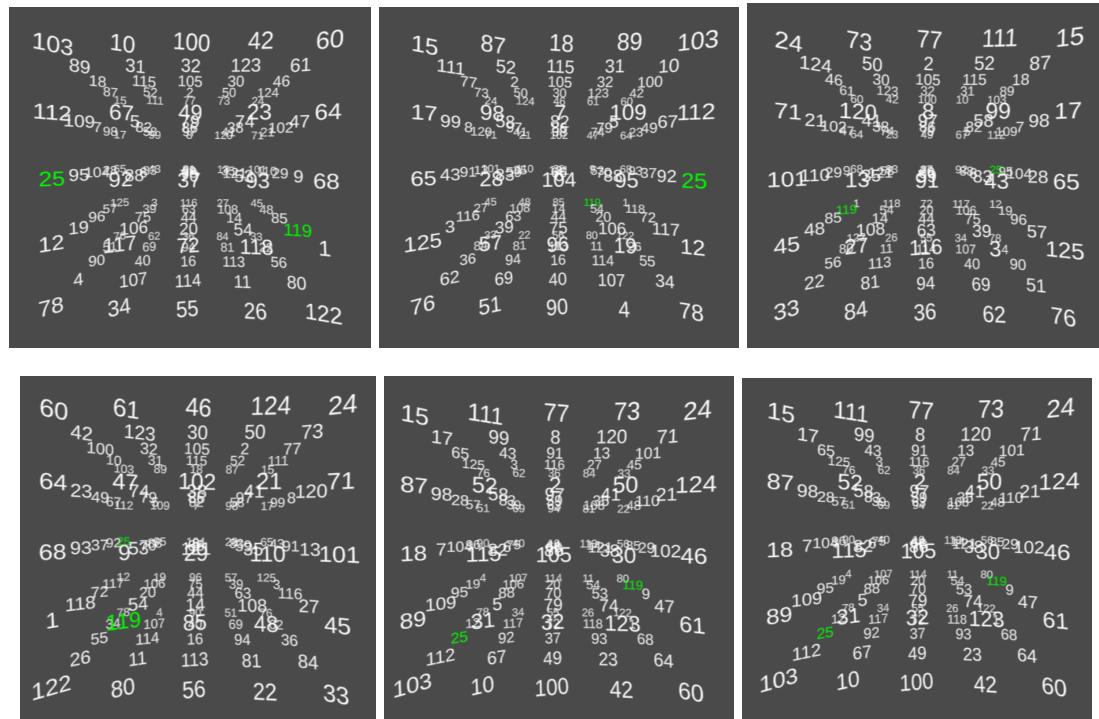
- i. **Tugas :** menjalankan proses utama dari algoritma genetika untuk mencari solusi optimal dalam suatu ruang solusi berdasarkan nilai heuristik. Lalu akan melakukan iterasi selama sejumlah generasi dengan proses mutasi, crossover, dan seleksi, dan akhirnya mengembalikan aksi (Sequence) dari solusi terbaik yang ditemukan
- ii. **Parameter :**
 1. int[,,] values : array tiga dimensi yang digunakan sebagai State awal untuk setiap child
 2. int population (opsional, default 100) : jumlah individu dalam populasi awal
 3. int generation (opsional, default 500) : jumlah generasi yang akan diiterasi oleh algoritma untuk menghasilkan solusi
- iii. **Cara kerja :**
 1. Fungsi memulai dengan membuat populasi awal lastPopulation dari setiap individu dalam populasi dihasilkan dari initial yang dimutasi

2. Untuk setiap generasi, currentPopulation diisi dengan individu baru yang dihasilkan dari:
 - a. Mutasi : 55% dari populasi baru dihasilkan melalui mutasi dari individu yang dipilih secara acak dari lastPopulation
 - b. Crossover : 40% dari populasi baru dihasilkan melalui crossover antara dua individu yang dipilih secara acak
 - c. Seleksi langsung : 5% dari populasi baru diisi dengan individu yang dipilih langsung dari lastPopulation tanpa modifikasi
3. Setelah setiap generasi selesai, lastPopulation diperbarui dengan copy dari currentPopulation dan currentPopulation dikosongkan
4. Setelah semua generasi selesai fungsi akan mencari individu dengan nilai Heuristic terendah dalam lastPopulation,yang mewakili solusi terbaik yang ditemukan
5. Fungsi akhirnya mengembalikan Sequence dari individu terbaik tersebut

C. Hasil eksperimen dan analisis

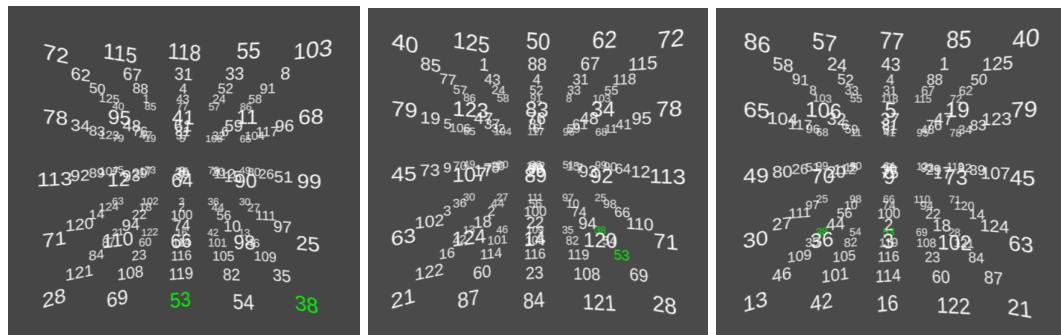
a. State awal dari setiap kubus:

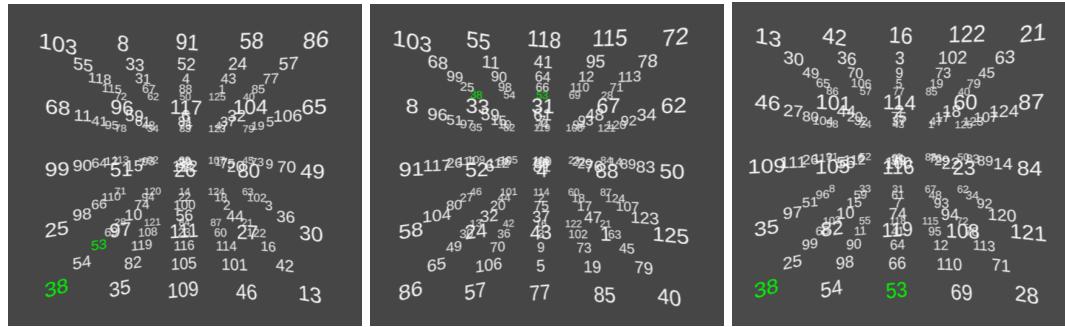
Test Case 1



Gambar 2.3.1. Kondisi awal test case 1

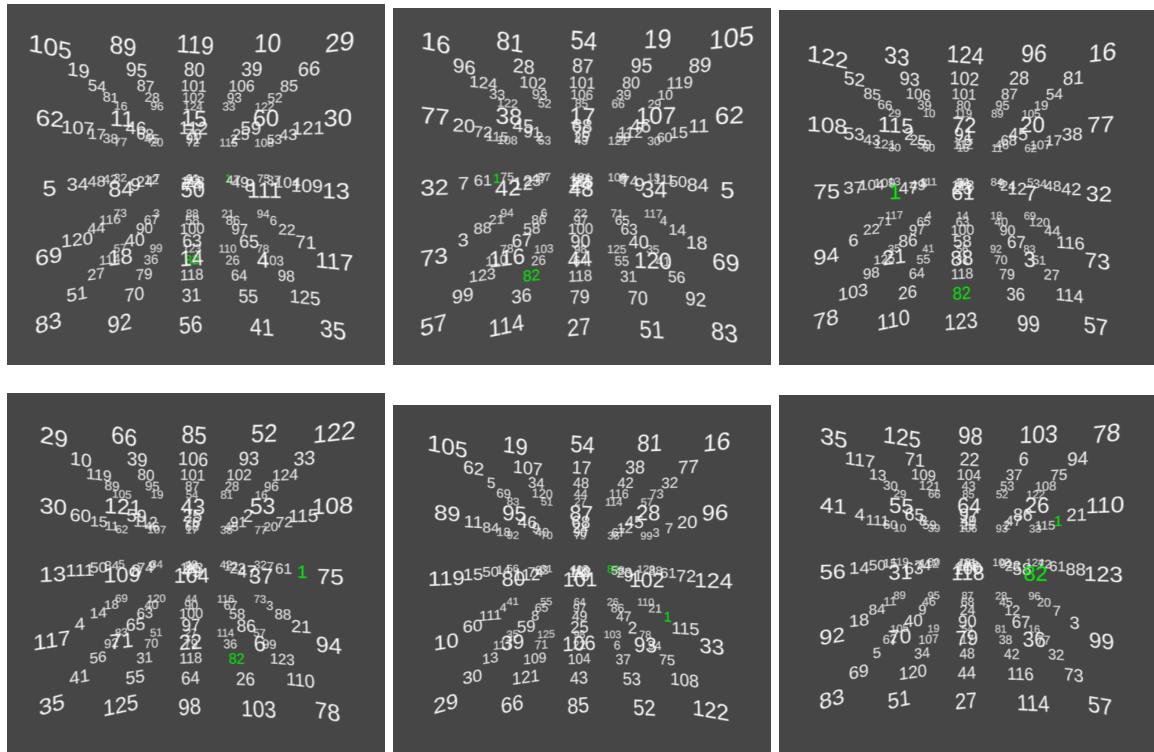
Test Case 2





Gambar 2.3.2. Kondisi awal test case 2

Test Case 3



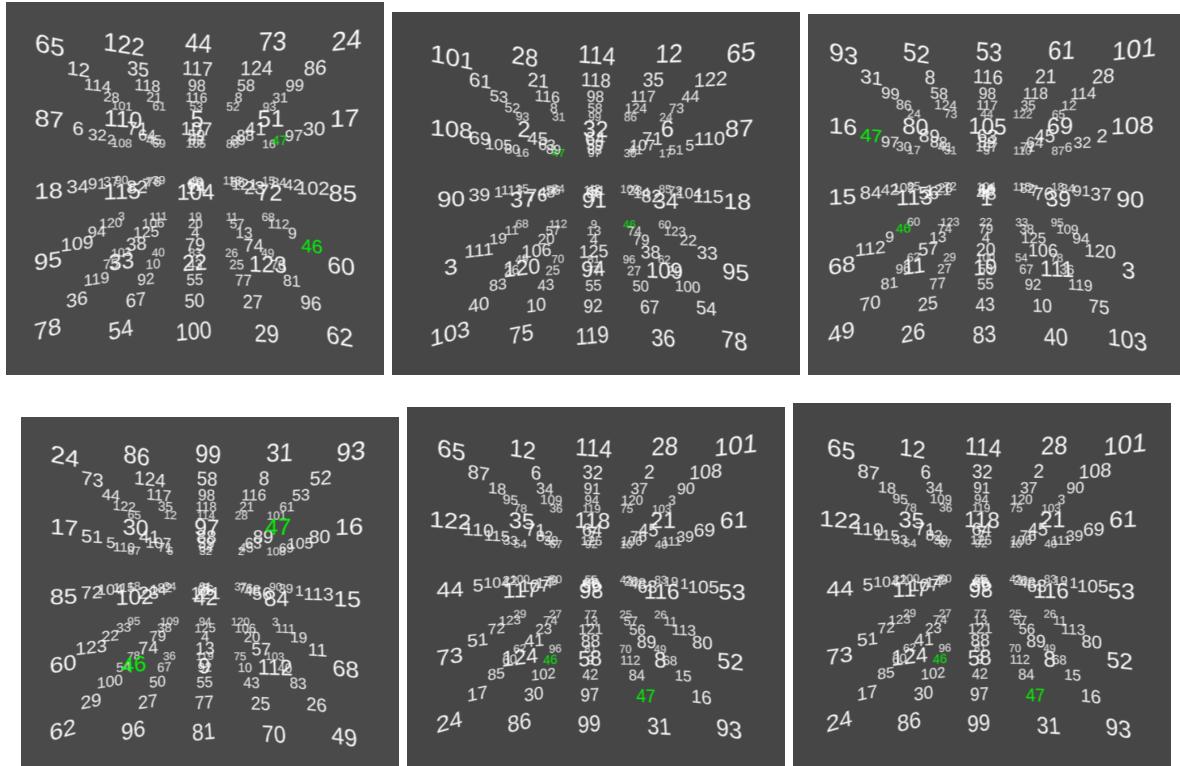
Gambar 2.3.3. Kondisi awal test case 3

1. Steepest Ascent Hill-Climbing with Sideways Move

Pada pengujian dari algoritma *Steepest Ascent Hill-Climbing with Sideways Move* akan menggunakan parameter *current state* dari *magic cube* dan koordinat kubus pertama dan kedua yang akan ditukar.

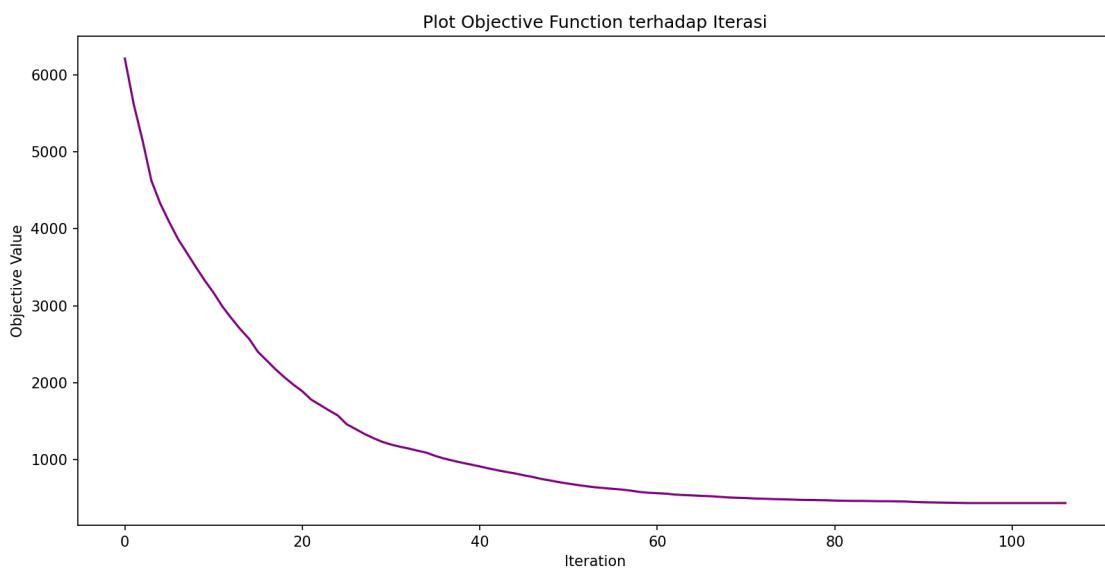
Test Case 1

- Hasil akhir :



Gambar 2.3.4. Hasil akhir case 1 Sideways Move Hill Climbing

- Nilai objective function akhir yang dicapai : 438
- Plot nilai objective function terhadap banyak iterasi yang telah dilewati.

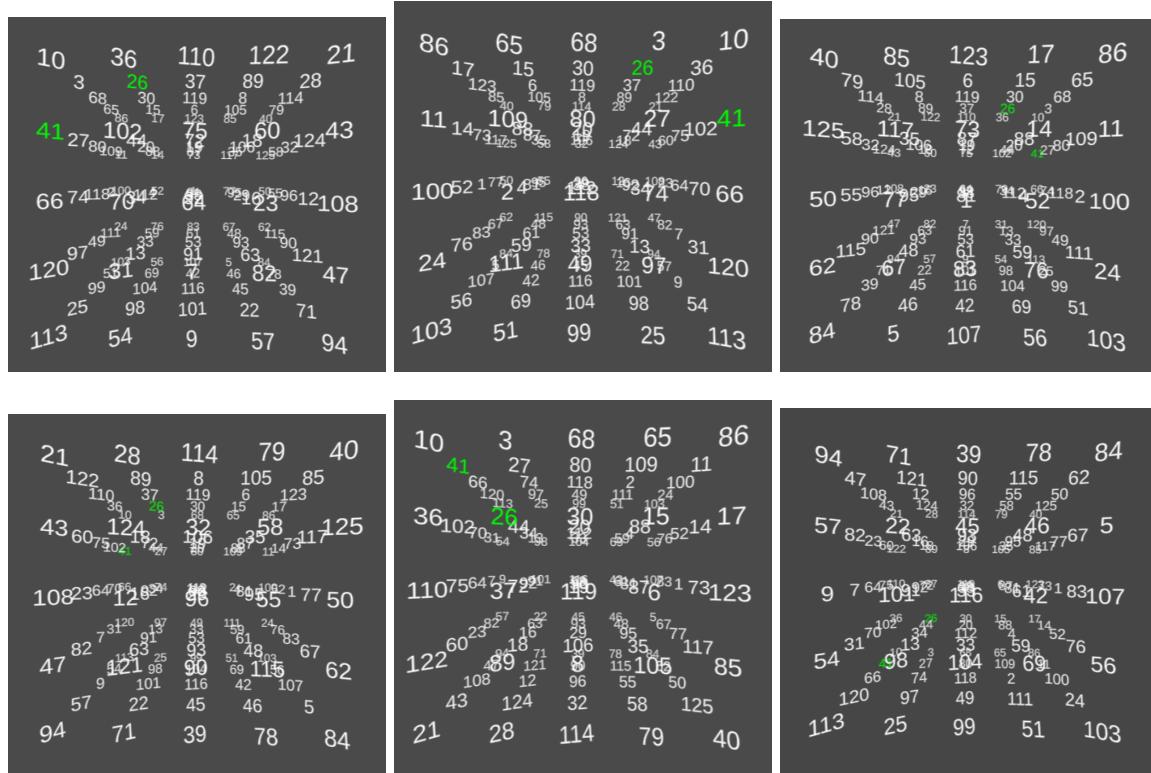


Gambar 2.3.5. Plot nilai *objective function* terhadap banyak iterasi

- **Durasi proses pencarian :** 6113 milisekon
- **Banyak iterasi hingga berhenti :** 106

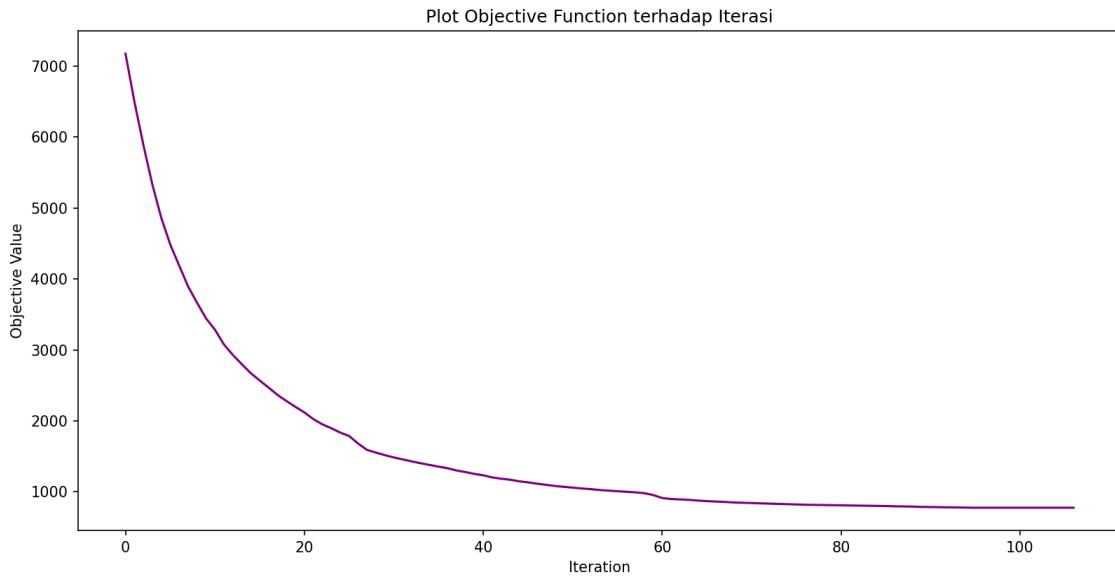
Test Case 2

- **Hasil akhir**



Gambar 2.3.6. Hasil akhir case 2 *Sideways Move Hill Climbing*

- **Nilai objective function akhir yang dicapai :** 777
- **Plot nilai objective function terhadap banyak iterasi yang telah dilewati.**

