TUGAS BESAR 1

IF3070 Dasar Inteligensi Artifisial

Pencarian Solusi Diagonal Magic Cube dengan Local Search

Disusun Oleh Kelompok 28:

18222008	Abel Apriliani
18222036	Olivia Christy Lismanto
18222044	Khansa Adilla Reva
18221062	Nafisha Virgin

Dosen Pengampu :

Dr.Nur Ulfa Maulidevi, ST,M.Sc.

Sekolah Teknik Elektro dan Informatika

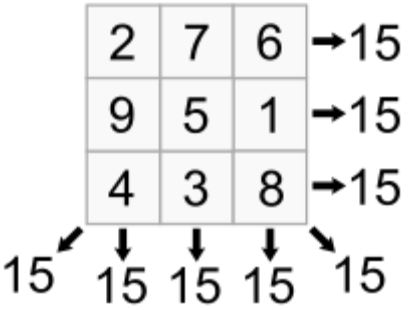
Institut Teknologi Bandung

2024

DAFTAR ISI

BAB 1
DESKRIPSI PERSOALAN

Magic Square dalam matematika rekreasi dan desain kombinatorial adalah $n \times n$ kotak persegi (n adalah jumlah kotak di setiap sisi) yang diisi dengan bilangan asli positif mulai dari $1, 2, \ldots, n^2$, sampai semua kotak terisi dengan bilangan asli positif yang berbeda, kemudian bilangan asli positif di setiap baris, kolom dan diagonal jika dijumlahkan menghasilkan angka yang sama (15). Jumlahnya disebut *magic constant* atau *magic sum* dari *magic square*.

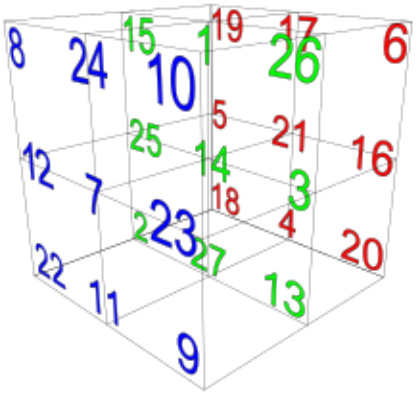


Gambar 1. Model persegi ajaib yang terkecil, terdiri dari 3 baris

Dalam matematika, *magic cube* adalah versi 3 dimensi dari *magic square*, yaitu kumpulan bilangan bulat yang diatur dalam pola $n \times n \times n$ sedemikian rupa sehingga jumlah angka pada setiap baris, setiap kolom, setiap pilar, dan masing-masing dari empat diagonal ruang utama adalah sama, yang disebut *magic constant* dari kubus, dilambangkan dengan $M_3(n)$. Jika *magic cube* terdiri dari angka-angka 1, 2, ..., n^3. Dengan demikian, maka kubus tersebut memiliki *magic constant*:

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

Selain itu, jika jumlah angka pada setiap diagonal penampang juga sama dengan *magic number* kubus, maka kubus tersebut disebut *perfect magic cube*; jika tidak, maka disebut *semiperfect magic cube*. Angka n disebut ordo dari *magic cube*. Jika jumlah angka pada diagonal ruang yang terputus dari *magic cube* juga sama dengan angka *magic cube* maka kubus tersebut disebut *pandiagonal magic cube.*



Gambar 2. Simple Magic Cube

Pada tugas ini, terdapat *diagonal magic cube* yang merupakan kubus yang tersusun dari angka 1 hingga 5^3 secara acak tanpa pengulangan dengan 5 merupakan panjang sisi kubus magic cube tersebut. Kemudian, tiap iterasi pada algoritma *local search*, langkah yang diperbolehkan adalah menukar posisi dari 2 angka pada kubus tersebut. Angka-angka yang tersusun harus memenuhi ketentuan sebagai berikut.

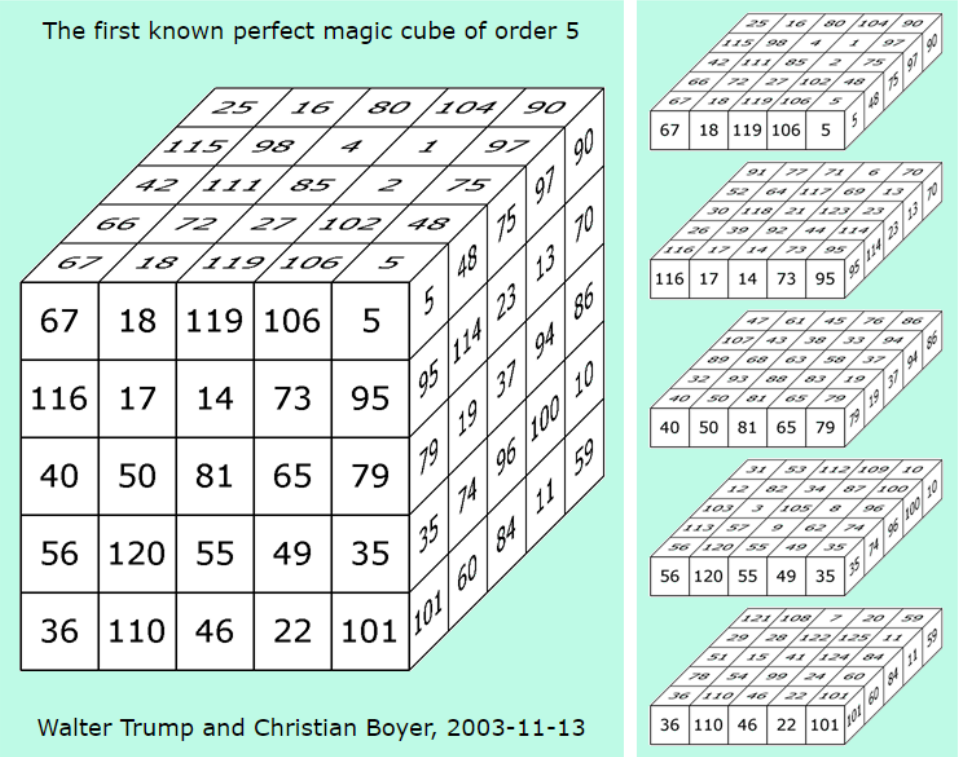
- Terdapat satu angka yang merupakan *magic number* dari kubus tersebut yang tidak masuk rentang angka 1 hingga 5^3, magic number juga bukan angka yang masuk kedalam kubus tersebut.
- Jumlah angka di setiap baris sama dengan *magic number*
- Jumlah angka di setiap kolom sama dengan *magic number*
- Jumlah angka di setiap tiang (pillars) sama dengan *magic number*
- Jumlah angka di setiap diagonal untuk semua diagonal dalam kubus sama dengan *magic number*
- Jumlah angka di setiap diagonal ruang untuk semua diagonal ruang dalam kubus tersebut sama dengan *magic number*

Berdasarkan persoalan diatas, diharuskan melakukan pencarian *local search* terbaik untuk menyelesaikan permasalahan ini dengan melakukan perbandingan dengan antara setiap algoritma *local search.*

BAB 2
PEMBAHASAN

2.1. Objective Function

h = – jumlah baris, kolom, tiang, diagonal, dan diagonal ruang (*straight lines of magic cube*) yang memiliki nilai tidak sama dengan 315



Gambar 3. Perfect Magic Cube

Kubus 5 x 5 terdiri dari angka 1 sampai 125. Untuk mencapai *perfect magic cube,* maka setiap baris, kolom, pilar, diagonal, dan diagonal ruang harus bernilai 315. Angka 315 ini dapat dicari dengan cara mencari total nilai dari 1 sampai 125. Lalu, didapatkan bahwa totalnya adalah 7875. Angka tersebut dibagi dengan 25 (banyak baris/kolom/pilar). Didapatlah 315. Oleh karena itu, untuk menilai *current value* dari *state* dapat menggunakan nilai 315 sebagai patokan.

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

Magic number bagi suatu *magic cub*e adalah konstan (magic constant). Maka menurut rumus diatas, bagi kubus berukuran 5 × 5 × 5, *magic number* atau *current valu*e dari *state* dari kubus ini adalah 315.

Terdapat dua jenis *neighbor* dalam algoritma local search, yaitu :

- ***highest-valued successor***

Perubahan *state* dilakukan dengan cara menukar 2 angka pada kubus. Karena terdapat 125 angka, maka banyak kemungkinan penukaran dapat dihitung dengan menggunakan rumus kombinasi.

$$C(n,r) = \frac{n!}{r!(n-r)!}$$

$$C\ (125,\ 2) = \frac{125!}{2!(125-2)!} = \frac{125 \times 124}{2} = 7750$$

Setiap *state* memiliki 7750 *successor. Neighbor* adalah *successor* dengan nilai tertinggi. Apabila nilai tertingginya berada di lebih dari satu tempat, maka akan dipilih secara *random* di antara nilai yang sama tersebut.

- ***random successor***

Successor function mengembalikan *random state* dengan menukar dua angka pada kubus secara *random.*

2.2. Implementasi Algoritma Local Search

Untuk mengimplementasikan local search dalam menyelesaikan persoalan Diagonal Magic Cube, kami menggunakan bahasa pemrograman C++.

2.2.1. Local Search

Berikut adalah penjelasan implementasi dari algoritma LocalSearch.

Nama File	
LocalSearch	Merepresentasikan sebuah *Magic Cube* dan digunakan untuk menghitung objective functionnya.
Variable	
value	Merepresentasikan jumlah garis (baris, kolom, tiang, diagonal) yang tidak memenuhi *magic constant* 315
state	Representasi Matriks 3 dimensi dalam sebuah array satu dimensi
CUBE	Struktur data yang berisi atribut state (array tiga dimensi) dan value
Fungsi dan Prosedur	
findValue	Menghitung jumlah garis yang tidak memenuhi nilai 315

highestSuccessor	Mencari neighbor dengan nilai value terbaik
randomSuccessor	Menghasilkan neighbor acak dengan menukar dua elemen
generateInitialState	Mengisi magic cube dengan nilai acak untuk initial state
printState	Menampilkan susunan elemen magic cube

Berikut merupakan potongan source code untuk file ini.

```
#include <bits/stdc++.h>
#include "LocalSearch.hpp"
using namespace std;


int findValue(CUBE c, int maxVal)
{
	int cnt=maxVal;
	for(int i=1; i<=5; i++)
	{
	for(int j=1; j<=5; j++)
	{
		if(c.state[1][i][j]+c.state[2][i][j]+c.state[3][i][j]+
		c.state[4][i][j]+c.state[5][i][j]!=315) //baris
		{
		cnt--;
		}
		if(c.state[i][1][j]+c.state[i][2][j]+c.state[i][3][j]+
		c.state[i][4][j]+c.state[i][5][j]!=315) //kolom
		{
		cnt--;
		}
		if(c.state[i][j][1]+c.state[i][j][2]+c.state[i][j][3]+
		c.state[i][j][4]+c.state[i][j][4]!=315) //tiang
		{
		cnt--;
		}
	}
	}
	for(int i=1; i<=5; i++)
	{
	if(c.state[1][1][i]+c.state[2][2][i]+c.state[3][3][i]+
	c.state[4][4][i]+c.state[5][5][i]!=315) cnt--;
	if(c.state[1][5][i]+c.state[2][4][i]+c.state[3][3][i]+
	c.state[4][2][i]+c.state[5][1][i]!=315) cnt--;

	if(c.state[1][i][1]+c.state[2][i][2]+c.state[3][i][3]+
	c.state[4][i][4]+c.state[5][i][5]!=315) cnt--;
	if(c.state[1][i][5]+c.state[2][i][4]+c.state[3][i][3]+
	c.state[4][i][2]+c.state[5][i][1]!=315) cnt--;

	if(c.state[i][1][1]+c.state[i][2][2]+c.state[i][3][3]+
	c.state[i][4][4]+c.state[i][5][5]!=315) cnt--;
	if(c.state[i][1][5]+c.state[i][2][4]+c.state[i][3][3]+
```

```
	c.state[i][4][2]+c.state[i][5][1]!=315) cnt--;
	}

if(c.state[1][1][1]+c.state[2][2][2]+c.state[3][3][3]+c.state[4][4][
4]+c.state[5][5][5]) cnt--;

if(c.state[5][1][1]+c.state[4][2][2]+c.state[3][3][3]+c.state[2][4][
4]+c.state[1][5][5]) cnt--;

if(c.state[1][5][1]+c.state[2][4][2]+c.state[3][3][3]+c.state[4][2][
4]+c.state[5][1][5]) cnt--;

if(c.state[5][5][1]+c.state[4][4][2]+c.state[3][3][3]+c.state[2][2][
4]+c.state[1][1][5]) cnt--;
	return cnt;
}

CUBE highestSuccessor(CUBE c)
{
	CUBE h;
	h.value=-316;
	int i_ans, j_ans, k_ans, l_ans, m_ans, n_ans;
	for(int i=1; i<=5; i++)
	{
	for(int j=1; j<=5; j++)
	{
		for(int k=1; k<=5; k++)
		{
		for(int l=1; l<=5; l++)
		{
			for(int m=1; m<=5; m++)
			{
				for (int n=1; n<=5; n++)
				{
				swap(c.state[i][j][k],c.state[l][m][n]);
				if(findValue(c,0)>h.value)
				{
					h.value=findValue(c,0);
					i_ans=i; j_ans=j; k_ans=k;
					l_ans=l; m_ans=m; n_ans=n;
				}
				swap(c.state[i][j][k],c.state[l][m][n]);
				}
			}
		}
		}
	}
	}

swap(c.state[i_ans][j_ans][k_ans],c.state[l_ans][m_ans][n_ans]);
	return c;
}

CUBE randomSuccessor(CUBE c)
{
	int i=((rand()%5)+1);
	int j=((rand()%5)+1);
	int k=((rand()%5)+1);
```

```
	int l=((rand()%5)+1);
	int m=((rand()%5)+1);
	int n=((rand()%5)+1);
	while(i==l && j==m && k==n)
	{
	k=((rand()%5)+1);
	}
	swap(c.state[i][j][k],c.state[l][m][n]);
	c.value=findValue(c,0);
	return c;
}

void generateInitialState(CUBE *c)
{
	int new_random;
	bool unique;
	bool check[126];
	memset(check, 0, sizeof(check));
	for (int i=1; i<=5; i++)
	{
	for (int j=1; j<=5; j++)
	{
		for (int k=1; k<=5; k++)
		{
		do
		{
			new_random=((rand()%125)+1);
			unique=true;
			if(check[new_random]) unique=false;
		}
		while(!unique);
		check[new_random]=true;
		(*c).state[i][j][k]=new_random;
		}
	}
	}
}

void printState(CUBE c)
{
	for(int i=1; i<=5; i++)
	{
	cout<<"Layer "<<i<<endl;
	for(int j=1; j<=5; j++)
	{
		for (int k=1; k<=5; k++)
		{
		cout<<c.state[i][j][k]<<" ";
		}
		cout<<endl;
	}
	cout<<endl;
	}
}
```

2.2.2. Hill-Climbing Steepest Ascent

Hill-Climbing Steepest Ascent adalah salah satu jenis algoritma pencarian lokal (*local search*) yang digunakan untuk menemukan solusi terbaik (*global optima*) atau solusi yang mendekati optimal (*local optima*) dalam suatu masalah yang memiliki mekanisme utama untuk memilih *neighbour* yang paling meningkatkan nilai fungsi objektif dari seluruh *neighbour* yang mungkin dan memutuskan apakah *state* dari *neighbour* tersebut akan dipilih sebagai *current state* atau menghentikan proses pencarian karena tidak ada *state neighbour* yang lebih baik dari *current state*.

Berikut adalah implementasi dari algoritma Hill-Climbing Steepest Ascent.