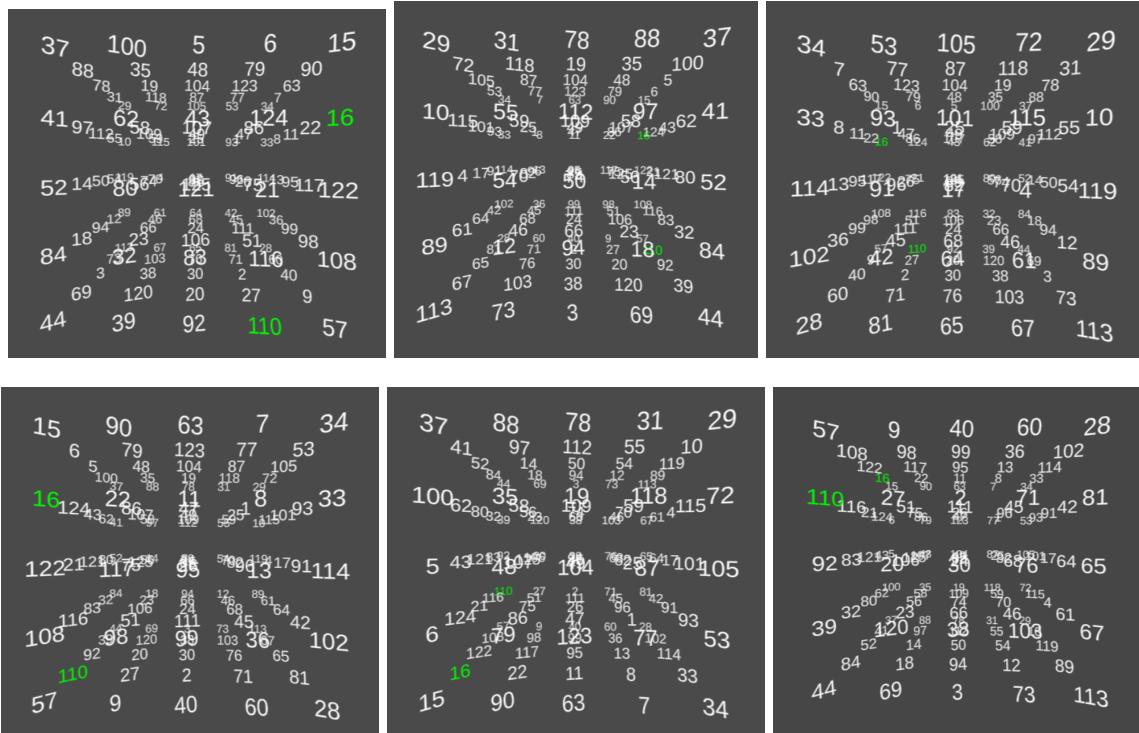


Gambar 2.3.7. Plot nilai *objective function* terhadap banyak iterasi

- **Durasi proses pencarian :** 6206 milisekom
- **Banyak iterasi hingga berhenti :** 106

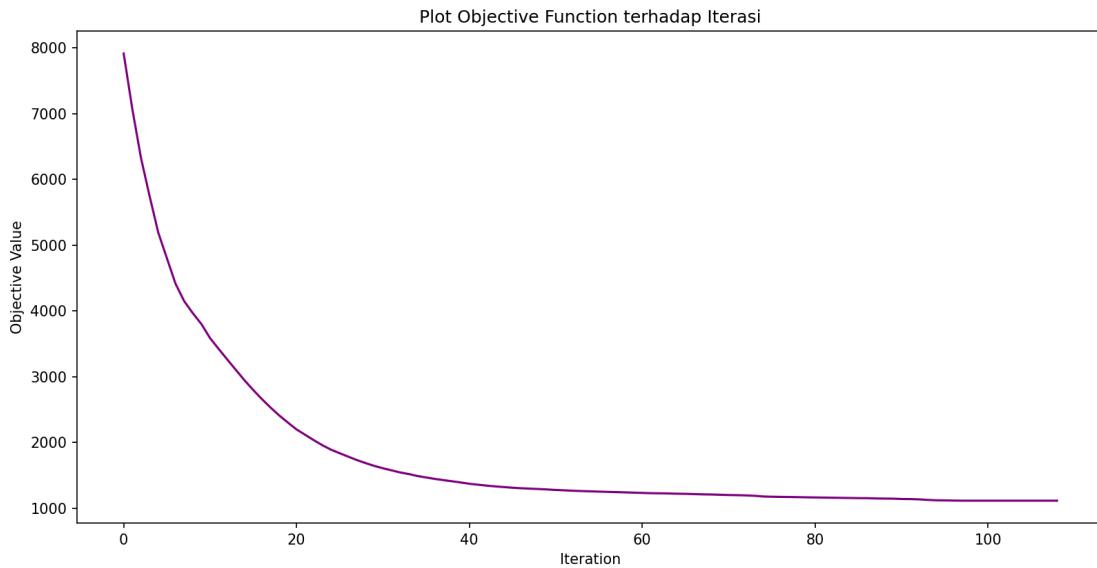
Test Case 3

- **Hasil akhir**



Gambar 2.3.8. Hasil akhir case 3 *Sideways Move Hill Climbing*

- **Nilai objective function akhir yang dicapai : 1114**
- **Plot nilai objective function terhadap banyak iterasi yang telah dilewati.**



Gambar 2.3.9. Plot nilai *objective function* terhadap banyak iterasi

- **Durasi proses pencarian : 6278 milisekon**
- **Banyak iterasi hingga berhenti : 108**

Hasil Analisis Steepest Ascent Hill-Climbing with Sideways Move

Berdasarkan analisis yang ada, algoritma ini menunjukkan kemampuan yang terbatas dalam mendekati global optima. Meskipun *sideways move* memungkinkan pergerakan ke titik yang memiliki *objective value* sama, algoritma ini tetap cenderung terjebak di local optima, terutama pada struktur masalah kompleks seperti magic cube. Hal ini terjadi karena algoritma hanya mengevaluasi satu solusi terbaik dalam lingkup lokal tanpa mekanisme eksplorasi yang lebih luas.

Dari segi durasi, Steepest Ascent Hill-Climbing dengan sideways move bekerja relatif cepat karena pendekatannya sederhana dan langsung menuju perbaikan secara iteratif. Namun, kecepatannya tidak diimbangi dengan kemampuan mencapai hasil optimal secara konsisten. Algoritma ini sering kali memberikan hasil akhir yang variatif dalam tiap eksperimen, tergantung pada posisi awal dan distribusi sideways move yang dijalankan.

Berdasarkan ketiga test case yang ada, solusi terbaik berada pada nilai 777 dengan jumlah iterasi sebanyak 106 kali dalam waktu 6206 milisekon.

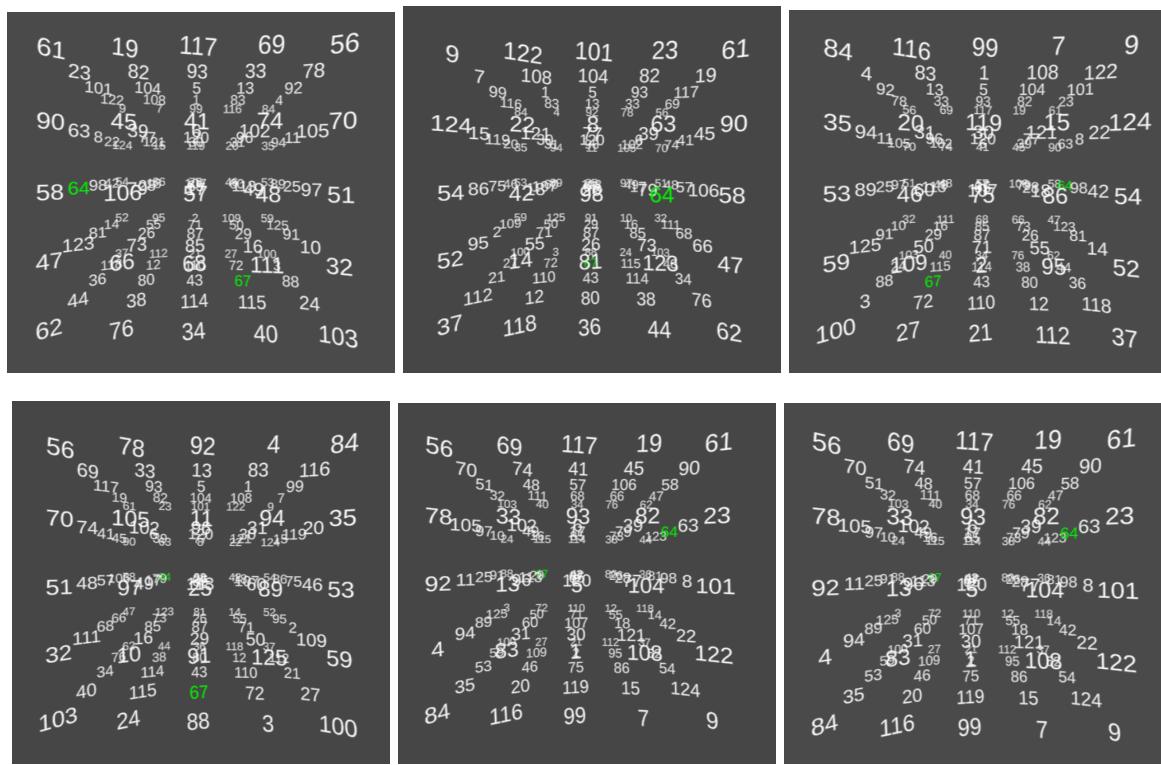
2. Simulated Annealing

Pada pengujian dari algoritma ***Simulated Annealing*** akan menggunakan parameter dan initial state yang berbeda-beda :