Nama File	
HillClimbingSteepestAscent	Class ini merupakan hasil implementasi dari Local Search Hill-Climbing Steepest Ascent
Variabel	
values	Merepresentasikan vektor bertipe integer untuk menyimpan *objective function* setiap iterasi
duration	Menyimpan durasi waktu yang dibutuhkan untuk menjalankan algoritma.
initial	Menyimpan *state* atau kondisi awal dari solusi yang digunakan untuk memulai algoritma.
current	Menyimpan *state* dari solusi saat ini
currentValue	Menyimpan nilai *objective function* dari variabel *current* yang menyimpan *current state* .
values	Merepresentasikan vektor bertipe integer untuk menyimpan *objective function* setiap iterasi
Fungsi dan Prosedur	
HillClimbingSteepestAscent(CUBE initial)	Melakukan local search dengan algoritma Hill-Climbing Steepest Ascent.
highestSuccessor(CUBE c)	Mendapatkan sebuah neighbor yang memiliki nilai objective function tertinggi di antara kemungkinan semua neighbour
generateInitialState(CUBE *c)	Mengisi magic cube dengan nilai acak untuk initial state
findValue(CUBE c, int maxVal)	Menghitung dan mengembalikan nilai objective

	function dari state
printState(CUBE c)	Menampilkan representasi visual dari suatu state

Berikut merupakan potongan Source Code untuk file tersebut.

```
#include <bits/stdc++.h>
#include "LocalSearch.hpp"

using namespace std;

void HillClimbingSteepestAscent(CUBE initial)
{
    CUBE current = initial;
    int currentValue = findValue(current,0);
    vector<int> values;
    int iterations = 0;

    ofstream outfile("objective_values.txt");

    while (true) // Limit iterasi maksimum
    {
        CUBE next = highestSuccessor(current);
        int nextValue = findValue(next,0);

        if (nextValue > currentValue)
        {
            current = next;
            currentValue = nextValue;
        }
        else{
            break;
        }

        values.push_back(currentValue);
        outfile << currentValue << endl;
        iterations++;

        if (currentValue == 0) // Tujuan tercapai
            break;
    }

    outfile.close();

    // Tampilkan hasil

    // nilai objective function per iterasi
    cout << "\nNilai Objective Function per Iterasi:" << endl;
    for (int i = 0; i < (values.size()); i++) //values.size() - 1
itu supaya tidak memasukkan yang state awal
    {
        cout << "Iterasi " << i + 1 << ": " << values[i] << endl;
    }

    cout << "\nGrafik Plot objective function terhadap banyak
iterasi dapat dilihat dengan menjalankan file objectiveValue.py" <<
endl;
```

```
    cout << "\nState Awal:" << endl;
    printState(initial);
    cout << "Nilai Objective Function Awal: " <<
findValue(initial,0) << endl;

    cout << "\nState Akhir:" << endl;
    printState(current);
    cout << "Nilai Objective Function Akhir: " << currentValue <<
endl;

    cout << "\nTotal Iterasi: " << iterations << endl;
}

int main()
{
    srand(time(0));
    auto start = chrono::high_resolution_clock::now();
    CUBE cube;
    generateInitialState(&cube);
    HillClimbingSteepestAscent(cube);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    cout << "\nDurasi: " << duration.count() << " detik\n" << endl;
    return 0;
}
```

2.2.3. Hill-Climbing with Sideways Move

Hill-Climbing with Sideways Move adalah salah satu jenis algoritma pencarian lokal (*local search*) yang digunakan untuk menemukan solusi terbaik (*global optima*) atau solusi yang mendekati optimal (*local optima*) dalam suatu masalah yang mengatasi masalah plateau dalam ruang pencarian. Plateau adalah area di mana semua *neighbour* dari *current state* memiliki nilai *objective function* yang sama. Mekanisme utamanya adalah memilih *neighbour* yang paling meningkatkan nilai *objective function* dari seluruh *neighbour* yang mungkin atau memiliki nilai *objective function* yang sama dengan *current state* (kondisi ini hanya dapat dijalankan dalam batas *sideways move* tertentu) untuk membantu algoritma keluar dari plateau dan memberikan kesempatan lebih besar untuk menemukan solusi yang lebih baik. Algoritma ini nanti akan memutuskan apakah *state* dari *neighbour* tersebut akan dipilih sebagai *current state* atau menghentikan proses pencarian karena tidak adanya *state neighbour* yang lebih baik dari *current state* ataupun habisnya batas *sideways move* yang telah ditetapkan sebelumnya.

Berikut adalah implementasi dari algoritma Hill-Climbing with Sideways Move.

Nama File	
HillClimbingSidewaysMove	Class ini merupakan hasil implementasi dari Local Search Hill-Climbing Sideways Move
Variabel	

values	Merepresentasikan vektor bertipe integer untuk menyimpan *objective function* setiap iterasi
duration	Menyimpan durasi waktu yang dibutuhkan untuk menjalankan algoritma.
maxsideways	Menyimpan batas nilai sideways move
initial	Menyimpan *state* atau kondisi awal dari solusi yang digunakan untuk memulai algoritma.
current	Menyimpan *state* dari solusi saat ini
currentValue	Menyimpan nilai *objective function* dari variabel *current* yang menyimpan *current state* .
values	Merepresentasikan vektor bertipe integer untuk menyimpan *objective function* setiap iterasi
Fungsi dan Prosedur	
HillClimbingSidewaysMove(CUBE initial, int maxsideways)	Melakukan local search dengan algoritma Hill-Climbing Sideways Move.
highestSuccessor(CUBE c)	Mendapatkan sebuah neighbor yang memiliki nilai objective function tertinggi di antara kemungkinan semua neighbour
generateInitialState(CUBE *c)	Mengisi magic cube dengan nilai acak untuk initial state
findValue(CUBE c, int maxVal)	Menghitung dan mengembalikan nilai objective function dari state
printState(CUBE c)	Menampilkan representasi visual dari suatu state

Berikut merupakan potongan Source Code untuk file tersebut.

```
#include <bits/stdc++.h>
#include "LocalSearch.hpp"

using namespace std;

void HillClimbingSidewaysMove(CUBE initial, int maxsideways)
{

    CUBE current = initial;
    int currentValue = findValue(current,0);
    vector<int> values;

    int iterations = 0;
    int sideways = 0;
```

```
    ofstream outfile("objective_values.txt");

    while (true)
    {
        CUBE next = highestSuccessor(current);
        int nextValue = findValue(next,0);

        if (nextValue > currentValue)
        {
            // Jika nilai meningkat, terima keadaan baru
            current = next;
            currentValue = nextValue;
            sideways = 0; // Reset sideways move counter
        }
        else if (nextValue == currentValue && sideways <
maxsideways)
        {
            // Jika nilai sama, lakukan sideways move
            current = next;
            currentValue = nextValue;
            sideways++;
        }
        else
        {
            // Tidak ada peningkatan, hentikan pencarian
            break;
        }

        values.push_back(currentValue);
        outfile << currentValue << endl;
        iterations++;

        // Jika mencapai nilai objective function yang optimal
        if (currentValue == 0)
            break;
    }

    outfile.close();

    // Tampilkan hasil
    cout << "\nNilai Objective Function per Iterasi:" << endl;
    for (int i = 0; i < values.size(); i++)
    {
        cout << "Iterasi " << i + 1 << ": " << values[i] << endl;
    }

    cout << "\nGrafik Plot objective function terhadap banyak
iterasi dapat dilihat dengan menjalankan file objectiveValue.py" <<
endl;
    cout << "\nState Awal:" << endl;
    printState(initial);
    cout << "Nilai Objective Function Awal: " <<
findValue(initial,0) << endl;

    cout << "\nState Akhir:" << endl;
    printState(current);
```

```
    cout << "Nilai Objective Function Akhir: " << currentValue <<
endl;
    cout << "\nTotal Iterasi: " << iterations << endl;
}

int main()
{
    int maxsideways; // Menyimpan jumlah iterasi sideways maksimum

    // Meminta input jumlah iterasi sideways maksimum dari pengguna
    cout << "\nMasukkan jumlah sideways move maksimum: ";
    cin >> maxsideways;
    srand(time(0));
    auto start = chrono::high_resolution_clock::now();
    CUBE cube;
    generateInitialState(&cube);
    HillClimbingSidewaysMove(cube, maxsideways);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    cout << "\nDurasi: " << duration.count() << " detik\n" << endl;
    return 0;
}
```

2.2.4. Random Restart Hill-Climbing

Random Restart Hill-Climbing adalah salah satu jenis algoritma pencarian lokal (*local search*) yang digunakan untuk menemukan solusi terbaik (*global optima*) atau solusi yang mendekati optimal (*local optima*) dalam suatu masalah yang memiliki mekanisme utama untuk memilih memilih *neighbour* yang paling meningkatkan nilai *objective function* dari seluruh *neighbour* yang mungkin dan jika algoritma menemui kebuntuan maka dapat melakukan beberapa kali restart dari titik awal yang dipilih secara acak setiap kali menemui kebuntuan selama belum melewati batas jumlah restart yang ada.

Berikut adalah implementasi dari algoritma Random restart Hill-Climbing.

Nama File	
RandomRestartHillClimbing	Class ini merupakan hasil implementasi dari Local Search Random Restart Hill-Climbing
Variabel	
values	Merepresentasikan vektor bertipe integer untuk menyimpan *objective function* setiap iterasi
duration	Menyimpan durasi waktu yang dibutuhkan untuk menjalankan algoritma.
jmlIterations	Merepresentasikan vektor bertipe integer untuk menyimpan jumlah iterasi per restart

restart	Total restart yang dilakukan selama menjalankan algoritma
maxRestart	Batas jumlah restart maksimum
initial	Menyimpan *state* atau kondisi awal dari solusi yang digunakan untuk memulai algoritma.
current	Menyimpan *state* dari solusi saat ini
currentValue	Menyimpan nilai *objective function* dari variabel *current* yang menyimpan *current state* .
values	Merepresentasikan vektor bertipe integer untuk menyimpan *objective function* setiap iterasi
Fungsi dan Prosedur	
RandomRestartHillClimbing(CUBE initial, int maxRestart)	Melakukan local search dengan algoritma Random Restart Hill-Climbing.
highestSuccessor(CUBE c)	Mendapatkan sebuah neighbor yang memiliki nilai objective function tertinggi di antara kemungkinan semua neighbour
generateInitialState(CUBE *c)	Mengisi magic cube dengan nilai acak untuk initial state
findValue(CUBE c, int maxVal)	Menghitung dan mengembalikan nilai objective function dari state
printState(CUBE c)	Menampilkan representasi visual dari suatu state

Berikut merupakan potongan Source Code untuk algoritma tersebut.

```
#include <bits/stdc++.h>
#include "LocalSearch.hpp"

using namespace std;

void RandomRestartHillClimbing(CUBE initial, int maxRestart)
{

    CUBE current = initial;
    int currentValue = findValue(current, 0);

    vector<int> values;
    vector<int> jmlIterations;

    int restart = 0; //banyak restart
    int iterationsperrestart = 0; //iterasi per restart
```

```
    ofstream outfile("objective_values.txt");

    while (restart < maxRestart)
    {
        CUBE next = highestSuccessor(current);
        int nextValue = findValue(next, 0);

        if (nextValue > currentValue)
        {
            current = next;
            currentValue = nextValue;
            iterationsperrestart++;
        }
        else
        {
            // Simpan data iterasi dan restart
            jmlIterations.push_back(iterationsperrestart);
            restart++;

            // Restart dengan state baru
            if (restart != maxRestart)
            {
                generateInitialState(&current);
                currentValue = findValue(current, 0);
                iterationsperrestart = 0;
            }
        }

        values.push_back(currentValue);
        outfile << currentValue << endl;

        if (currentValue == 0) // Tujuan tercapai
            break;
    }

    outfile.close();
    // Tampilkan hasil
    cout << "\nNilai Objective Function per Iterasi:" << endl;
    for (size_t i = 0; i < values.size(); i++)
    {
        cout << "Iterasi " << i+1 << ": " << values[i] << endl;
    }

    cout << "\nGrafik Plot objective function terhadap banyak
iterasi dapat dilihat dengan menjalankan file objectiveValue.py" <<
endl;

    // Tampilkan jumlah restart dan iterasi per restart
    cout << "\nJumlah restart yang dilakukan: " << restart << endl;
    for (size_t i = 0; i < jmlIterations.size(); i++)
    {
        cout << "Jumlah iterasi pada restart ke-" << i + 1 << ": "
<< jmlIterations[i] << endl;
    }

    // State awal dan akhir
    cout << "\nState Awal:" << endl;
    printState(initial);
```

```
    cout << "Nilai Objective Function Awal: " << findValue(initial,
0) << endl;

    cout << "\nState Akhir:" << endl;
    printState(current);
    cout << "Nilai Objective Function Akhir: " << currentValue <<
endl;

    cout << "\nTotal restart: " << restart << endl;
    cout << "\nTotal Iterasi: " << values.size() << endl;
}

int main()
{
    int maxRestart; // Menyimpan jumlah restart maksimal

    // Meminta input jumlah restart dari pengguna
    cout << "\nMasukkan jumlah restart maksimum: ";
    cin >> maxRestart;
    srand(static_cast<unsigned>(time(0)));
    auto start = chrono::high_resolution_clock::now();
    CUBE cube;
    generateInitialState(&cube);
    RandomRestartHillClimbing(cube, maxRestart);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    cout << "\nDurasi: " << duration.count() << " detik\n" << endl;
    return 0;
}
```

2.2.5. Stochastic Hill-Climbing

Stochastic Hill Climbing adalah salah satu jenis algoritma pencarian lokal (*local search*) yang digunakan untuk menemukan solusi terbaik (*global optima*) atau solusi yang mendekati optimal (*local optima*) dalam suatu masalah yang memiliki mekanisme utama untuk memilih satu *neighbour* secara acak dan memutuskan apakah *state* dari *neighbour* tersebut akan dipilih sebagai *current state* atau melakukan iterasi lagi untuk memeriksa *state neighbour* lain.

Berikut adalah implementasi dari algoritma Stochastic Hill-Climbing.

Nama File	
StochasticHillClimbing	Sebagai implementasi dari Local Search Stochastic Hill Climbing
Variabel	
values	Merepresentasikan vektor bertipe integer untuk menyimpan objective function setiap iterasi

duration	Menyimpan durasi waktu yang dibutuhkan untuk menjalankan algoritma.
iterations	Menyimpan jumlah iterasi yang telah dijalankan oleh algoritma.
initial	Menyimpan *state* atau kondisi awal dari solusi yang digunakan untuk memulai algoritma.
current	Menyimpan *state* dari solusi saat ini
currentValue	Menyimpan nilai objective function dari variabel *current* yang menyimpan *current state* .
Fungsi dan Prosedur	
StochasticHillClimbing(CUBE initial, int maxIterations)	Melakukan local search dengan algoritma Stochastic Hill-Climbing
randomSuccessor(CUBE c)	Mendapatkan sebuah neighbor secara random
generateInitialState(CUBE *c)	Mengisi magic cube dengan nilai acak untuk initial state
findValue(CUBE c, int maxVal)	Menghitung dan mengembalikan nilai objective function dari state
printState(CUBE c)	Menampilkan representasi visual dari suatu state

Berikut merupakan potongan Source Code untuk algoritma tersebut.

```
#include <bits/stdc++.h>
#include "LocalSearch.hpp"

using namespace std;

void StochasticHillClimbing(CUBE initial, int maxIterations)
{

    CUBE current = initial;
    int currentValue = findValue(current,0);
    vector<int> values;
    int iterations = 0;

    ofstream outfile("objective_values.txt");

    while (iterations < maxIterations) // Limit iterasi maksimum
    {
        CUBE next = randomSuccessor(current);
        int nextValue = findValue(next,0);
```

```
        if (nextValue > currentValue)
        {
            current = next;
            currentValue = nextValue;
        }

        values.push_back(currentValue);
        outfile << currentValue << endl;
        iterations++;

        if (currentValue == 0) // Tujuan tercapai
            break;
    }

    outfile.close();
    // Tampilkan hasil
    cout << "\nNilai Objective Function per Iterasi:" << endl;
    for (int i = 0; i < values.size(); i++) //values.size() - 1 itu
supaya tidak memasukkan yang state awal
    {
        cout << "Iterasi " << i + 1 << ": " << values[i] << endl;
    }
    cout << "\nGrafik Plot objective function terhadap banyak
iterasi dapat dilihat dengan menjalankan file objectiveValue.py" <<
endl;
    cout << "\nState Awal:" << endl;
    printState(initial);
    cout << "Nilai Objective Function Awal: " <<
findValue(initial,0) << endl;

    cout << "\nState Akhir:" << endl;
    printState(current);
    cout << "Nilai Objective Function Akhir: " << currentValue <<
endl;

    cout << "\nTotal Iterasi: " << iterations << endl;
}

int main()
{
    int maxIterations; // Menyimpan jumlah iterasi maksimal

    // Meminta input jumlah iterasi dari pengguna
    cout << "\nMasukkan jumlah iterasi maksimum: ";
    cin >> maxIterations;
    srand(time(0));
    auto start = chrono::high_resolution_clock::now();
    CUBE cube;
    generateInitialState(&cube);
    StochasticHillClimbing(cube,maxIterations);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    cout << "\nDurasi: " << duration.count() << " detik\n" << endl;
    return 0;
}
```

2.2.6. Simulated Annealing

Simulated annealing mengkombinasikan antara *hill climbing* dan *random walk*. *Hill climbing* memiliki kelebihan berupa efisien, tetapi memiliki kekurangan berupa kemungkinan *stuck* di local maximum. *Random walk* memiliki kelebihan berupa kepastian mendapatkan *global maximum*, tetapi memiliki kekurangan berupa sangat tidak efisien. Karena simulated annealing mengkombinasikan keduanya, maka simulated annealing menjadi algoritma yang sangat optimal.

Ketika menggunakan algoritma *simulated annealing*, angka-angka pada kubus akan diinisiasi secara *random* terlebih dahulu. Dilakukan iterasi selama T>0. Angka yang ditukar dipilih secara *random.* Apabila *neighbor value* lebih besar dibanding *current value,* maka *current state* akan di-*update* menjadi *neighbor state.* Namun, apabila *neighbor value* tidak lebih besar dibanding *current value* dilakukan pengujian terhadap $e^{\Delta E/T}$. Apabila memenuhi, maka meskipun *neighbor value* tidak lebih besar dibanding *current value* akan tetap dilakukan penukaran angka pada kubus.

Kelonggaran pada penukaran ini, memungkinkan algoritma untuk tidak *stuck* di local maximum. Bahkan, apabila fungsi *schedule*(T) dibuat dengan tepat, probabilitas mendapatkan global maximum adalah 1, pasti akan didapatkan.

Berikut adalah implementasi dari algoritma Simulated Annealing

Nama File	
SimulatedAnnealing	Sebagai implementasi dari Simulated Annealing.
Variabel	
value	Merepresentasikan jumlah garis (baris, kolom, tiang, diagonal) yang tidak memenuhi *magic constant* 315
state	Representasi Matriks 3 dimensi dalam sebuah array satu dimensi
MAX_TIME	Batas iterasi maksimum
temperature	Menyimpan nilai temperatur awal
threshold	Batas probabilitas minimum untuk perubahan konfigurasi
stuck_count	Menghitung iterasi yang mengalami kondisi *stuck*
probability	Probabilitas untuk menerima neighbor yang lebih buruk, dihitung dengan rumus exp(ΔE/T)

values	Merepresentasikan vektor bertipe integer untuk menyimpan objective function setiap iterasi
probability_values	Menyimpan nilai probabilitas pada setiap iterasi
Fungsi dan Prosedur	
simulated_annealing()	Melakukan local search dengan algoritma Simulated Annealing
calculateTemperature	Mengurangi suhu secara bertahap untuk simulated annealing
randomSuccessor	Mendapatkan sebuah neighbor secara random
generateInitialState	Mengisi magic cube dengan nilai acak untuk initial state

Berikut merupakan potongan Source Code untuk algoritma Simulated Annealing.

```
#include <bits/stdc++.h>
#include "LocalSearch.hpp"

using namespace std;

double calculateTemperature(double Temperature, int time) {
	return Temperature - 0.000000001*time; //Nilai temperature
dikurang dengan 0.000000001*time agar turun secara perlahan
}

void simulatedAnnealing(CUBE &current) {
	double threshold = 0.55;
	int MAX_TIME = 50000;
	int time = 0;
	CUBE bestState = current;
	int bestValue = findValue(current,0);

	int stuck_count = 0;
	int unchanged_iterations = 0;
	const int stuck_threshold = 10;

	double temperature = 2;

	int currentValue = findValue(current, 0);

	ofstream outfile("objective_values.txt");
	ofstream probfile("probability_values.txt");

	while (time <= MAX_TIME-1) {
	temperature=calculateTemperature(temperature,time);
	if(temperature<=0) {cout << "Total stuck count (local
optima): " << stuck_count << endl; return;}
	CUBE neighbor = randomSuccessor(current);
	int neighborValue = findValue(neighbor,0);
```

```
        int deltaE = neighborValue - findValue(current,0);
        double probability;
        if(deltaE>0) probability=1;
        else probability = exp(double(deltaE) / temperature); //
e^(deltaE / T)

        probfile << probability << endl;

        cout << "Iteration: " << time+1
                << ", Temperature: " << temperature
                << ", DeltaE: " << deltaE
                << ", Probability (e^(DeltaE/T)): " << probability
                << ", Threshold: " << threshold
                << ", Value: " << current.value << endl;

        if (deltaE > 0) {
                current = neighbor;
        } else {
                if((probability > threshold && ((double)rand() /
RAND_MAX) < probability)) current=neighbor;
                currentValue = neighborValue;
                stuck_count++;
        }

        outfile << currentValue << endl;
        time++;
        }

        outfile.close();
        probfile.close();
        cout << "Total stuck count (local optima): " << stuck_count
<< endl;

}

int main() {
        clock_t start, end;
        start = clock();
        srand(time(0));
        CUBE current;
        generateInitialState(&current);
        current.value = findValue(current, 0);

        cout << "Initial State:" << endl;
        printState(current);
        cout << "Initial Value: " << current.value << endl;

        simulatedAnnealing(current);

        cout << "Final State after Simulated Annealing:" << endl;
        printState(current);
        cout << "Final Value: " << findValue(current,0) << endl;
        end = clock();

        // Menghitung lama jalannya program
        double time_taken = double(end - start) /
double(CLOCKS_PER_SEC);
        cout << "Time taken by program is : " << fixed
```

```
        << time_taken << setprecision(5);
        cout << " sec " << endl;
        return 0;
}
```

2.2.7. Genetic Algorithm

Untuk mendapatkan state value dari masing-masing state berikut adalah implementasi dari fungsi objektif untuk magic cube.

Nama File	
GeneticAlgorithm	Sebagai implementasi dari Genetic Algorithm
Variabel	
value	Merepresentasikan jumlah garis (baris, kolom, tiang, diagonal) yang memenuhi *magic constant* 315
state	Representasi Matriks 3 dimensi dalam sebuah array satu dimensi
POPULATION_SIZE	Jumlah individu populasi di tiap iterasi. Nilainya berdasarkan input user.
ITERATION_TOTAL	Total jumlah iterasi. Nilainya berdasarkan input user.
population	Sekumpulan individu yang di-generate bersamaan atau dihasilkan beriringan
generation	Memantau jumlah iterasi yang sudah dilakukan.
bestGenValue, sumGenValue	Memantau nilai dari tiap generasi, untuk kebutuhan visualisasi grafik
finalState	Individu dengan nilai state terbaik di iterasi terakhir
Fungsi dan Prosedur	
tournamentSelection	Melakukan seleksi parent dari sejumlah individu pada populasi. Dipilih acak di antara kandidat yang berjumlah seperempat dari jumlah populasi. Memilih kandidat dengan value tertinggi untuk lanjut ke tahap crossover.
crossover	Melakukan crossover dengan swap pasangan

	individu. Pada layer tengah, dilakukan swap segmen. Lalu, pada layer tepi, dilakukan penyesuaian dengan menukarkan angka-angka yang double.
randomSuccessor	Mendapatkan sebuah neighbor secara random
generateInitialState	Mengisi magic cube dengan nilai acak untuk initial state

Berikut merupakan Source Code implementasi algoritma tersebut.

```
#include <bits/stdc++.h>
#include "LocalSearch.hpp"
using namespace std;

CUBE population[10000];
int POPULATION_SIZE;
int ITERATION_TOTAL;

void tournamentSelection()
{
    CUBE selected[POPULATION_SIZE];
    int maxVal=-1;
    for(int i=0; i<POPULATION_SIZE; i++)
    {
        CUBE candidates[POPULATION_SIZE/4];
        for(int j=0; j<POPULATION_SIZE/4; j++)
        {
            candidates[j]=population[(rand()%POPULATION_SIZE)];
            if(findValue(candidates[j],109)>maxVal)
{maxVal=candidates[j].value; selected[i]=candidates[j];}
        }
        maxVal=-1;
    }
    for(int i=0; i<POPULATION_SIZE; i++) population[i]=selected[i];
}

bool isDouble(int x, int num)
{
    for(int i=2; i<=4; i++)
    {
        for(int j=1; j<=5; j++)
        {
            for(int k=1; k<=5; k++)
            {
                if(population[x].state[i][j][k]==num) return true;
            }
        }
    }
    return false;
}
void crossover(int x, int y)
{
```

```
    for(int i=2; i<=4; i++)
    {
        for(int j=1; j<=5; j++)
        {
            for(int k=1; k<=5; k++)
            {

swap(population[x].state[i][j][k],population[y].state[i][j][k]);
            }
        }
    }
    struct GeneticAlgoritm
    {
        int i;
        int j;
        int k;
    };
    vector<GeneticAlgoritm> A,B;
    for(int i=1; i<=5; i+=4)
    {
        for(int j=1; j<=5; j++)
        {
            for(int k=1; k<=5; k++)
            {
                if(isDouble(x,population[x].state[i][j][k]))
                {
                    GeneticAlgoritm a;
                    a.i=i; a.j=j; a.k=k;
                    A.push_back(a);
                }
                if(isDouble(y,population[y].state[i][j][k]))
                {
                    GeneticAlgoritm a;
                    a.i=i; a.j=j; a.k=k;
                    B.push_back(a);
                }
            }
        }
    }
    for(int i=0; i<A.size(); i++)
    {

swap(population[x].state[A[i].i][A[i].j][A[i].k],population[y].state
[B[i].i][B[i].j][B[i].k]);
    }
    for(int i=0; i<POPULATION_SIZE; i++)
    {
        population[x].value=findValue(population[x],109);
        population[y].value=findValue(population[y],109);
    }
}

int main() {
    clock_t start, end;
    start = clock();
    srand(time(0));
    cin>>POPULATION_SIZE>>ITERATION_TOTAL;
    bool found = false;
```

```
       int generation = 0;
    CUBE finalState;
    int bestValue, idxBestState = 0;

    ofstream bestfile("best_objective_values.txt");
    ofstream avgfile("avg_objective_values.txt");

    //generate inisial population
    for(int i=0; i<POPULATION_SIZE; i++)
    {
        generateInitialState(&population[i]);
        population[i].value=findValue(population[i],109);
    }

    // print best initial state
    cout<<"INITIAL STATE: "<<endl;
    CUBE bestInitialState = population[0];
    int bestInitialValue = findValue(bestInitialState, 109);
    int sumInitialValue = findValue(bestInitialState, 109);

    for (int i=1; i<POPULATION_SIZE; i++)
    {
        sumInitialValue += findValue(population[i], 109);
        if (findValue(population[i], 109) > bestInitialValue) {
            bestInitialState = population[i];
            bestInitialValue = findValue(bestInitialState, 109);
        }
    }
    printState(bestInitialState);
    cout<<"INITIAL OBJECTIVE VALUE: "<<bestInitialValue<<endl<<endl;
    bestfile << bestInitialValue << endl;
    avgfile <<
double(sumInitialValue)/double(POPULATION_SIZE)<<endl;


       while (!found && generation<ITERATION_TOTAL)
       {
        tournamentSelection();
        // if the individual having 0 fitness score then we know
that we have reached to the target so break the loop
        for(int i=0; i<POPULATION_SIZE; i++)
        {
            if(population[i].value == 109)
            {
                found = true;
                idxBestState = i;
                break;
            }
        }

        for(int i=0; i<POPULATION_SIZE/2; i+=2)
        {
            crossover(i,i+1);
        }

        //mutation
        for (int i=0; i<POPULATION_SIZE; i++) {
            population[i]=randomSuccessor(population[i]);
```

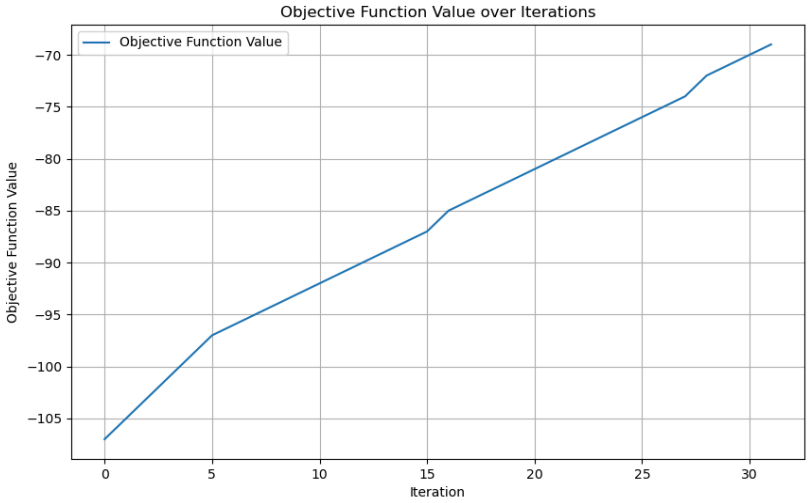
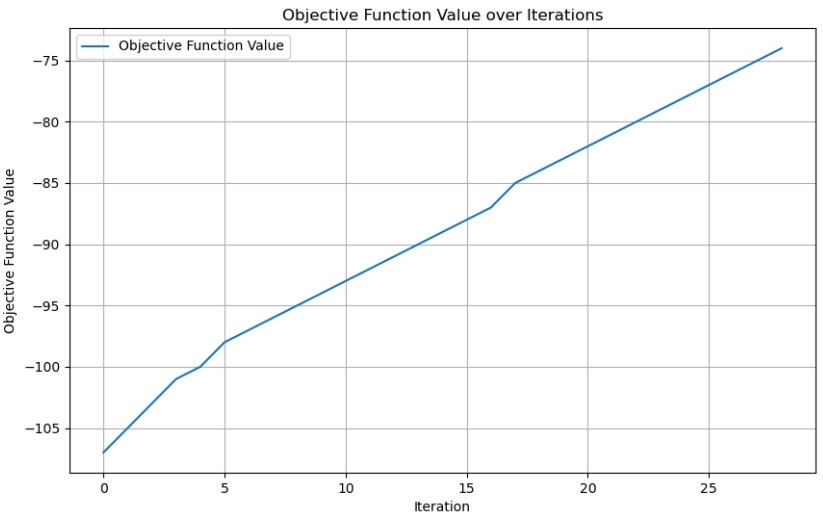
```
        }

        int bestGenValue = 0;
        int sumGenValue = 0;
        for (int i=0; i<POPULATION_SIZE; i++) {
            sumGenValue += findValue(population[i], 109);
            if (findValue(population[i], 109) > bestGenValue) {
                bestGenValue = findValue(population[i], 109);
            }
        }
        bestfile << bestGenValue << endl;
        avgfile <<
double(sumGenValue)/double(POPULATION_SIZE)<<endl;

        generation++;
    }

    for(int i=0; i<POPULATION_SIZE; i++)
    {
        if (findValue(population[i], 109) > bestValue)
        {
            bestValue = findValue(population[i], 109);
            idxBestState = i;
        }
    }

    finalState = population[idxBestState];
    cout<<"FINAL STATE: "<<endl;
    printState(finalState);
    cout<<"FINAL OBJECTIVE VALUE: "<<findValue(finalState,
109)<<endl<<endl;
    bestfile << findValue(finalState, 109) << endl;
    end = clock();
    // Calculating total time taken by the program.
    double time_taken = double(end - start) /
double(CLOCKS_PER_SEC);
    cout << "Time taken by program is : " << fixed
         << time_taken << setprecision(5);
    cout << " sec " << endl;
}
```