Test Case 1

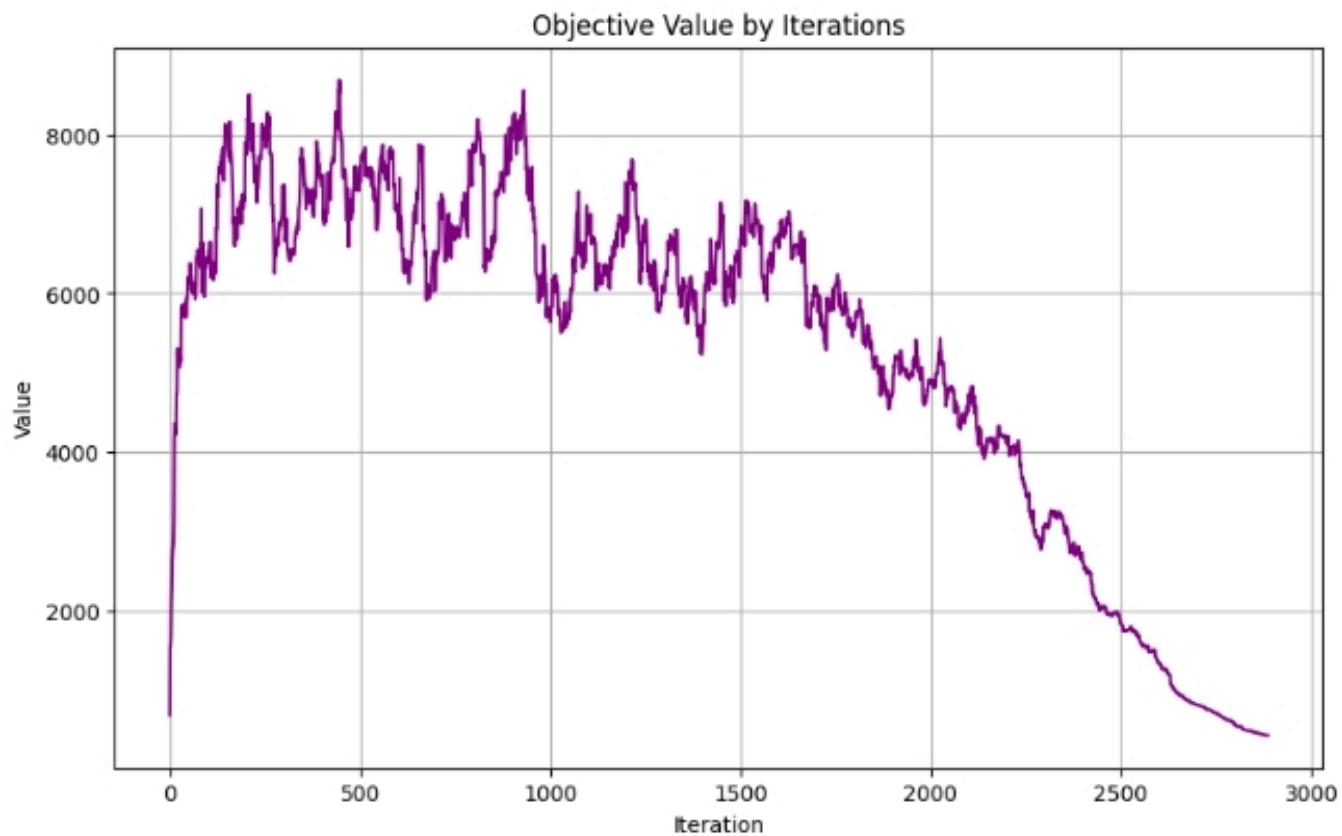
T0 = 1000, Cooling Rate = 0.999, dan threshold (T1) = 0.1.

- **Hasil Akhir**



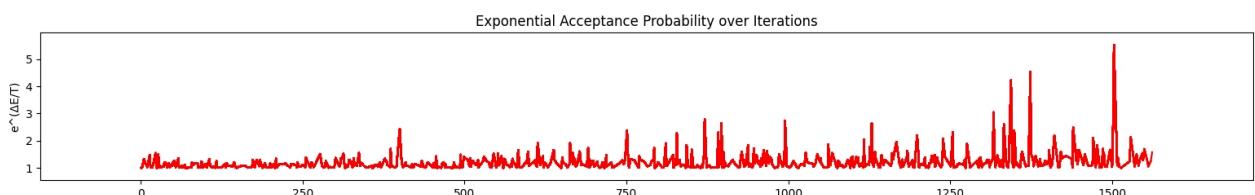
Gambar 2.3.10. Hasil akhir case 1 *Simulated Annealing*

Pada percobaan terhadap kubus 1 dilakukan plot *objective function* terhadap setiap iterasi yang dapat dilihat sebagai berikut :



Gambar 2.3.11. Plot *objective function* terhadap iterasi *Simulated Annealing*

Setelah itu dilakukan plot dari $e^{\Delta E/t}$ terhadap setiap iterasi sebagai berikut :

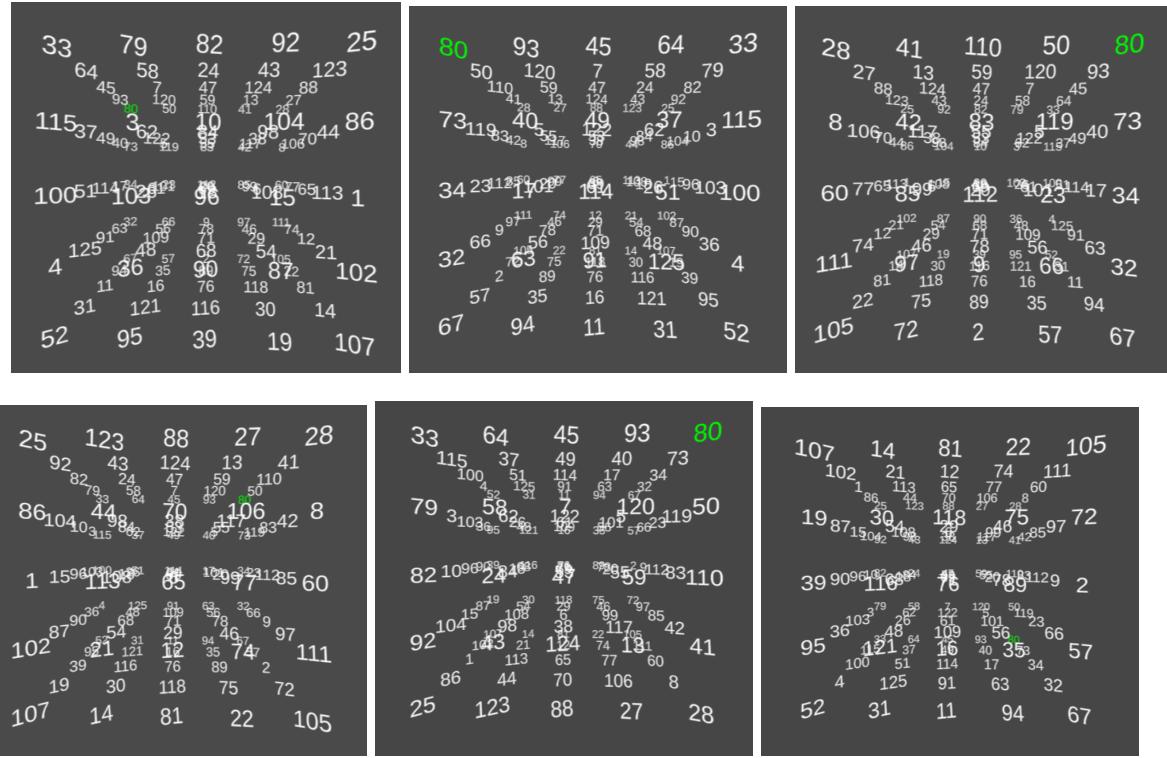


Gambar 2.3.12. Plot $e^{\Delta E/t}$ terhadap setiap iterasi

Dari percobaan ini dapat dilihat bahwa dari kubus 1 apabila menggunakan algoritma ***simulated annealing*** didapatkan perubahan cost dari 1543 ke 431 dengan waktu yang dibutuhkan yaitu 156 detik dan frekuensi *stuck* di *local optima* yaitu sebanyak 427 kali dan jumlah iterasi yang diperlukan yaitu 2888 iterasi.

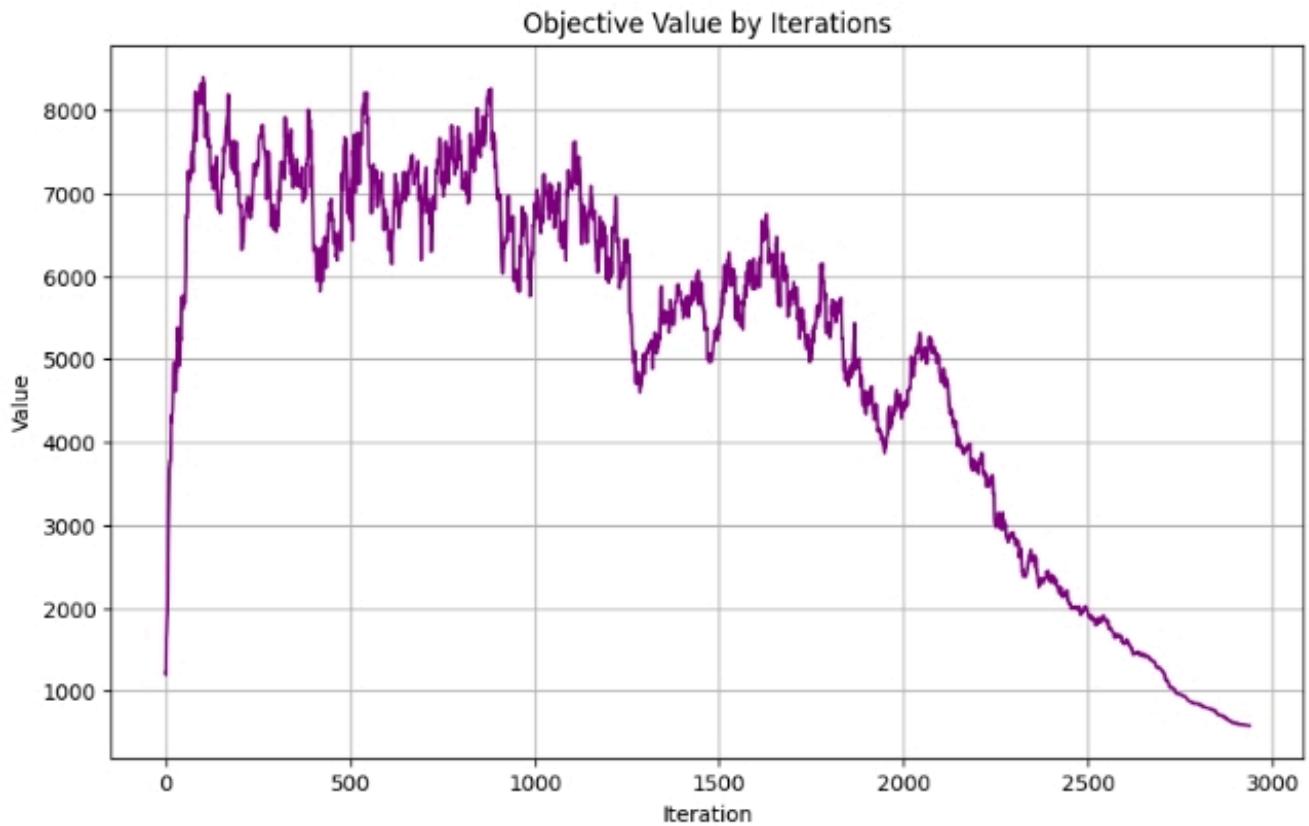
Test Case 2

• Hasil Akhir



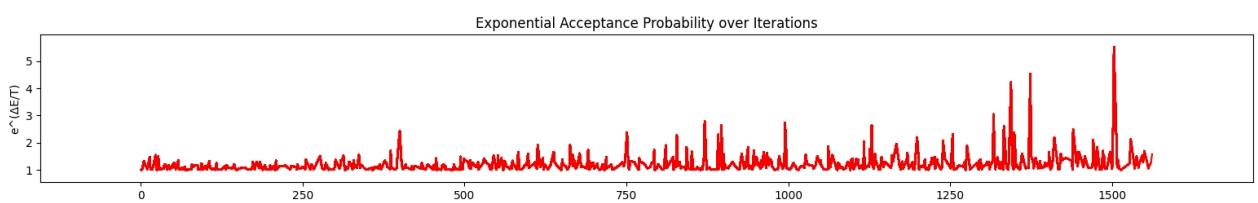
Gambar 2.3.13. Hasil akhir case 2 *Simulated Annealing*