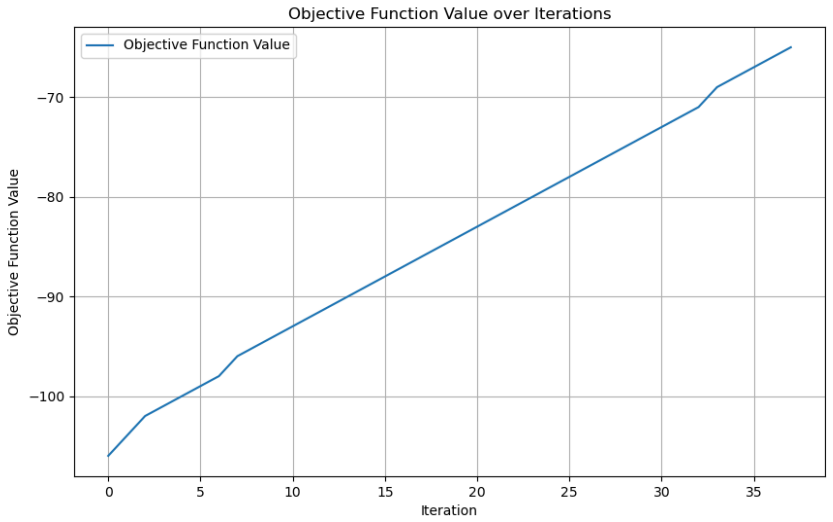
BAB 3
HASIL DAN ANALISIS

3.1. Hasil Eksperimen

3.1.1. Hill-Climbing Steepest Ascent

Pada bagian ini, diterapkan tiga percobaan yang masing-masing memiliki konfigurasi jumlah iterasi maksimum yang berbeda untuk algoritma Hill Climbing Steepest Ascent.

Percobaan	Grafik State Value Terhadap Jumlah Iterasi
1	
2	

3	

Berikut adalah state awal dan akhir dari masing-masing percobaan

Percobaan	**Initial State**	**Final State**

1	State Awal: Layer 1 20 19 122 32 12 8 1 89 88 44 30 97 54 112 36 100 94 66 29 87 99 93 55 104 83 Layer 2 23 53 67 34 43 117 10 63 45 2 111 86 80 98 22 95 119 96 35 73 41 116 39 51 78 Layer 3 37 61 11 123 15 52 21 76 48 59 125 40 50 102 13 26 106 77 47 14 64 62 9 121 70 Layer 4 49 27 6 25 46 109 110 120 58 85 28 124 60 105 108 74 90 17 71 38 57 18 113 84 69 Layer 5 75 115 5 103 114 3 24 4 42 33 82 107 68 56 81 101 31 65 92 91 7 72 118 79 16 Nilai Objective Function Awal: -109	State Akhir: Layer 1 20 19 32 122 27 66 12 61 88 81 30 97 7 55 36 100 94 125 2 60 99 93 109 48 42 Layer 2 112 53 67 34 113 117 10 98 45 29 23 86 80 63 22 95 119 31 35 73 41 116 39 106 78 Layer 3 37 89 103 123 46 52 111 76 104 87 21 40 50 102 1 26 51 77 47 14 64 16 9 121 70 Layer 4 49 13 6 25 15 107 110 120 58 85 28 84 4 105 108 74 90 17 83 38 57 18 43 124 69 Layer 5 75 115 5 11 114 3 24 59 71 33 82 8 68 62 44 101 96 65 92 91 54 72 118 79 56 Nilai Objective Function Akhir: -69

2	State Awal: Layer 1 42 19 94 71 30 91 14 76 99 72 3 96 82 57 60 102 89 70 35 66 17 32 45 20 109 Layer 2 25 41 39 116 98 5 115 10 64 13 104 87 23 33 124 4 121 61 47 46 122 7 24 123 113 Layer 3 68 81 88 69 101 111 49 52 92 67 84 80 74 6 31 97 77 59 27 118 9 8 114 65 44 Layer 4 62 100 12 34 125 110 93 40 48 83 95 119 18 1 2 16 117 105 22 55 75 58 50 103 29 Layer 5 53 107 85 73 28 38 90 106 54 36 43 63 11 112 56 120 37 108 26 79 15 51 78 21 86 Nilai Objective Function Awal: -109	State Akhir: Layer 1 25 39 42 71 8 91 88 76 99 72 3 50 82 90 60 102 19 70 35 66 94 119 45 20 109 Layer 2 17 41 93 86 53 5 115 114 64 13 104 87 23 106 44 67 65 61 47 46 122 7 24 123 113 Layer 3 68 95 14 69 101 30 49 52 92 111 84 73 74 6 16 124 77 59 27 120 9 80 10 121 97 Layer 4 18 100 12 34 125 110 89 40 21 83 81 32 62 1 2 31 117 105 22 4 75 58 96 103 29 Layer 5 98 107 85 55 28 38 57 33 54 36 43 63 11 112 56 118 37 108 26 79 15 51 78 48 116 Nilai Objective Function Akhir: -74

3	State Awal: Layer 1 59 107 81 113 110 12 120 112 29 67 44 124 104 9 87 121 108 27 8 40 64 38 73 88 92 Layer 2 41 2 47 115 63 25 62 125 119 17 20 26 45 66 95 68 43 11 16 21 90 58 99 76 94 Layer 3 85 84 96 80 114 93 46 19 51 49 122 35 72 77 53 117 116 10 105 109 89 48 24 14 60 Layer 4 15 57 83 18 42 103 56 22 106 111 71 33 86 69 97 101 74 75 4 39 37 23 7 5 78 Layer 5 50 65 102 70 36 118 100 55 34 52 123 3 32 61 30 82 1 91 31 54 98 13 79 28 6 Nilai Objective Function Awal: -108	State Akhir: Layer 1 124 107 56 14 110 117 120 20 5 67 118 75 104 9 40 121 12 27 103 4 114 1 108 46 94 Layer 2 41 2 47 38 63 80 86 113 119 44 112 26 45 66 95 91 43 11 16 21 6 58 99 76 92 Layer 3 85 84 96 25 64 93 101 19 51 49 52 35 72 77 53 7 39 10 37 109 89 48 24 125 60 Layer 4 15 57 79 116 42 8 73 22 106 111 59 88 62 69 97 33 74 71 87 18 31 23 81 29 78 Layer 5 50 65 102 70 36 17 100 55 34 105 68 83 32 61 30 82 115 123 122 54 98 13 3 28 90 Nilai Objective Function Akhir: -65

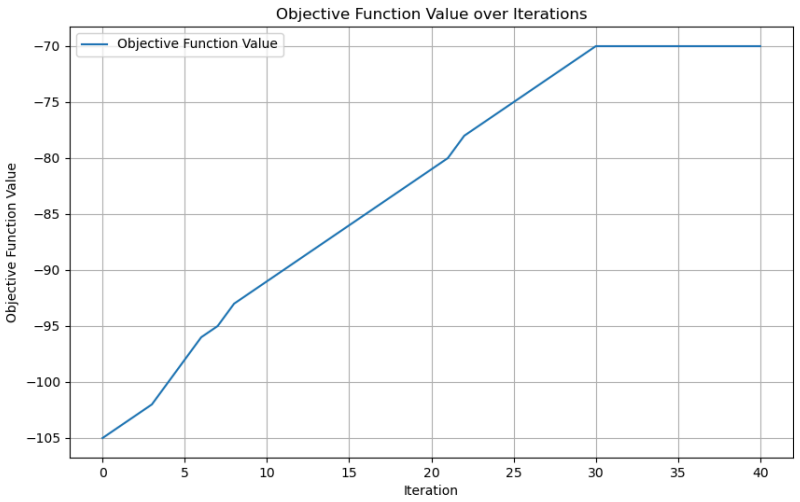
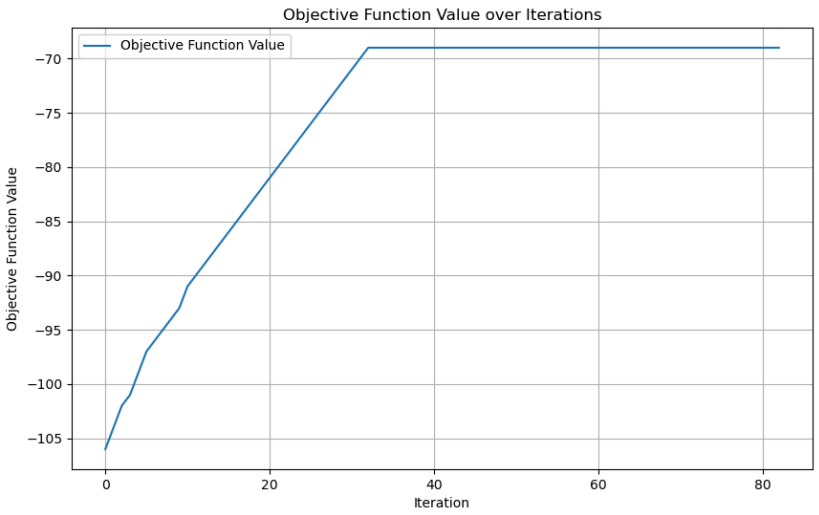
Parameter	Durasi	Jumlah Iterasi	Initial Value	Final Value
Percobaan 1	0.31464 detik	32	-109	-69
Percobaan 2	0.241704 detik	29	-109	-74
Percobaan 3	0.297749 detik	38	-108	-65

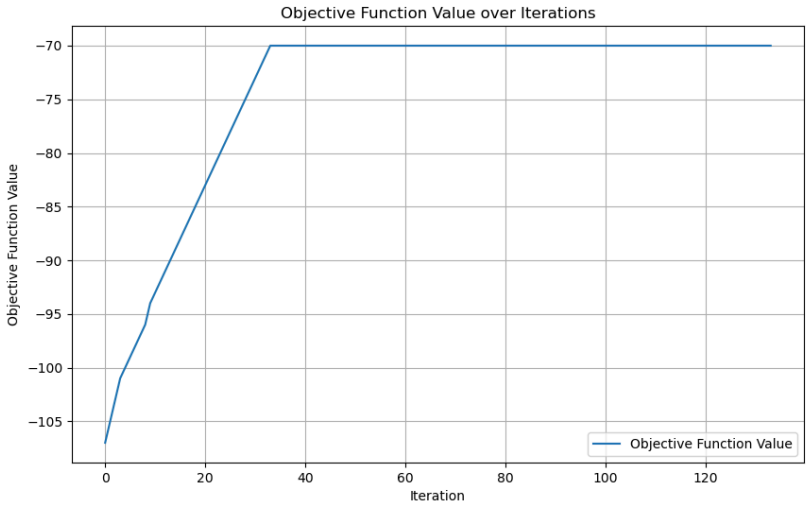
Berdasarkan hasil percobaan menggunakan algoritma Hill Climbing Steepest Ascent, terlihat bahwa semakin banyak iterasi, cenderung menghasilkan nilai final state yang lebih baik. Dalam hal ini, *state value* berbanding lurus dengan jumlah iterasi. Pada Percobaan 1, dengan durasi 0.31464 detik dan 32 iterasi, diperoleh nilai final state sebesar -69 dari initial value -109. Sementara itu, Percobaan 2 yang melalui 29 iterasi dan memiliki durasi lebih singkat sebesar 0.241704 detik hanya mencapai nilai final state -74, meskipun memiliki initial

value yang sama dengan percobaan 1. Percobaan 3 yang melalui 38 iterasi dengan initial value yang lebih tinggi dari yang lain sebesar -108 dengan durasi pencarian sebesar 0.297749 detik mencapai nilai final state terbaik sebesar -65. Hasil ini menunjukkan bahwa semakin lama waktu pencarian dan semakin banyak iterasi yang dilakukan, algoritma memiliki peluang lebih besar untuk menemukan solusi yang lebih optimal, meskipun hasilnya juga bergantung pada state awal yang dipilih. Algoritma berhenti ketika tidak ditemukan state neighbour yang lebih baik, sehingga lebih banyak iterasi atau durasi memungkinkan pencarian solusi yang lebih baik.

3.1.2. Hill-Climbing with Sideways Move

Pada bagian ini, diterapkan tiga percobaan yang masing-masing memiliki konfigurasi jumlah iterasi maksimum yang berbeda untuk algoritma Hill Climbing with Sideways Move

Percobaan	Grafik State Value Terhadap Jumlah Iterasi
1	
2	

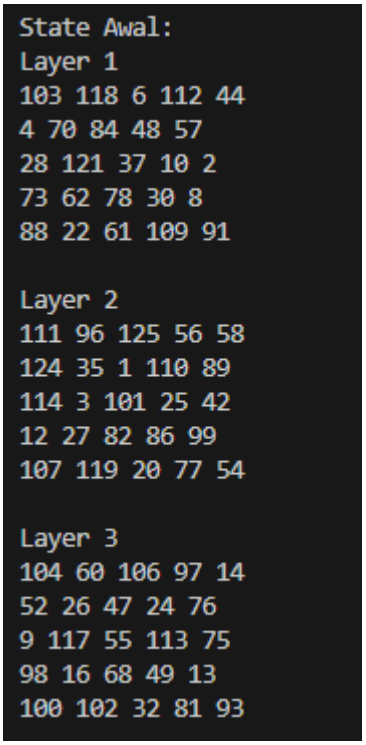
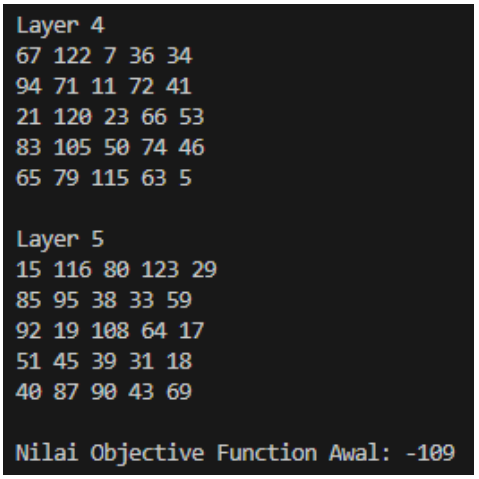
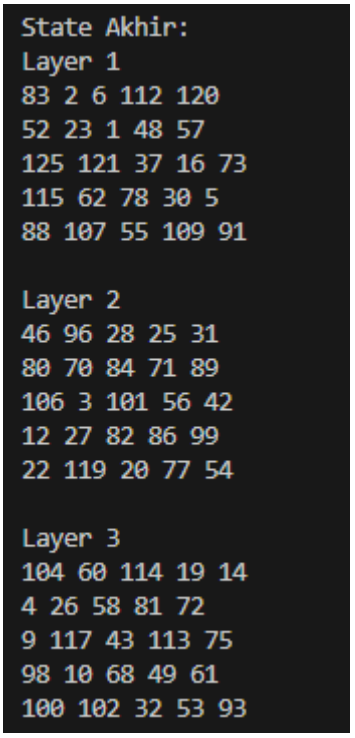
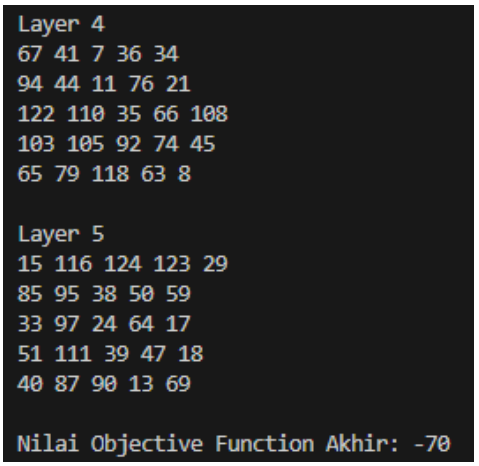
3	

Berikut adalah state awal dan akhir dari masing-masing percobaan

Percobaan	**Initial State**	**Final State**

1	State Awal: Layer 1 48 36 11 125 123 76 58 16 61 71 24 55 13 53 73 97 34 96 8 42 110 15 72 69 52 Layer 2 120 100 118 105 57 17 112 67 115 10 95 108 83 62 116 38 35 92 119 84 22 25 85 9 70 Layer 3 64 113 94 21 65 30 122 63 32 77 46 37 60 101 29 75 3 93 7 104 114 54 1 106 81 Layer 4 4 74 44 68 79 87 88 14 12 47 121 82 51 31 18 50 5 28 56 19 86 109 102 45 90 Layer 5 43 26 59 33 80 41 91 103 40 23 99 39 2 6 107 111 98 124 117 49 89 78 27 20 66 Nilai Objective Function Awal: -107	State Akhir: Layer 1 8 83 118 55 123 76 58 16 35 25 24 125 13 108 73 97 34 96 48 42 110 15 72 69 52 Layer 2 62 32 11 105 61 122 112 67 115 10 71 117 60 47 116 38 57 92 64 84 22 65 85 75 44 Layer 3 119 100 94 21 49 9 17 63 113 86 46 37 36 68 1 30 3 93 7 98 114 54 29 106 81 Layer 4 4 74 120 101 79 87 88 14 12 109 121 82 51 31 18 50 5 28 56 19 53 70 102 45 90 Layer 5 43 26 59 33 80 41 91 103 40 23 77 95 2 6 107 111 104 124 99 39 89 78 27 20 66 Nilai Objective Function Akhir: -70

2	State Awal: Layer 1 83 23 59 66 69 78 102 121 50 56 52 55 21 93 110 63 28 39 8 3 47 109 73 18 12 Layer 2 29 31 101 34 105 24 17 100 85 106 124 49 54 64 117 40 119 5 74 92 104 15 96 76 65 Layer 3 1 42 125 19 89 82 112 32 14 86 103 79 108 16 9 27 41 38 10 43 113 115 84 53 45 Layer 4 37 7 58 33 120 13 75 48 122 51 71 4 36 81 20 57 35 80 94 6 62 123 61 98 116 Layer 5 111 25 2 68 88 107 22 118 90 11 30 114 60 97 87 70 99 44 67 46 26 72 91 77 95 Nilai Objective Function Awal: -108	State Akhir: Layer 1 105 28 59 108 69 89 102 121 63 56 51 55 23 93 110 113 21 39 8 3 47 109 73 43 77 Layer 2 31 115 101 34 18 24 17 19 85 106 124 49 14 64 116 48 119 114 25 10 104 15 67 29 65 Layer 3 30 33 95 100 78 82 112 32 54 90 103 79 66 16 9 27 41 38 92 83 76 50 84 53 40 Layer 4 122 7 58 5 120 13 75 45 37 52 36 46 71 81 20 57 35 80 94 6 62 123 61 98 117 Layer 5 111 74 2 68 88 107 22 118 42 11 1 86 60 97 87 70 99 44 96 4 26 72 91 12 125 Nilai Objective Function Akhir: -69

3	 	 

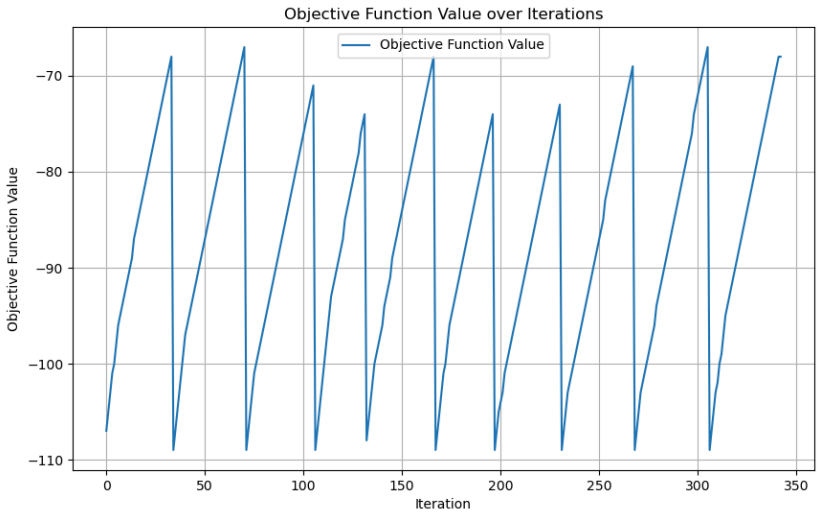
Parameter	Durasi	Jumlah Iterasi	Initial Value	Final Value	Max Sideways Move
Percobaan 1	0.317402 detik	41	-107	-70	10
Percobaan 2	0.595917 detik	83	-108	-69	50
Percobaan 3	0.921831 detik	134	-109	-70	100

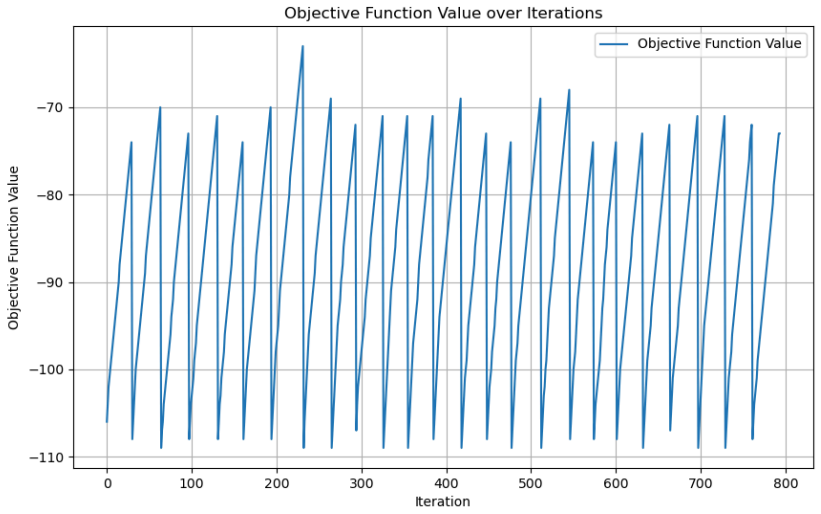
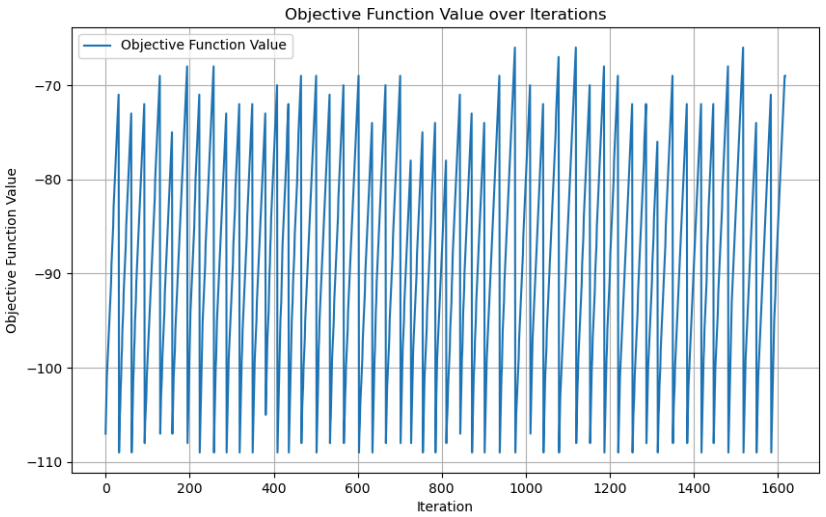
Berdasarkan hasil percobaan, pencarian solusi magic cube menggunakan Hill Climbing with Sideways Move menunjukkan hasil yang konsisten dalam mencapai nilai final

state yang hampir serupa, meskipun dengan disertai adanya variasi dalam durasi dan jumlah iterasi. Pada Percobaan 1, nilai final state sebesar -70 dicapai dalam 0.317 detik dengan 41 iterasi dan maksimum 10 sideways moves. Percobaan 2 menghasilkan nilai final state -69 dalam 0.596 detik dan 83 iterasi dengan 50 sideways moves. Sementara itu, Percobaan 3 membutuhkan waktu terlama, yaitu 0.922 detik dan melalui 134 iterasi, namun hanya menghasilkan final state -70 meskipun menggunakan hingga 100 sideways moves. Hasil ini menunjukkan bahwa peningkatan jumlah sideways moves dapat membantu algoritma untuk keluar dari local optima, namun tidak selalu menjamin peningkatan signifikan pada nilai *final state*, terutama ketika nilai *initial state* semakin rendah. Algoritma ini cenderung terjebak pada *local optima* sehingga tidak mencapai nilai state yang jauh lebih baik di akhir iterasi.

3.1.3. Random Restart Hill-Climbing

Pada bagian ini, diterapkan tiga percobaan yang masing-masing memiliki konfigurasi jumlah iterasi maksimum yang berbeda untuk algoritma Random Restart Hill Climbing.

Percobaan	Grafik State Value Terhadap Jumlah Iterasi
1	

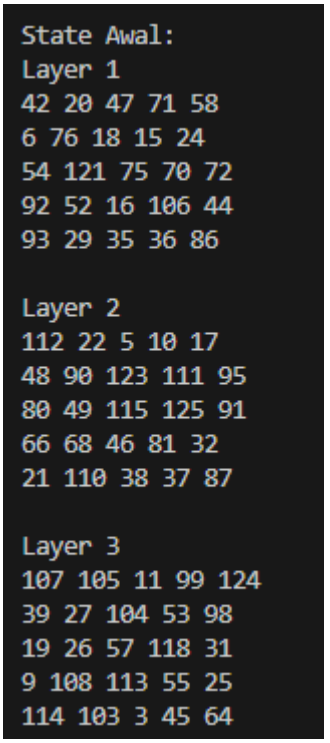
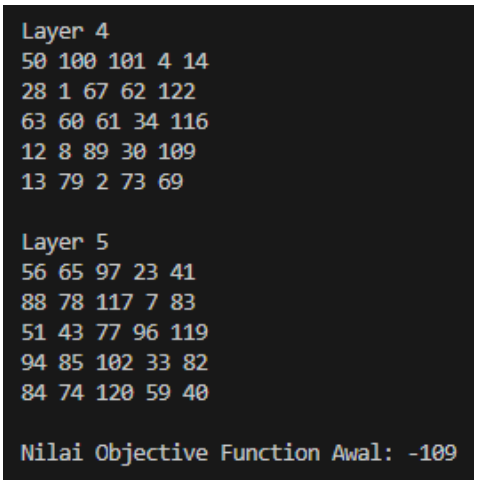
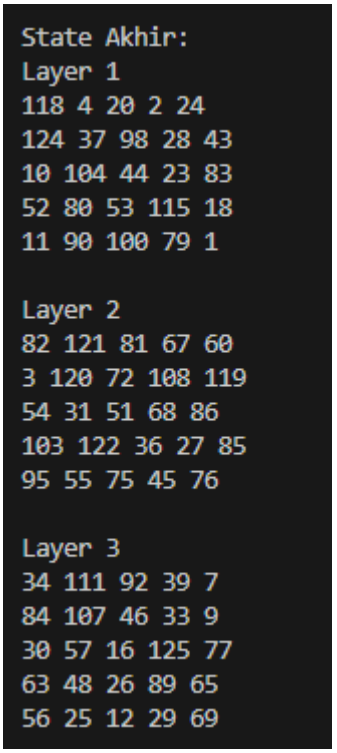
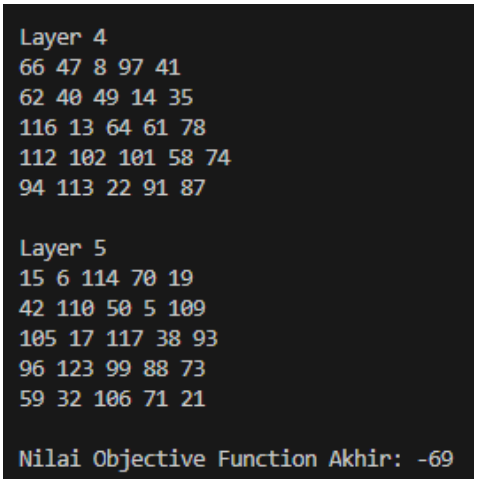
2	
3	

Berikut adalah state awal dan akhir dari masing-masing percobaan

Percobaan	**Initial State**	**Final State**

1	State Awal: Layer 1 14 90 89 70 104 119 13 81 47 87 16 118 122 110 10 26 56 115 20 120 29 84 102 91 79 Layer 2 95 32 62 53 61 63 82 51 99 9 6 101 86 1 43 75 8 3 34 40 105 21 5 49 125 Layer 3 117 69 68 22 30 58 18 121 112 116 46 76 54 98 100 93 50 55 88 7 65 94 37 108 107 Layer 4 60 64 36 78 12 80 111 66 23 106 124 41 4 109 35 77 103 96 97 42 45 17 67 74 113 Layer 5 2 73 92 52 48 85 114 11 39 27 33 31 57 25 24 83 38 123 19 71 44 28 59 72 15 Nilai Objective Function Awal: -109	State Akhir: Layer 1 12 90 57 78 9 112 92 7 74 71 70 91 62 46 125 67 17 65 120 81 54 25 124 56 29 Layer 2 49 34 4 114 94 22 15 27 42 61 41 11 52 37 28 24 50 36 26 84 77 58 111 96 48 Layer 3 6 38 69 3 116 39 51 105 60 110 102 123 115 13 107 86 2 104 44 5 82 101 80 23 99 Layer 4 75 121 79 20 83 109 117 73 21 113 14 55 18 122 66 53 19 100 93 43 64 63 45 59 10 Layer 5 31 32 106 76 108 33 40 87 118 8 88 35 119 97 103 85 72 98 30 1 47 68 16 89 95 Nilai Objective Function Akhir: -68

2	State Awal: Layer 1 41 122 42 31 99 23 124 120 62 63 68 106 95 113 17 92 10 59 105 103 7 66 61 44 73 Layer 2 20 74 57 54 19 35 3 104 100 90 48 9 58 11 39 2 40 37 108 33 64 118 84 13 6 Layer 3 77 109 51 38 123 49 28 4 46 82 16 75 91 26 80 69 67 24 115 21 71 50 52 60 29 Layer 4 114 86 116 27 89 45 88 22 70 119 117 34 32 112 36 87 101 98 94 81 76 1 25 12 14 Layer 5 15 83 107 65 97 55 56 110 111 8 18 78 30 5 102 96 121 72 43 47 53 79 125 85 93 Nilai Objective Function Awal: -108	State Akhir: Layer 1 102 53 110 103 73 95 86 100 17 76 10 14 59 39 75 21 96 104 37 114 87 66 35 119 118 Layer 2 4 13 24 12 93 84 124 106 20 70 68 74 115 29 67 121 40 30 2 19 47 64 94 55 27 Layer 3 65 78 28 72 125 23 69 16 49 44 6 42 25 54 45 80 58 109 34 32 48 5 91 117 122 Layer 4 71 60 92 8 108 98 82 36 18 46 7 97 107 11 38 83 89 41 33 101 56 81 52 63 22 Layer 5 3 111 61 120 62 15 123 57 50 79 116 88 9 51 90 1 113 31 85 112 77 99 43 105 26 Nilai Objective Function Akhir: -73