Pada percobaan terhadap kubus 1 dilakukan plot *objective function* terhadap setiap iterasi yang dapat dilihat sebagai berikut :



Gambar 2.3.14. Plot *objective function* terhadap iterasi *Simulated Annealing*

Setelah itu dilakukan plot dari $e^{\Delta E/t}$ terhadap setiap iterasi sebagai berikut :

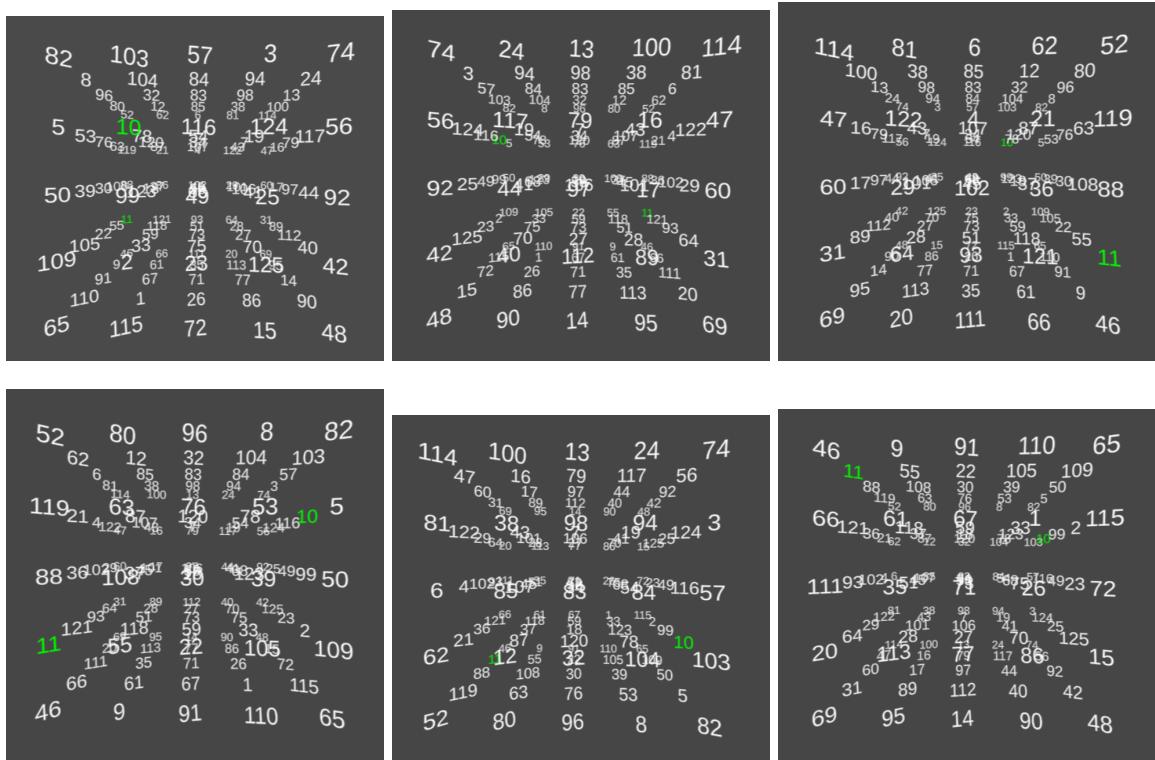


Gambar 2.3.15. Plot $e^{\Delta E/t}$ terhadap setiap iterasi

Dari percobaan ini dapat dilihat bahwa dari kubus 2 apabila menggunakan algoritma ***simulated annealing*** didapatkan perubahan cost dari 1227 ke 581 dengan waktu yang dibutuhkan yaitu 135 detik dan frekuensi *stuck* di *local optima* yaitu sebanyak 439 kali dan jumlah iterasi yang diperlukan yaitu 2937 iterasi.

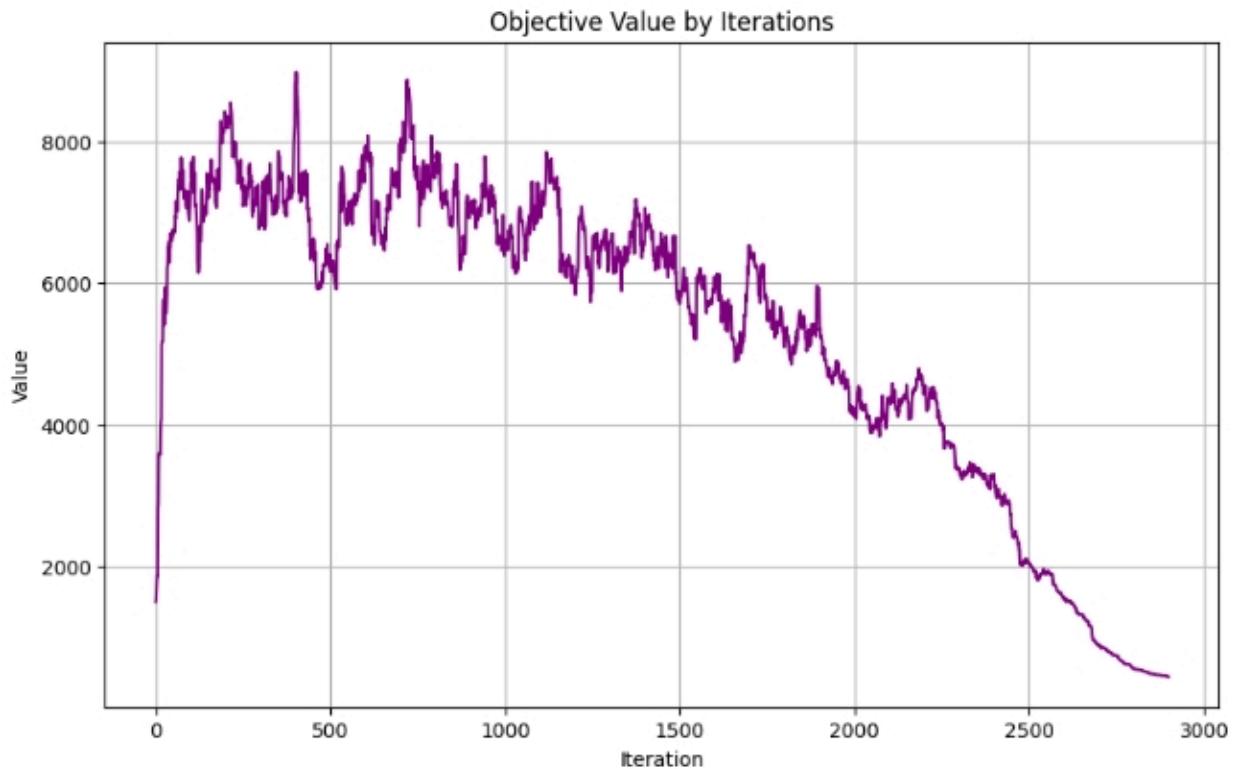
Test Case 3

- **Hasil Akhir**



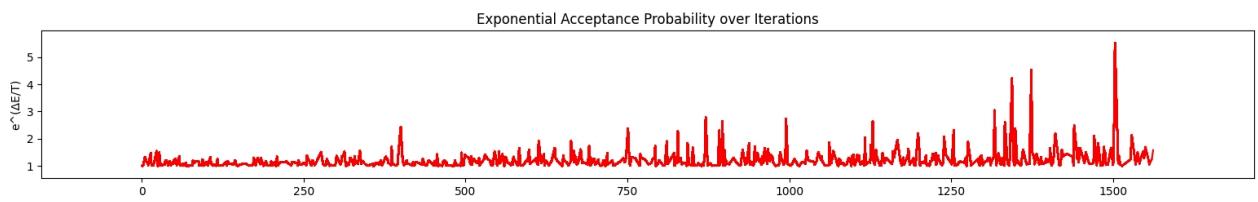
Gambar 2.3.16. Hasil akhir case 3 *Simulated Annealing*

Pada percobaan terhadap kubus 3 dilakukan plot *objective function* terhadap setiap iterasi yang dapat dilihat sebagai berikut :



Gambar 2.3.17. Plot *objective function* terhadap iterasi *Simulated Annealing*

Setelah itu dilakukan plot dari $e^{\Delta E/t}$ terhadap setiap iterasi sebagai berikut :



Gambar 2.3.18. Plot $e^{\Delta E/t}$ terhadap setiap iterasi

Dari percobaan ini dapat dilihat bahwa dari kubus 3 apabila menggunakan algoritma ***simulated annealing*** didapatkan perubahan cost dari 1505 ke 446 dengan waktu yang dibutuhkan yaitu 146 detik dan frekuensi *stuck* di *local optima* yaitu sebanyak 372 kali dengan jumlah iterasi sebanyak 2898 iterasi.