3	 	 

Parameter	Durasi	Jumlah Iterasi	Initial Value	Final Value	Banyak Restart	Max Jumlah Restart
Percobaan 1	2.35006 detik	343	-109	-68	10	10
Percobaan 2	5.18891 detik	794	-108	-73	25	25
Percobaan 3	10.3179 detik	1618	-109	-69	50	50

Parameter	Banyak Iterasi Per Restart
Percobaan 1	Jumlah iterasi pada restart ke-1: 34 Jumlah iterasi pada restart ke-2: 36 Jumlah iterasi pada restart ke-3: 34 Jumlah iterasi pada restart ke-4: 25 Jumlah iterasi pada restart ke-5: 34 Jumlah iterasi pada restart ke-6: 29 Jumlah iterasi pada restart ke-7: 33 Jumlah iterasi pada restart ke-8: 36 Jumlah iterasi pada restart ke-9: 37 Jumlah iterasi pada restart ke-10: 35
Percobaan 2	Jumlah iterasi pada restart ke-1: 30 Jumlah iterasi pada restart ke-2: 33 Jumlah iterasi pada restart ke-3: 32 Jumlah iterasi pada restart ke-4: 33 Jumlah iterasi pada restart ke-5: 29 Jumlah iterasi pada restart ke-6: 32 Jumlah iterasi pada restart ke-7: 37 Jumlah iterasi pada restart ke-8: 32 Jumlah iterasi pada restart ke-9: 28 Jumlah iterasi pada restart ke-10: 31 Jumlah iterasi pada restart ke-11: 28 Jumlah iterasi pada restart ke-12: 29 Jumlah iterasi pada restart ke-13: 32 Jumlah iterasi pada restart ke-14: 29 Jumlah iterasi pada restart ke-15: 28 Jumlah iterasi pada restart ke-16: 34 Jumlah iterasi pada restart ke-17: 33 Jumlah iterasi pada restart ke-18: 27 Jumlah iterasi pada restart ke-19: 26 Jumlah iterasi pada restart ke-20: 30 Jumlah iterasi pada restart ke-21: 31 Jumlah iterasi pada restart ke-22: 32 Jumlah iterasi pada restart ke-23: 31 Jumlah iterasi pada restart ke-24: 31 Jumlah iterasi pada restart ke-25: 31
Percobaan 3	Jumlah iterasi pada restart ke-1: 32 Jumlah iterasi pada restart ke-2: 29 Jumlah iterasi pada restart ke-3: 30 Jumlah iterasi pada restart ke-4: 36 Jumlah iterasi pada restart ke-5: 28 Jumlah iterasi pada restart ke-6: 35 Jumlah iterasi pada restart ke-7: 28

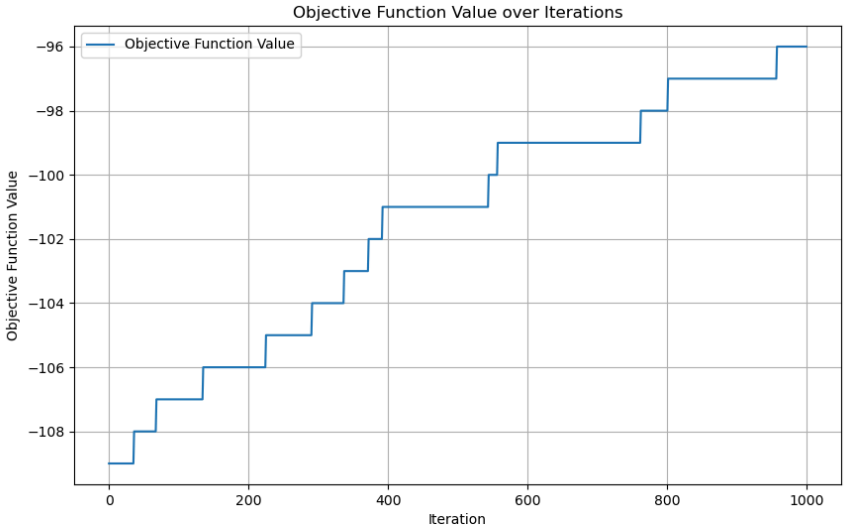
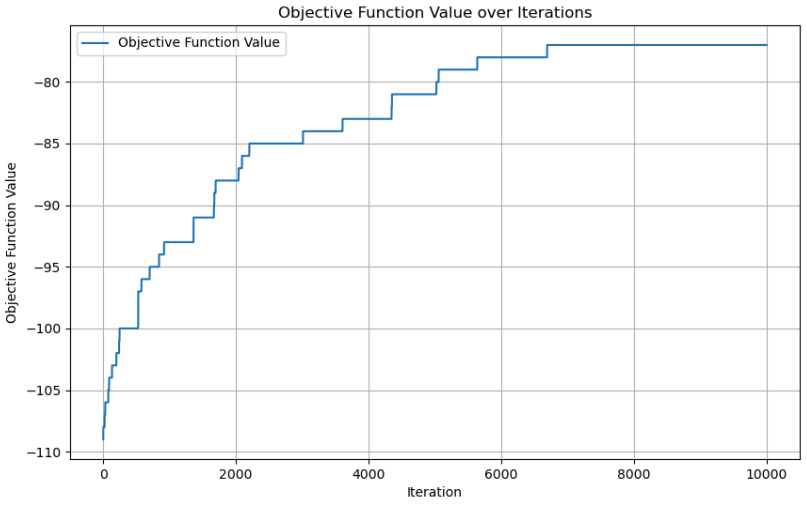
	Jumlah iterasi pada restart ke-8: 33 Jumlah iterasi pada restart ke-9: 29 Jumlah iterasi pada restart ke-10: 30 Jumlah iterasi pada restart ke-11: 30 Jumlah iterasi pada restart ke-12: 30 Jumlah iterasi pada restart ke-13: 27 Jumlah iterasi pada restart ke-14: 26 Jumlah iterasi pada restart ke-15: 29 Jumlah iterasi pada restart ke-16: 35 Jumlah iterasi pada restart ke-17: 31 Jumlah iterasi pada restart ke-18: 32 Jumlah iterasi pada restart ke-19: 35 Jumlah iterasi pada restart ke-20: 31 Jumlah iterasi pada restart ke-21: 31 Jumlah iterasi pada restart ke-22: 34 Jumlah iterasi pada restart ke-23: 24 Jumlah iterasi pada restart ke-24: 27 Jumlah iterasi pada restart ke-25: 29 Jumlah iterasi pada restart ke-26: 25 Jumlah iterasi pada restart ke-27: 32 Jumlah iterasi pada restart ke-28: 27 Jumlah iterasi pada restart ke-29: 29 Jumlah iterasi pada restart ke-30: 35 Jumlah iterasi pada restart ke-31: 36 Jumlah iterasi pada restart ke-32: 35 Jumlah iterasi pada restart ke-33: 30 Jumlah iterasi pada restart ke-34: 36 Jumlah iterasi pada restart ke-35: 40 Jumlah iterasi pada restart ke-36: 32 Jumlah iterasi pada restart ke-37: 33 Jumlah iterasi pada restart ke-38: 32 Jumlah iterasi pada restart ke-39: 33 Jumlah iterasi pada restart ke-40: 32 Jumlah iterasi pada restart ke-41: 26 Jumlah iterasi pada restart ke-42: 35 Jumlah iterasi pada restart ke-43: 33 Jumlah iterasi pada restart ke-44: 33 Jumlah iterasi pada restart ke-45: 28 Jumlah iterasi pada restart ke-46: 34 Jumlah iterasi pada restart ke-47: 35 Jumlah iterasi pada restart ke-48: 30 Jumlah iterasi pada restart ke-49: 34 Jumlah iterasi pada restart ke-50: 32

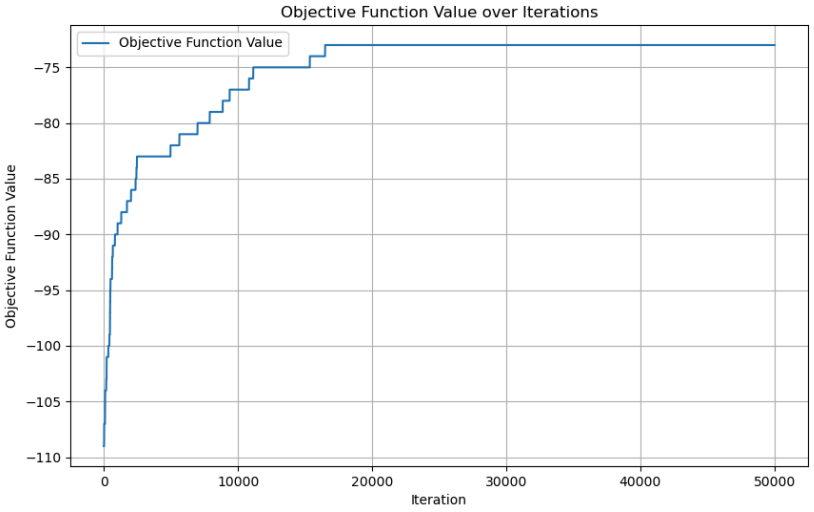
Berdasarkan data yang diperoleh, peningkatan jumlah iterasi dan restart tidak selalu menjamin kualitas solusi yang lebih baik, karena adanya faktor restart yang acak dalam algoritma Hill Climbing dengan random restart. Pada percobaan pertama, dengan **10 restart** dan **343 iterasi**, algoritma menghasilkan nilai akhir -68 dalam durasi **2.35006 detik**.

Meskipun percobaan kedua memiliki lebih banyak **restart (25)** dan **iterasi (794)**, nilai akhir yang dihasilkan adalah -73, menunjukkan peningkatan yang tidak signifikan meskipun durasi eksekusi lebih lama (**5.18891 detik**) dan jumlah restart yang lebih banyak. Percobaan ketiga, dengan **50 restart** dan **1618 iterasi**, menghasilkan nilai akhir -69 dalam durasi lebih lama lagi (**10.3179 detik**). Perbedaan jumlah iterasi per restart yang cukup besar di setiap percobaan menunjukkan bahwa meskipun algoritma memiliki lebih banyak kesempatan untuk mencari solusi, faktor restart yang acak menyebabkan kualitas solusi kurang bisa optimal.

3.1.4. Stochastic Hill-Climbing

Pada bagian ini, diterapkan tiga percobaan yang masing-masing memiliki konfigurasi jumlah iterasi maksimum yang berbeda untuk algoritma Stochastic Hill Climbing.

Percobaan	Grafik State Value Terhadap Jumlah Iterasi
1	
2	

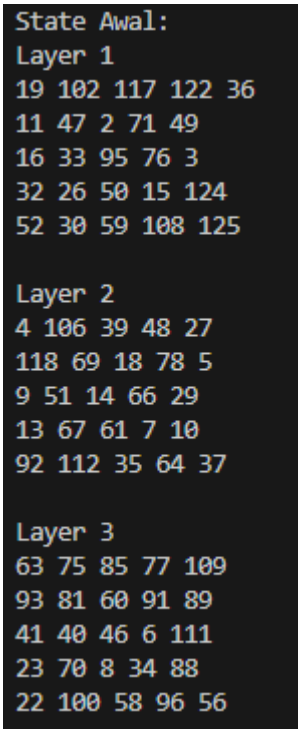
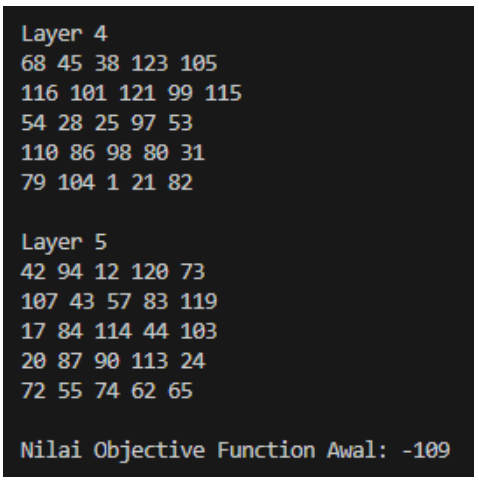
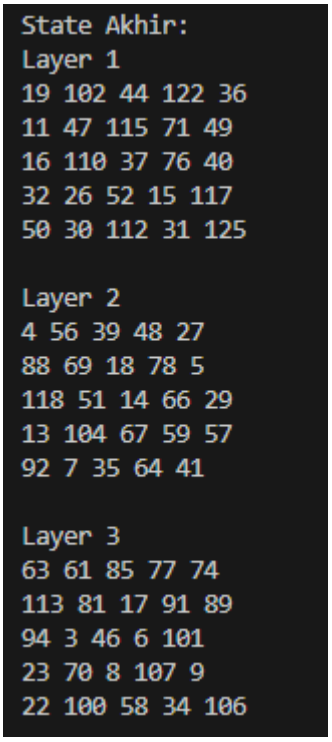
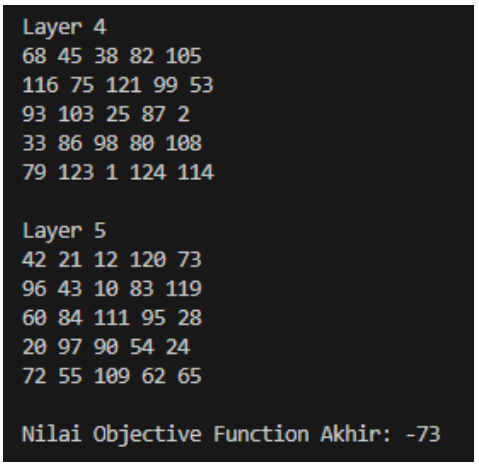
3	

Berikut adalah state awal dan akhir dari masing-masing percobaan

Percobaan	**Initial State**	**Final State**

1	State Awal: Layer 1 36 114 78 42 59 106 79 1 50 91 18 43 73 117 84 4 6 100 24 86 107 20 54 39 101 Layer 2 108 71 55 28 67 8 89 29 94 105 32 90 52 3 10 109 121 56 57 17 110 118 58 96 41 Layer 3 92 81 9 122 87 44 30 46 72 49 97 119 85 98 66 13 69 51 102 62 25 88 111 53 22 Layer 4 16 31 113 76 103 123 64 45 40 65 27 5 21 80 38 68 61 47 15 60 112 125 34 19 26 Layer 5 115 83 37 116 120 70 12 77 95 93 23 124 11 7 99 82 74 2 63 35 33 104 14 48 75 Nilai Objective Function Awal: -109	State Akhir: Layer 1 80 114 78 42 59 106 79 1 32 91 18 43 73 117 84 4 6 100 24 86 107 70 63 39 101 Layer 2 12 71 55 28 67 8 30 29 94 105 50 90 52 3 85 109 121 56 57 17 110 118 58 96 41 Layer 3 92 112 9 122 87 44 89 46 72 49 97 119 10 98 66 13 2 51 102 62 69 88 111 113 22 Layer 4 16 31 53 76 103 123 64 45 40 65 27 5 21 36 38 68 61 47 15 82 81 125 34 19 26 Layer 5 115 83 37 116 120 20 108 95 77 93 23 124 11 7 99 60 74 25 54 35 33 104 14 48 75 Nilai Objective Function Akhir: -96

2	State Awal: Layer 1 103 92 113 46 118 56 8 48 32 124 68 21 27 30 5 107 120 53 75 91 95 40 16 65 52 Layer 2 101 13 36 51 93 9 18 41 7 100 86 98 10 64 117 38 43 72 15 77 54 24 49 12 90 Layer 3 115 88 14 20 89 85 123 39 109 61 81 74 44 104 11 94 111 31 42 80 37 55 50 112 2 Layer 4 4 102 26 125 105 83 59 84 35 34 17 29 23 57 76 82 19 22 45 67 108 3 6 25 79 Layer 5 60 78 87 71 58 97 110 62 119 70 96 106 121 73 69 47 122 116 114 99 33 1 28 63 66 Nilai Objective Function Awal: -109	State Akhir: Layer 1 110 92 113 46 118 56 8 63 32 124 109 21 27 47 5 107 120 10 39 91 95 74 16 65 125 Layer 2 52 101 60 51 93 9 18 31 72 100 12 98 53 64 117 38 43 7 102 77 54 40 49 86 90 Layer 3 115 88 14 20 89 70 57 75 37 24 81 61 121 104 11 94 111 26 42 80 58 55 50 112 2 Layer 4 25 15 41 36 105 83 78 84 35 34 17 29 23 123 30 82 44 22 45 67 108 3 6 4 79 Layer 5 13 59 87 71 68 97 103 62 119 85 96 106 66 73 69 76 122 116 114 99 33 1 28 48 19 Nilai Objective Function Akhir: -77

3	 	 

Parameter	Durasi	Jumlah Iterasi	Initial Value	Final Value	Max Iterasi
Percobaan 1	0.482091 detik	1000	-109	-96	1000
Percobaan 2	4.39294 detik	10000	-109	-77	10000
Percobaan 3	21.4581 detik	50000	-109	-73	50000

Berdasarkan hasil percobaan dengan algoritma Stochastic Hill Climbing, tampak bahwa peningkatan jumlah iterasi mengakibatkan peningkatan kualitas solusi yang ada.

Namun, semakin banyak iterasi yang dilakukan, ada satu titik dimana kualitas solusi cenderung stagnan. Pada percobaan pertama, dengan 1000 iterasi dalam waktu 0.482091 detik, nilai akhir yang tercapai adalah -96. Di percobaan kedua, dengan 10000 iterasi dalam durasi 4.39294 detik, nilai akhir meningkat menjadi -77. Sedangkan pada percobaan ketiga, dengan 50000 iterasi yang memakan waktu 21.4581 detik, nilai akhir mencapai -73. Meskipun ada peningkatan yang cukup signifikan pada peningkatan jumlah iterasi awal, namun semakin ditingkatkan jumlah iterasinya, peningkatan kualitas solusi cenderung tidak signifikan. Karen memang algoritm ini cenderung bekeja atau menyelesaikan permasalahan secara lebih "lambat" dibandingkan algoritma lain.

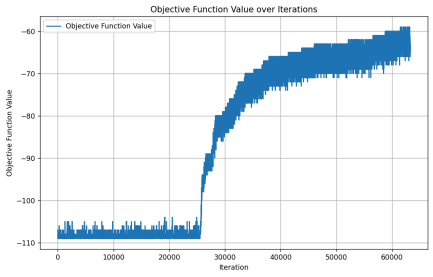
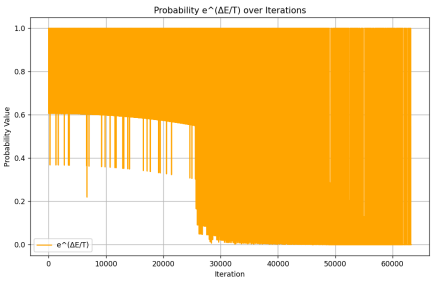
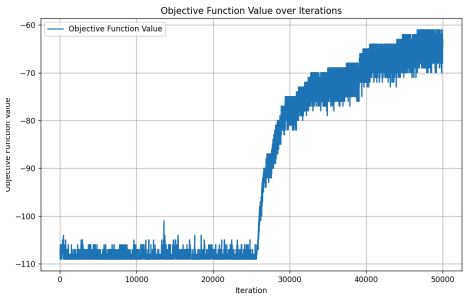
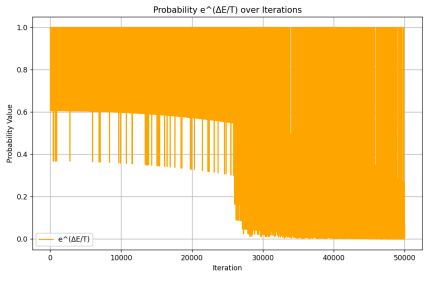
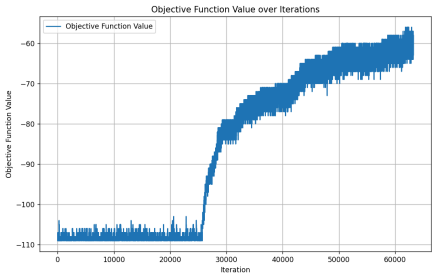
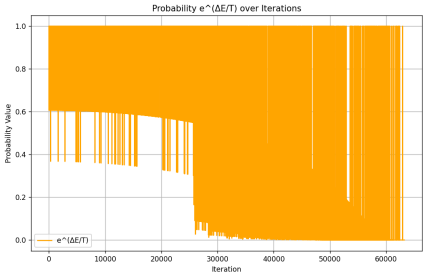
3.1.5. Simulated Annealing

Pada bagian ini, disajikan hasil percobaan dari Simulated Annealing sebanyak tiga kali dengan initial state dan final state sebagai berikut.

Percobaan	Initial State	Final State
1	Initial State: Layer 1 55 70 19 114 81 60 49 86 25 100 35 72 22 2 104 109 85 69 74 5 102 27 34 48 87 Layer 2 13 96 79 64 75 58 113 124 28 30 125 94 38 62 21 51 66 117 41 71 103 47 18 11 12 Layer 3 10 4 68 43 120 110 101 118 32 56 98 90 82 53 112 111 93 105 89 50 8 3 46 97 29 Layer 4 73 40 123 44 99 45 14 31 88 1 23 91 15 116 24 6 7 54 20 76 106 77 122 78 36 Layer 5 57 42 16 84 83 95 80 65 39 92 26 33 63 9 115 17 121 37 107 59 67 61 108 52 119 Initial Value: -107	Total stuck count (local optima): 62389 Final State after Simulated Annealing: Layer 1 25 37 81 11 68 113 93 56 118 65 111 102 61 48 7 3 123 23 83 122 43 103 59 55 53 Layer 2 2 115 24 106 117 114 82 101 9 5 12 19 124 80 85 110 86 46 21 66 77 13 27 99 42 Layer 3 120 87 52 28 35 49 14 20 116 90 32 76 60 22 54 36 26 69 92 107 78 108 15 57 29 Layer 4 34 39 84 79 51 10 109 105 16 98 97 95 47 38 62 94 31 73 121 100 30 41 6 119 4 Layer 5 18 70 45 91 44 112 17 33 1 71 63 75 67 58 8 72 64 74 125 104 50 89 96 40 88 Final Value: -59 Time taken by program is : 29.052000 sec

2	Initial State: Layer 1 82 109 111 94 104 53 19 26 39 66 20 65 59 114 33 1 6 73 44 74 113 119 37 78 51 Layer 2 107 103 79 9 17 75 50 32 106 90 36 124 25 88 10 22 8 60 100 31 62 67 101 105 13 Layer 3 54 21 14 2 91 72 116 99 47 117 86 108 81 27 58 42 123 87 122 71 112 77 18 7 120 Layer 4 5 52 12 45 89 4 61 3 41 57 118 28 49 110 24 38 29 43 40 98 63 15 48 23 115 Layer 5 125 56 55 93 95 121 102 69 35 76 80 85 83 16 92 68 11 70 30 96 64 34 97 46 84 Initial Value: -109	Total stuck count (local optima): 49062 Final State after Simulated Annealing: Layer 1 20 116 91 44 31 53 30 72 77 117 98 17 42 79 70 99 120 107 51 11 45 32 110 64 86 Layer 2 1 123 81 55 59 56 73 54 66 100 108 29 115 95 21 46 112 5 76 85 104 69 96 23 50 Layer 3 125 67 57 33 124 34 3 122 78 89 121 14 94 43 97 7 102 88 27 106 28 38 63 93 82 Layer 4 68 71 26 103 83 10 19 58 49 109 105 15 75 60 9 4 47 90 87 52 24 111 8 16 62 Layer 5 101 6 48 80 12 22 92 36 40 18 41 113 13 2 118 37 39 25 74 61 114 65 84 119 35 Final Value: -61 Time taken by program is : 14.746000 sec
3	Initial State: Layer 1 57 89 122 110 119 125 61 51 109 18 55 28 79 5 47 98 108 3 112 116 65 114 99 46 86 Layer 2 31 82 7 37 8 13 41 42 39 67 35 97 106 96 94 24 100 59 117 101 49 10 124 1 40 Layer 3 66 30 68 113 76 72 85 4 33 48 12 73 107 104 103 121 115 71 26 81 105 53 58 69 19 Layer 4 83 22 78 34 87 84 74 23 17 77 90 16 95 62 120 52 38 123 50 93 14 9 56 43 44 Layer 5 25 91 92 88 60 75 36 45 80 2 6 102 29 54 27 111 64 15 20 118 32 70 21 63 11 Initial Value: -109	Total stuck count (local optima): 62359 Final State after Simulated Annealing: Layer 1 122 29 124 115 77 125 87 67 18 120 2 26 86 113 93 74 69 116 28 10 65 104 24 61 15 Layer 2 110 119 58 106 36 35 59 118 33 76 13 91 83 64 39 95 101 75 22 123 62 20 53 90 41 Layer 3 23 19 5 7 79 32 71 66 73 94 114 49 108 45 30 43 9 31 112 16 103 46 81 78 1 Layer 4 57 98 72 25 27 38 11 52 107 51 40 47 88 70 109 117 99 14 105 21 63 8 89 42 54 Layer 5 3 50 56 80 96 85 34 12 84 60 6 102 97 55 44 100 37 82 48 4 121 92 68 17 111 Final Value: -56 Time taken by program is : 22.383000 sec

Berikut merupakan grafik state value terhadap jumlah iterasi dan plot probabilitas terhadap jumlah iterasi.

Percobaan	Grafik State Value terhadap Jumlah Iterasi	Grafik plot $e^{\Delta E/T}$ terhadap Jumlah Iterasi
1		
2		
3		

Berikut adalah state awal dan akhir dari masing-masing percobaan disertai durasi dan jumlah iterasinya pada algoritma Simulated Annealing ini.

Parameter	Durasi	Jumlah Iterasi Max	Initial Value	Final Value
Percobaan 1	29,052 detik	100.000	-107	-59
Percobaan 2	14,746 detik	50.000	-109	-61
Percobaan 3	22,383 detik	100.000	-109	-56

Berdasarkan hasil percobaan diatas, dapat disimpulkan bahwa semakin besar jumlah iterasi maksimal yang dilakukan maka akan semakin lama durasi eksekusi. Misalnya, pada percobaan 1 dan percobaan 3 dengan 100.000 iterasi, durasinya lebih lama sebesar 29 detik

serta 22 detik dibandingkan dengan percobaan 2 yang memiliki iterasi maksimal 50.000 dengan waktu eksekusi 14,7 detik. Namun, semakin banyak iterasi yang dilakukan, maka final value yang dihasilkan pun semakin baik ditunjukkan saat iterasi maksimalnya bernilai 100.000 menghasilkan final value -56. Nilai tersebut lebih baik saat dibandingkan jika iterasi maksimalnya 50.000 yang menghasilkan final value -61. Hal ini dapat terjadi karena jumlah iterasi yang lebih banyak memberikan algoritma lebih banyak kesempatan untuk menemukan solusi yang lebih optimal.

Berdasarkan hasil percobaan juga didapatkan bahwa hasilnya tidak pernah mencapai global optimum (final value bernilai 0) dikarenakan jumlah iterasi yang dibatasi hanya 50.000 dan 100.00 saja karena memakan waktu yang cukup banyak. Tetapi, solusi yang dihasilkan memiliki final value yang cukup besar yakni -56.

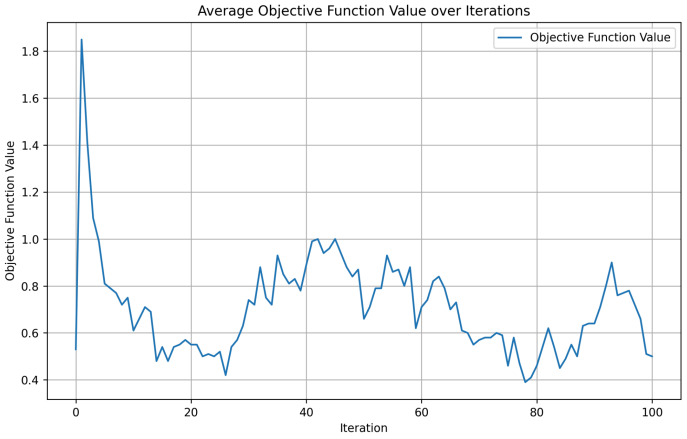
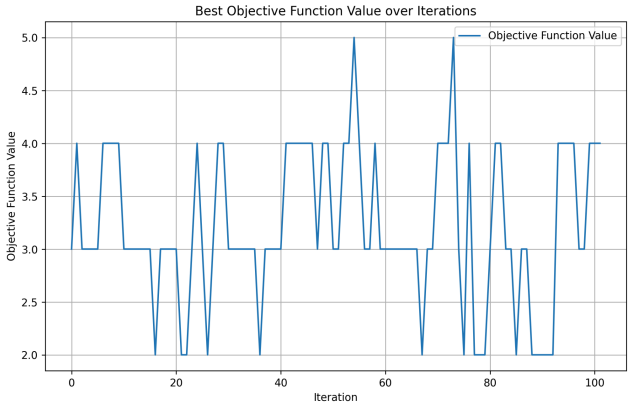
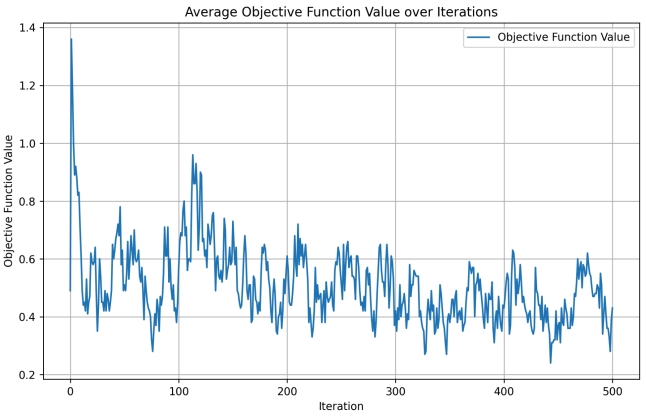
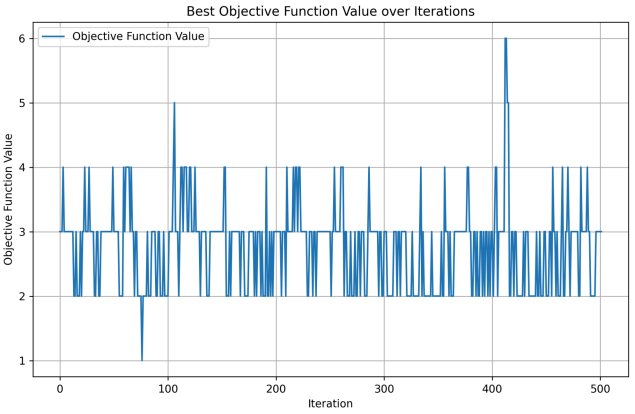
Berdasarkan grafik State Value terhadap Jumlah Iterasi menunjukkan bahwa nilai objective function awalnya hanya berkisaran di sekitar nilai initial value dan tidak mengalami peningkatan signifikan. Namun, setelah 20.000ribu lebih iterasi, nilai objective function mulai meningkat drastis. Hal ini disebabkan oleh penurunan temperatur, yang membuat algoritma semakin selektif dalam menerima solusi yang lebih buruk. Pada akhir proses, nilai objective function cenderung stabil, tetapi masih belum mencapai nilai optimal (0).