Hasil Analisis Simulated Annealing

T1 = 1000, temperature = 0.999, threshold = 0.01

Test Case	Cost Awal	Cost Setelah	Waktu	Total Stuck	Jumlah Iterasi
1	1543	431	156 ms	427	2888
2	1227	581	135 ms	439	2937
3	1505	446	146 ms	372	2898

Algoritma **Simulated Annealing** yang diterapkan pada konfigurasi *magic cube* menunjukkan hasil yang efektif dalam menurunkan nilai *cost* awal menjadi nilai akhir yang lebih rendah secara konsisten. Berdasarkan tiga percobaan, nilai *cost* awal masing-masing berada di angka 1543, 1227, dan 1505, yang kemudian berhasil diturunkan menjadi 431, 581, dan 446. Hal ini menunjukkan bahwa algoritma berhasil mencapai optimasi yang signifikan pada setiap percobaan.

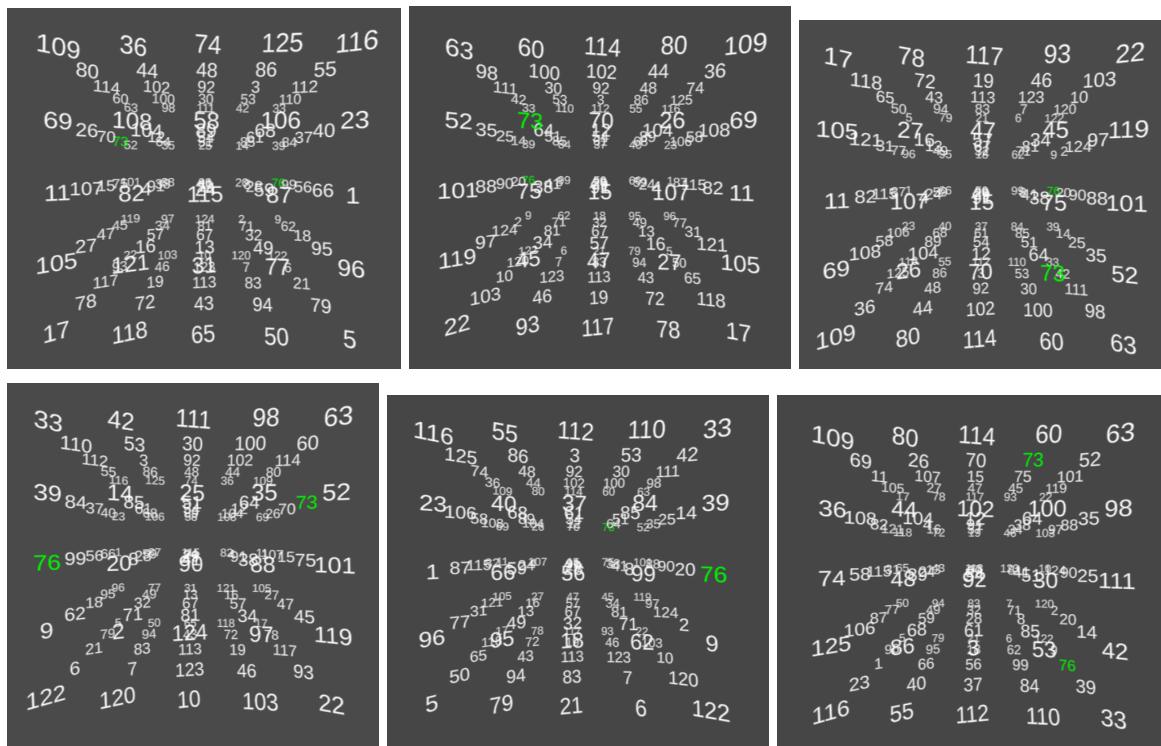
Waktu eksekusi yang dicatat pada tiap percobaan berkisar antara 135.000 ms hingga 156.000 ms, yang menunjukkan konsistensi kinerja dengan sedikit variasi, mungkin disebabkan oleh faktor acak atau jalur solusi yang berbeda pada setiap iterasi. Selain itu, metrik "Total Stuck" yang menunjukkan frekuensi terjebaknya algoritma dalam local optima memiliki nilai yang bervariasi, yakni 427, 439, dan 372, yang mengindikasikan adanya tingkat kesulitan berbeda dalam menghindari minima lokal di tiap percobaan. Jumlah iterasi pada tiap percobaan juga relatif konsisten, dengan angka 2888, 2937, dan 2898, yang menunjukkan bahwa algoritma beroperasi pada langkah-langkah yang cukup panjang sebelum mencapai konvergensi atau kriteria berhenti. Secara keseluruhan, algoritma **Simulated Annealing** ini terbukti efektif dalam mengurangi *cost* pada konfigurasi yang diuji, meskipun masih terdapat ruang untuk peningkatan, seperti penyesuaian parameter lebih lanjut dan eksperimen dengan jadwal pendinginan yang berbeda untuk mengoptimalkan kinerja.

3. Genetic Algorithm

Pada pengujian dari algoritma ***Genetic Algorithm*** akan menggunakan parameter populasi awal (konstan bernilai 100) dan jumlah maksimum iterasi yang dilakukan.

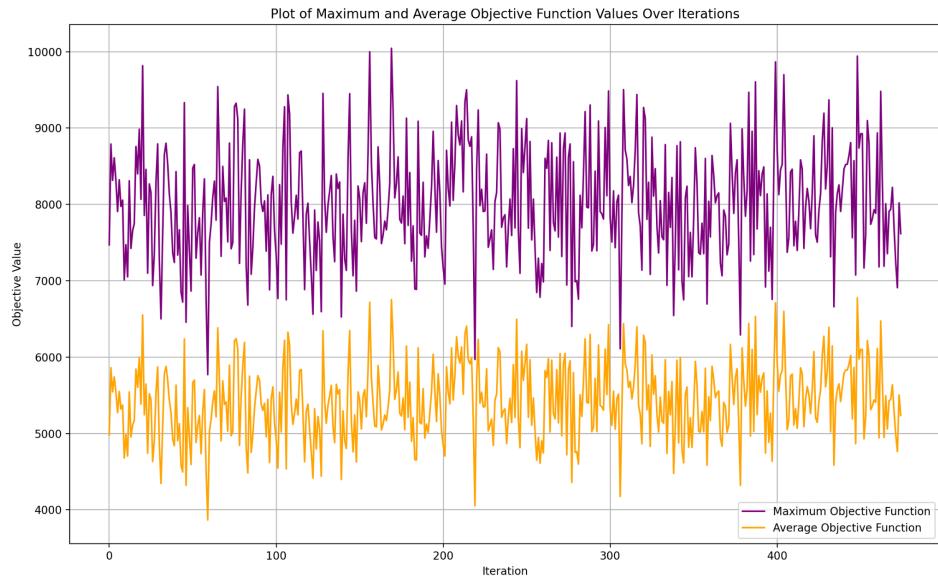
Test Case 1

- Hasil akhir :



Gambar 2.3.19. Hasil akhir case 1 *Genetic Algorithm*

- Jumlah populasi : 500
- Nilai objective function akhir yang dicapai : 7617
- Plot nilai objective function terhadap banyak iterasi yang telah dilewati.

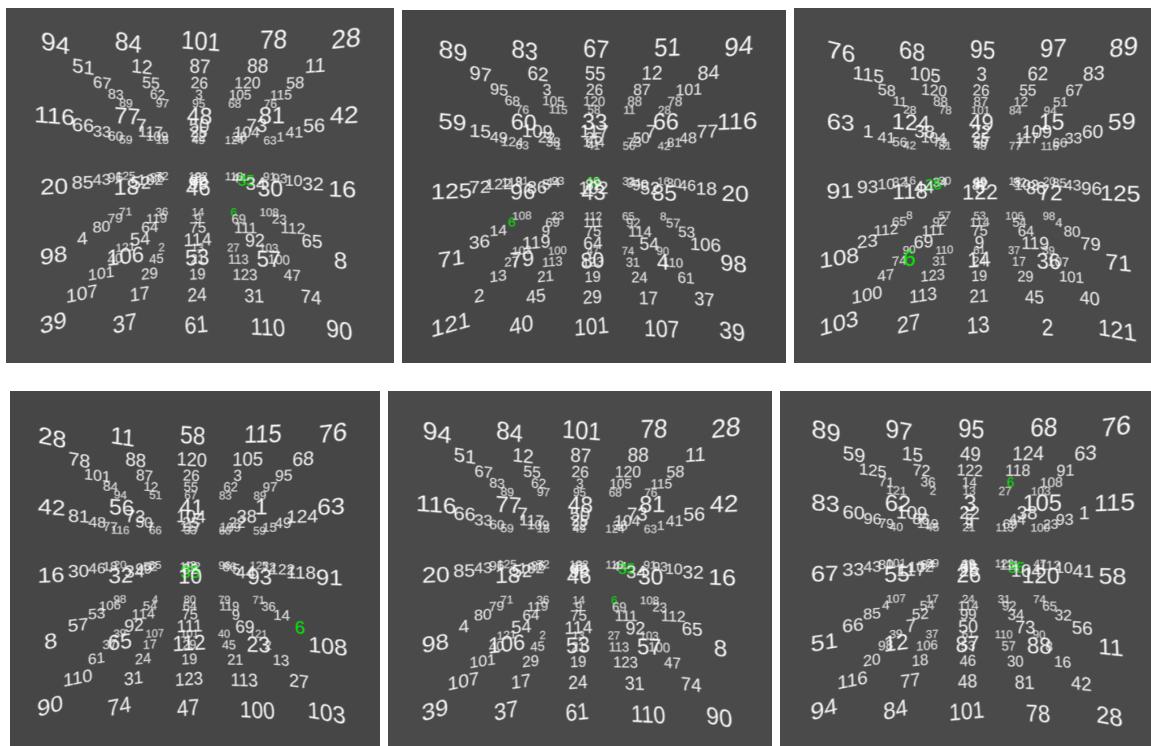


Gambar 2.3.20. Plot hasil *objective function* terhadap *iterations*

- Banyak iterasi : 474
- Durasi proses pencarian : 8453 ms

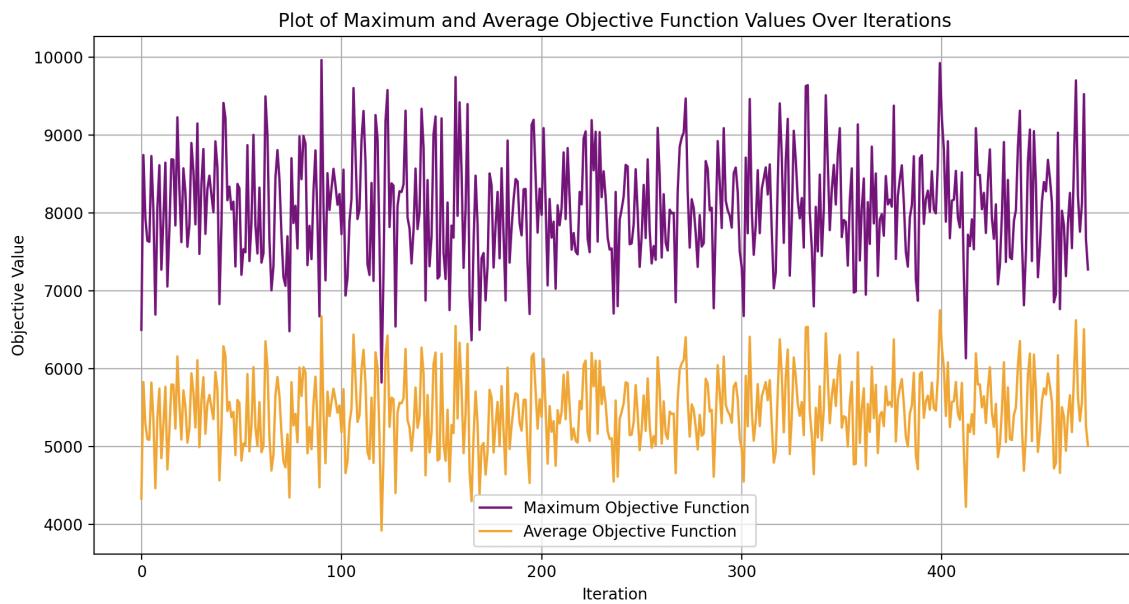
Test Case 2

- Hasil akhir :



Gambar 2.3.21. Hasil akhir case 2 *Genetic Algorithm*

- Jumlah populasi : 500
- Nilai objective function akhir yang dicapai : 7277
- Plot nilai objective function terhadap banyak iterasi yang telah dilewati.

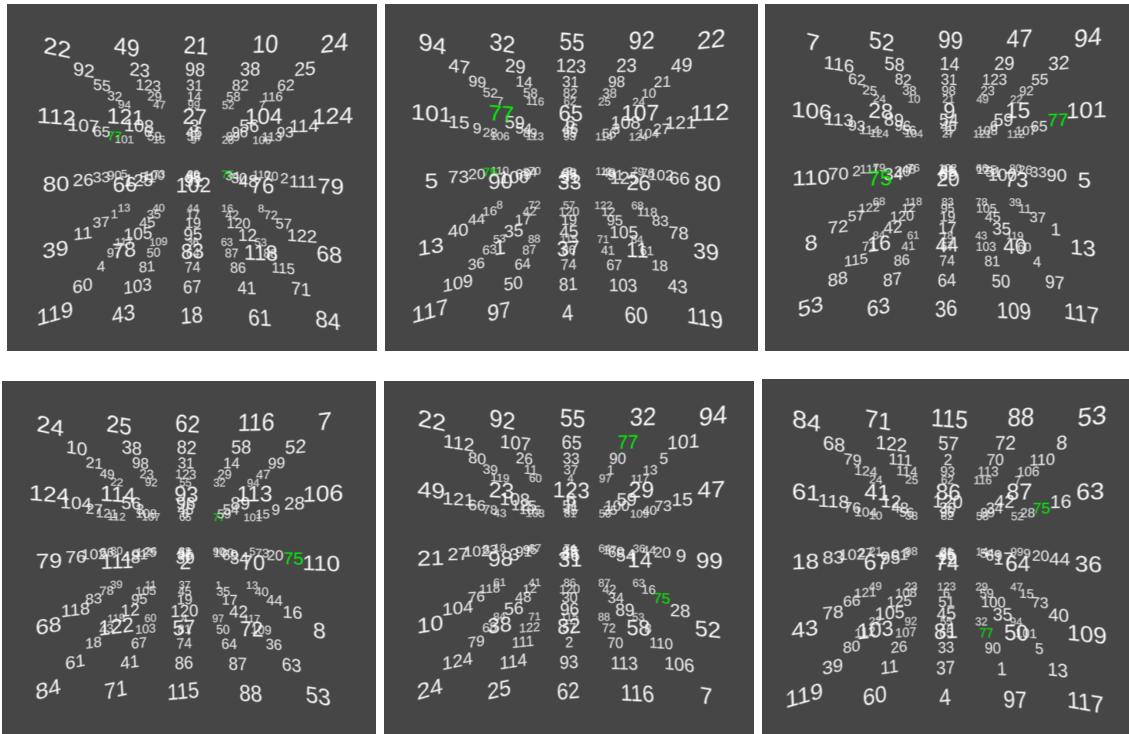


Gambar 2.3.22. Plot hasil *objective function* terhadap *iterations*

- Banyak iterasi : 473
- Durasi proses pencarian : 8687 ms

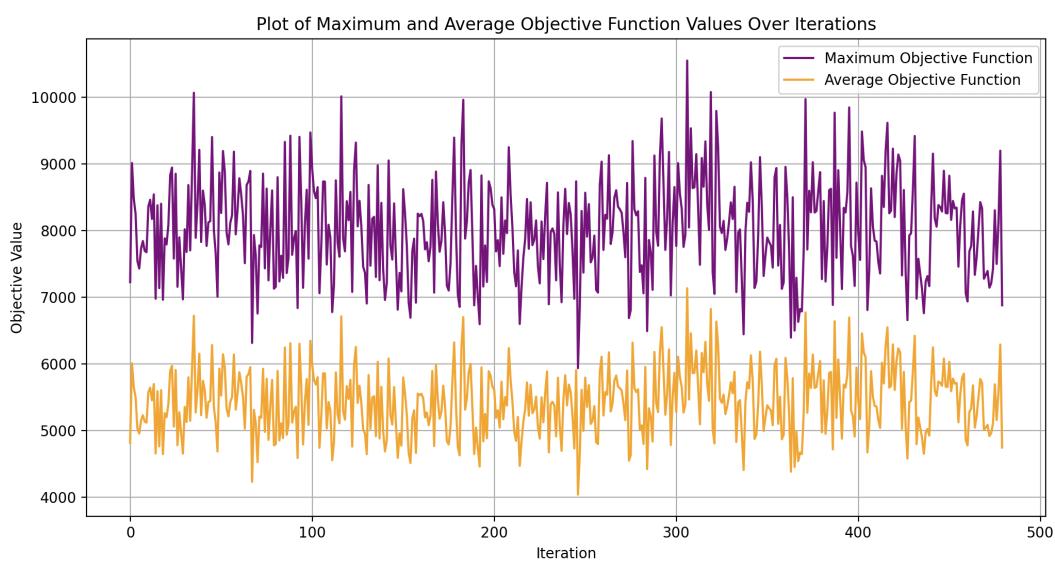
Test Case 3

- Hasil akhir :



Gambar 2.3.23. Hasil akhir case 3 *Genetic Algorithm*

- Jumlah populasi : 500
- Nilai objective function akhir yang dicapai : 6883
- Plot nilai objective function terhadap banyak iterasi yang telah dilewati.



Gambar 2.3.24. Plot hasil *objective function* terhadap *iterations*

- Banyak iterasi : 479
- Durasi proses pencarian : 8527 ms

Hasil Analisis Genetic Algorithm

Algoritma **Genetic Algorithm** menunjukkan kinerja yang baik dalam mengoptimalkan nilai objective function pada tiga test case yang diberikan. Dengan parameter awal berupa populasi konstan sebesar 100 dan iterasi maksimum 500, algoritma ini menghasilkan nilai objective function akhir yang semakin optimal pada setiap test case. Pada Test Case 1, nilai objective function yang dicapai adalah 7617 setelah 474 iterasi dengan durasi proses pencarian 8453 ms, sementara pada Test Case 2, nilai akhir yang lebih baik sebesar 7277 dicapai dalam 473 iterasi dengan durasi 8687 ms. Hasil terbaik ada pada Test Case 3 dengan nilai objective function akhir 6883 setelah 479 iterasi dan durasi 8527 ms.

Setiap generasi dalam algoritma ini terdiri dari 55% individu hasil mutasi, 40% dari proses crossover, dan 5% dari seleksi individu dari populasi sebelumnya, yang menjaga keseimbangan antara eksplorasi dan eksploitasi solusi. Proses iteratif digunakan supaya untuk memastikan adanya diversifikasi dalam pencarian solusi, sementara sebagian kecil dari populasi terbaik dipertahankan untuk stabilitas. Implementasi algoritma yang meliputi mutation, crossover, dan selection ini mampu mencapai konvergensi pada batas iterasi maksimum, meskipun terdapat sedikit variasi pada nilai akhir antar test case.

Secara keseluruhan, Genetic Algorithm yang diimplementasikan menunjukkan efektivitas dalam menemukan solusi optimal secara cepat, walaupun terdapat peluang untuk penyetelan parameter lebih lanjut, seperti probabilitas crossover dan mutasi, untuk mencapai hasil yang lebih konsisten dan optimal pada setiap test case.

BAB III

Analisis

A. Analisis Global Optima Setiap Algoritma

1. **Genetic Algorithm** : tidak mencapai nilai optimal global pada semua *test case* dan nilai *objective function* terakhir berkisar antara 6883 hingga 7617. Genetic algorithm tetap menghasilkan solusi suboptimal karena meskipun memiliki elemen crossover dan mutasi yang membantu eksplorasi, namun proses pemasangan yang acak masih rentan terhadap local optima, apalagi jika parameter mutasi dan crossover belum optimal.
2. **Steepest Ascent Hill-Climbing with Sideways Move** : menunjukkan algoritma ini cenderung terjebak dalam local optima, dengan nilai *objective function* yang tidak mendekati hasil optimal global. Hal ini terjadi karena algoritma ini hanya memperbaiki solusi terbaik dalam ruang lingkup lokal dan kurangnya mekanisme eksplorasi yang luas, membuatnya sulit keluar dari local optima.
3. **Simulated Annealing** : mendekati *global optima* lebih baik daripada *Hill-Climbing*, dengan hasil yang menunjukkan pengurangan cost awal yang konsisten pada ketiga test case. Kemampuan *Simulated Annealing* untuk menghindari local optima meningkat berkat penerapan "cooling rate" yang perlahan mengurangi probabilitas menerima solusi suboptimal, memungkinkan algoritma ini lebih mudah keluar dari jebakan local optima.

B. Perbandingan Setiap Algoritma

1. **Genetic Algorithm** : menghasilkan objective function dengan nilai yang lebih tinggi dari Simulated Annealing dan Hill-Climbing, menunjukkan performa yang kurang baik dalam mencapai global optima dibandingkan algoritma-algoritma tersebut.
2. **Steepest Ascent Hill-Climbing with Sideways Move** : menunjukkan hasil yang tidak konsisten dan lebih buruk daripada Simulated Annealing, tetapi lebih cepat

dibandingkan Genetic Algorithm. Dengan sideways move, Hill-Climbing memiliki kapasitas yang terbatas untuk keluar dari local optima, namun hasilnya tetap jauh dari optimal.

3. **Simulated Annealing** : memiliki performa terbaik dari ketiga algoritma dalam mencapai hasil yang lebih dekat ke global optima pada setiap test case. Proses annealing membantu menemukan solusi yang lebih baik daripada Hill-Climbing dengan sideways move dan Genetic Algorithm.

C. Durasi Pencarian Setiap Algoritma

1. **Genetic Algorithm** : memiliki durasi pencarian yang lebih cepat dibandingkan Simulated Annealing, sekitar 8453-8687 ms pada tiap test case. Namun, Genetic Algorithm masih memerlukan durasi lebih lama dibandingkan Steepest Ascent Hill-Climbing karena proses evolusi yang rumit.
2. **Steepest Ascent Hill-Climbing with Sideways Move** : merupakan algoritma yang paling cepat dengan durasi sekitar 6113-6278 ms. Ini disebabkan oleh sifat algoritma yang iteratif langsung tanpa eksplorasi solusi acak yang luas.
3. **Simulated Annealing** : memiliki durasi paling lambat, dengan waktu sekitar 135-156 detik. Lamanya proses disebabkan oleh banyaknya iterasi yang diperlukan untuk mengeksplorasi ruang solusi dan cooling rate yang secara bertahap menurunkan probabilitas penerimaan solusi kurang baik.

D. Kekonsistennan Hasil Akhir

1. **Genetic Algorithm** : memiliki hasil akhir yang cukup konsisten antara ketiga test case dengan objective function yang relatif tidak berbeda jauh (6883 hingga 7617). Variasi kecil ini menunjukkan konsistensi dalam kinerja, meskipun tidak mencapai nilai optimal.
2. **Steepest Ascent Hill-Climbing with Sideways Move** : memiliki hasil akhir yang tidak konsisten. Nilai objective function berkisar antara 438 hingga 1114, menunjukkan ketergantungan pada kondisi awal dan parameter algoritma, sehingga hasilnya dapat sangat berbeda setiap kali dijalankan.

3. **Simulated Annealing** : menunjukkan hasil yang paling konsisten di antara ketiga algoritma, dengan cost akhir yang berkisar antara 431 hingga 581. Variasi kecil ini menunjukkan bahwa Simulated Annealing mampu menurunkan cost secara stabil pada tiap percobaan.

E. Iterasi dan Jumlah Populasi Akhir

1. **Genetic Algorithm** : jumlah iterasi berpengaruh langsung pada kemampuan algoritma untuk mendekati solusi optimal. Dengan 474 hingga 479 iterasi dalam ketiga test case, algoritma mendekati konvergensi, tetapi masih di local optima. Jumlah populasi (500) juga mempengaruhi diversifikasi dalam pencarian solusi, memungkinkan lebih banyak kombinasi solusi untuk dieksplorasi. Namun, peningkatan lebih lanjut pada parameter iterasi dan populasi mungkin diperlukan untuk mencapai hasil yang lebih dekat ke global optima.
2. **Steepest Ascent Hill-Climbing with Sideways Move** : banyaknya iterasi membantu mengeksplorasi ruang solusi lokal, namun efeknya terbatas karena algoritma ini tidak memiliki mekanisme eksplorasi yang luas. Sideways move memungkinkan algoritma menghindari jebakan local optima untuk beberapa waktu, tetapi jika tidak ada perbaikan lebih lanjut dalam solusi, algoritma tetap akan terhenti di titik yang mendekati local optima. Dengan demikian, peningkatan iterasi sering kali hanya memperpanjang waktu tanpa memberikan hasil yang lebih optimal.
3. **Simulated Annealing** : jumlah iterasi dan cooling rate memiliki pengaruh signifikan terhadap kemampuan algoritma dalam mencapai solusi yang lebih optimal. Pada Simulated Annealing, semakin banyak iterasi yang dijalankan dengan cooling rate yang tepat, semakin besar kemampuannya untuk keluar dari jebakan local optima dan mendekati global optima. Namun, ini juga meningkatkan waktu pencarian secara substansial. Penurunan suhu yang terlalu cepat dapat membuat algoritma berhenti di local optima, sedangkan cooling rate yang lambat membantu eksplorasi lebih luas meskipun dengan durasi lebih panjang.

BAB IV

Spesifikasi Bonus

A. Bonus 1 : Implementasi Seluruh Algoritma Hill Climbing

Berikut merupakan tabel perbandingan setiap algoritma *hill climbing* :

Algoritma	Iterations	Final Cost (lower is better)
Hill Climbing with Sideways Move	85	722
Random Restart Hill Climbing	1210	262
Stochastic Hill Climbing	391	776

Tabel 3.1.1. Perbandingan algoritma *Hill Climbing*

Berdasarkan tabel di atas dapat disimpulkan bahwa dari semua percobaan yang telah kami lakukan, algoritma ***Hill Climbing with Sideways Move*** merupakan algoritma yang memiliki performa paling baik berdasarkan **banyak iterasi** dan **cost** akhirnya. Berikut merupakan seluruh code pembuktian dari setiap algoritma *hill climbing* :

Algoritma	Link Hasil Code Pembuktian
Hill Climbing with Sideways Move	SidewaysMove.cpp
Random Restart Hill Climbing	RandomRestart.cpp
Stochastic Hill Climbing	StochasticHill.cpp

Tabel 3.2.1. Code pembuktian algoritma *Hill Climbing*

B. Bonus 2 : Video player

Berikut merupakan tautan screen record hasil video player yang sudah mencakup hal berikut :

- a. Play/pause
- b. Progress bar (bisa *drag and drop*)
- c. Playback speed
- d. Load file hasil eksperimen (File eksperimen ini menyimpan tiap state/kondisi per iterasi dari proses pencarian)

Berikut adalah file unity untuk mem-visualisasikan pergerakan seluruh algoritma yang dibuat.

BAB IV

Kesimpulan dan Saran

A. Kesimpulan

Kesimpulannya, **Simulated Annealing** memiliki performa terbaik dalam mendekati global optima untuk masalah optimasi pada magic cube, meskipun membutuhkan durasi yang lebih lama. Algoritma ini efektif menghindari jebakan local optima, memberikan hasil yang konsisten di semua test case. **Genetic Algorithm** juga menunjukkan hasil yang stabil dan seimbang antara kualitas solusi dan waktu pencarian, namun tetap rentan terhadap local optima dan mungkin memerlukan penyesuaian parameter lebih lanjut. Sementara itu, **Steepest Ascent Hill-Climbing with Sideways Move** paling cepat dalam waktu pencarian, tetapi hasilnya kurang optimal dan tidak konsisten untuk masalah yang kompleks.

B. Saran

Saran yang dapat diberikan adalah melakukan pengujian lebih lanjut terhadap kombinasi dari algoritma-algoritma yang ada dan mempertimbangkan optimasi parameter yang dapat meningkatkan efisiensi dan akurasi dalam penyelesaian *Magic Cube*. Dibutuhkan juga pemahaman mendalam mengenai *magic number* dan variasi ukuran kubus yang lebih besar sehingga dapat memberikan wawasan tambahan untuk pengembangan solusi yang lebih inovatif di masa depan.

BAB VI

Pembagian tugas

Nama	NIM	Tugas
Jihan Aurelia	18222001	Genetic Algorithm
Nasywaa Anggun Athiefah	18222021	Genetic Algorithm
Ricky Wijaya	18222043	Simulated Annealing
Muhammad Adli Arindra	18222089	<ul style="list-style-type: none">• Steepest Ascent Hill Climbing with Sideways Move• Membuat bonus• Membuat Unity

BAB V

Referensi

- Brownlee, J. (2021, October 12). *Stochastic Hill Climbing in Python from Scratch - MachineLearningMastery.com*. Machine Learning Mastery. Retrieved October 2, 2024, from <https://machinelearningmastery.com/stochastic-hill-climbing-in-python-from-scratch/>
- Do, While and For loops in Pseudocode - PseudoEditor. (n.d.). Pseudocode Online Editor. Retrieved September 29, 2024, from <https://pseudoeditor.com/guides/loops-and-iteration>
- Jain, S. (2023, April 20). *Introduction to Hill Climbing | Artificial Intelligence*. GeeksforGeeks. Retrieved October 2, 2024, from <https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/>
- Jain, S. (2024, August 22). Local Search Algorithm in Artificial Intelligence. GeeksforGeeks. Retrieved September 29, 2024, from <https://www.geeksforgeeks.org/local-search-algorithm-in-artificial-intelligence/>
- Kothari, S. (2023, September 25). Local Search Algorithms in AI: A Comprehensive Guide. Simplilearn.com. Retrieved September 29, 2024, from <https://www.simplilearn.com/local-search-algorithms-in-ai-article>
- Magic cube. (n.d.). Wikipedia. Retrieved September 29, 2024, from https://en.wikipedia.org/wiki/Magic_cube
- Putria, N. E. (2022). Comparison of Simple Hill Climbing Algorithm and Stepest Ascent Hill Climbing Algorithm in The Game Order of Numbers. *International Journal of Information System & Technology*, 6(1), 33-40.
- Samosir, S. A. (2019). IMPLEMENTASI METODE STEEPEST ASCENT HILL CLIMBING DALAM PENCARIAN RUTE TERDEKAT PROMOSI KAMPUS STMIK BUDI DARMA. *Jurnal Pelita Informatika*, 8(2), 283-287. Silvilestari. (2021). Sideways Move Hill Climbing Algorithm To Solve Cases In Puzzle Game 8. *International Journal of Information System & Technology*, 5(4), 366-370.
- Stochastic Hill Climbing With Random Restarts*. (n.d.). Algorithm Afternoon. Retrieved October 2, 2024, from https://algorithmafternoon.com/stochastic/stochastic_hill_climbing_with_random_restarts/
- Slide Kuliah IF3070 - Foundations of Artificial Intelligence.