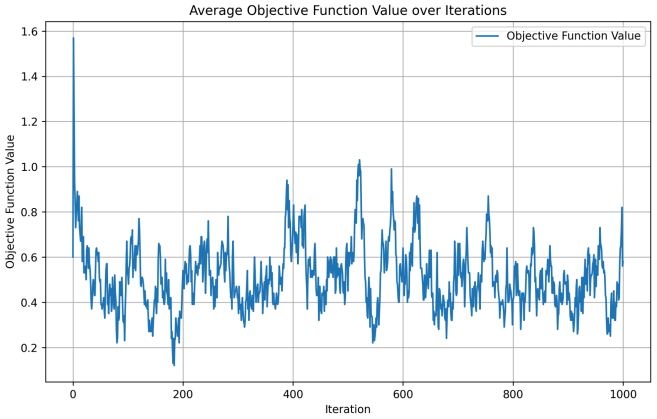
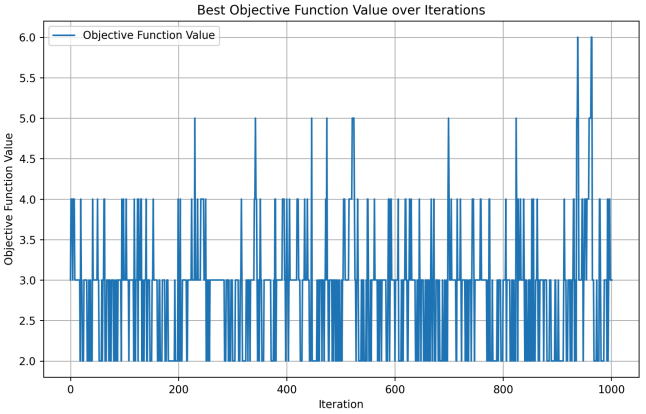
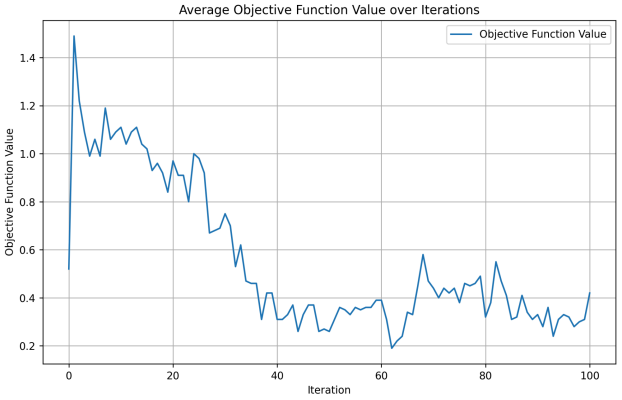
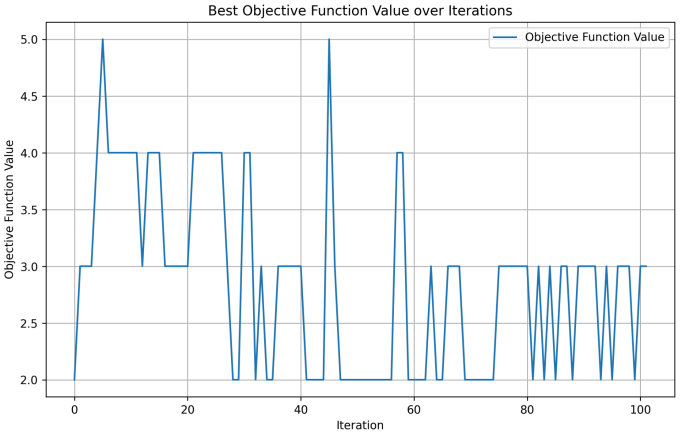
Grafik Probabilitas $e^{\Delta E/T}$ terhadap Jumlah Iterasi menggambarkan penurunan probabilitas penerimaan solusi yang lebih buruk seiring berjalannya waktu. Pada awal iterasi, probabilitas tinggi menunjukkan bahwa algoritma lebih sering menerima solusi yang kurang optimal untuk menjelajahi ruang solusi yang beragam. Namun, seiring menurunnya suhu, probabilitas ini menurun, yang mana menunjukan algoritma semakin fokus pada pencarian solusi yang lebih baik saja.

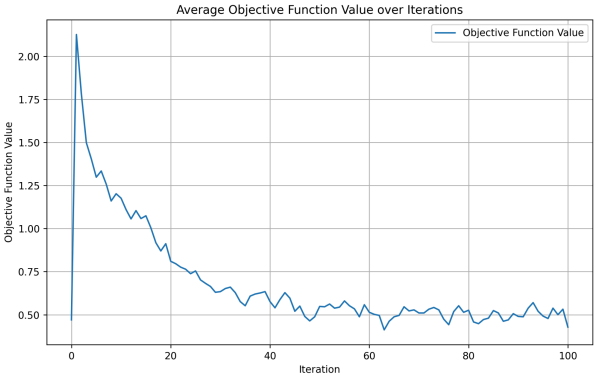
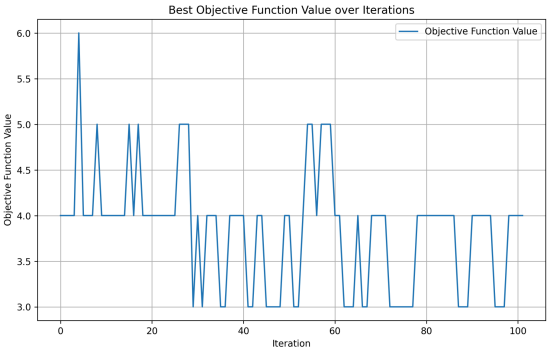
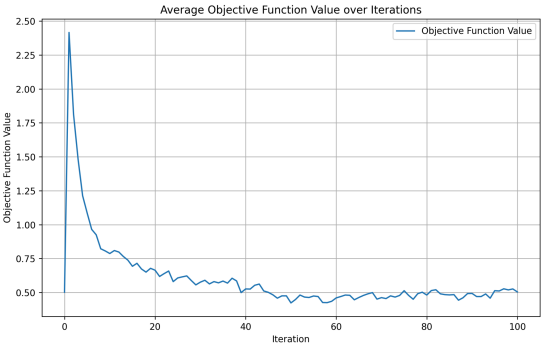
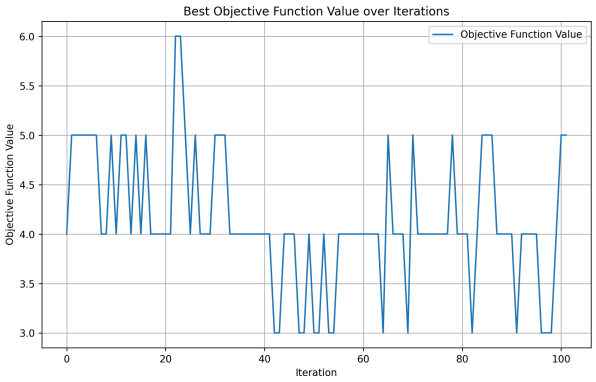
3.1.6. Genetic Algorithm

Pada bagian ini, diterapkan tiga percobaan yang masing-masing memiliki konfigurasi jumlah iterasi maksimum yang berbeda untuk algoritma Genetic Algorithm.

Percobaan	Grafik Average dan Best State Value Terhadap Jumlah Iterasi

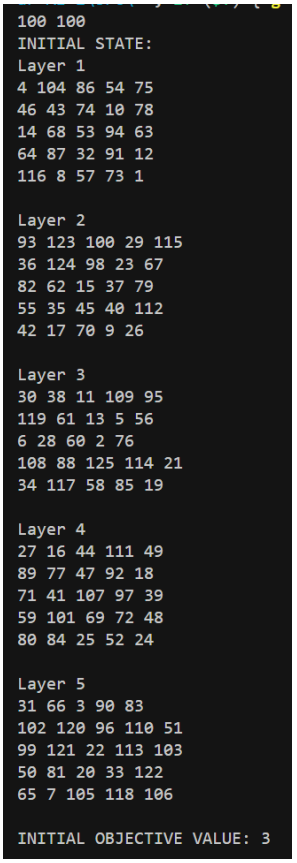
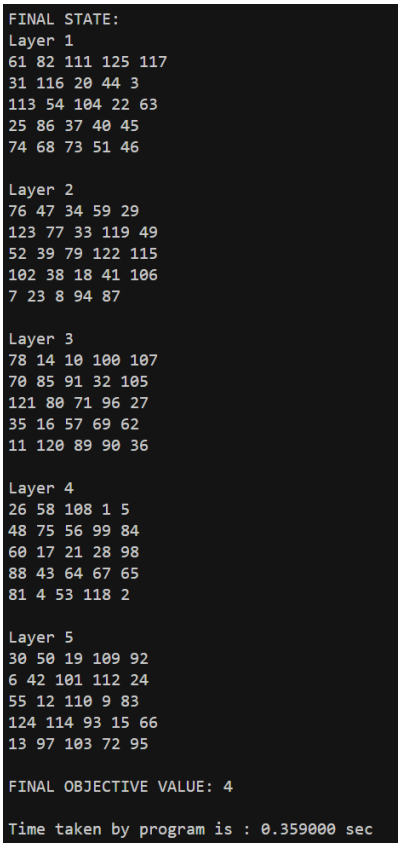
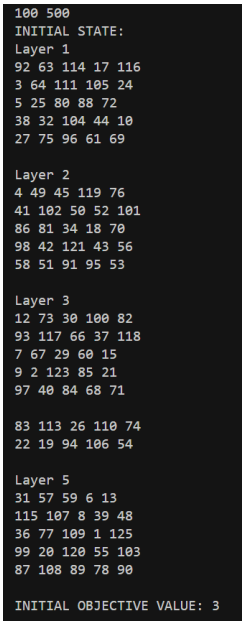
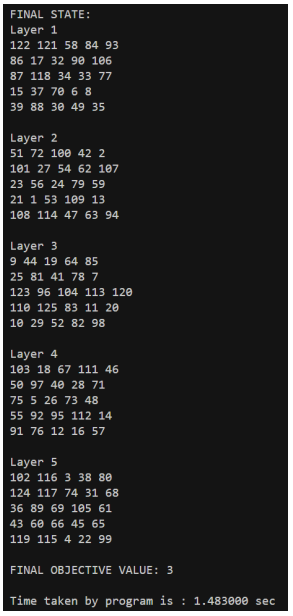
1	 Average Objective Function Value over Iterations Objective Function Value Objective Function Value Iteration  Best Objective Function Value over Iterations Objective Function Value Objective Function Value Iteration
2	 Average Objective Function Value over Iterations Objective Function Value Objective Function Value Iteration  Best Objective Function Value over Iterations Objective Function Value Objective Function Value Iteration

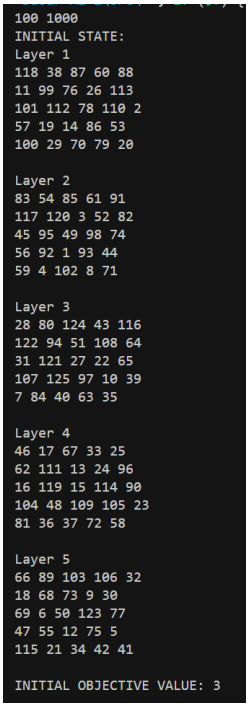
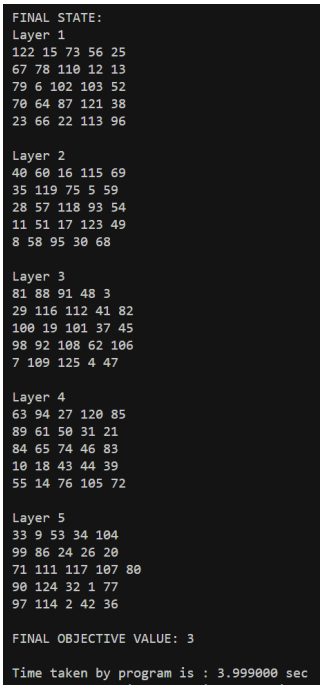
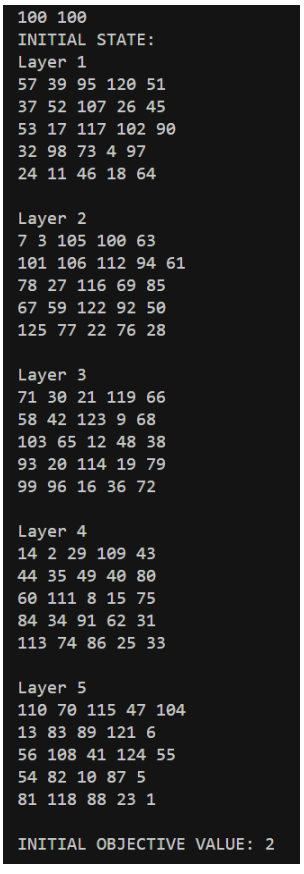
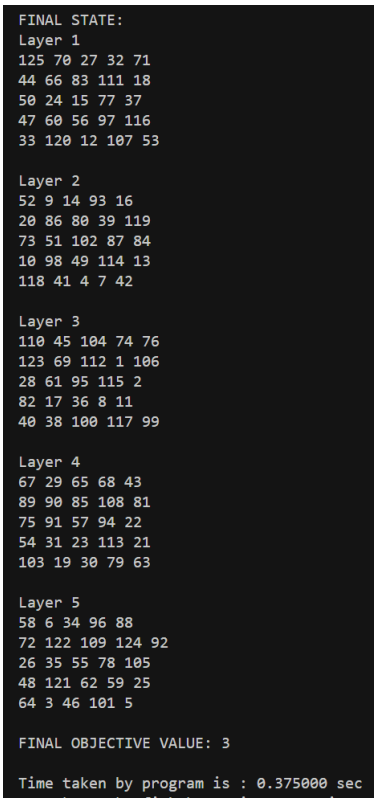
3	 Average Objective Function Value over Iterations Objective Function Value Objective Function Value Iteration  Best Objective Function Value over Iterations Objective Function Value Objective Function Value Iteration
4	 Average Objective Function Value over Iterations Objective Function Value Objective Function Value Iteration  Best Objective Function Value over Iterations Objective Function Value Objective Function Value Iteration

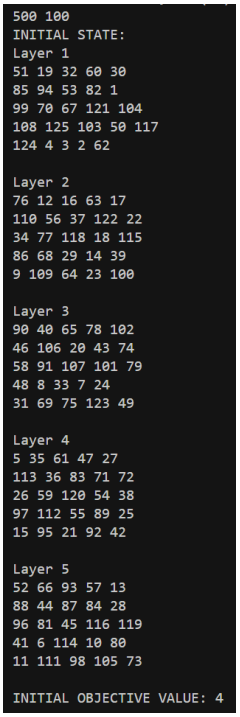
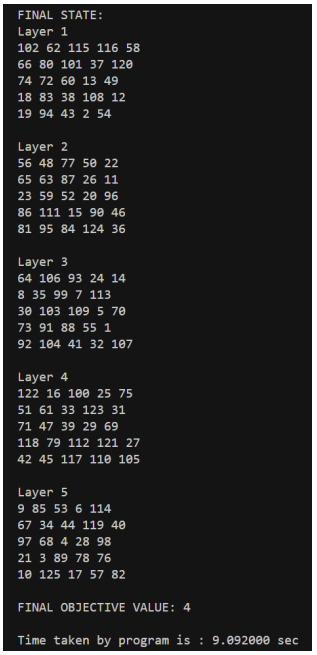
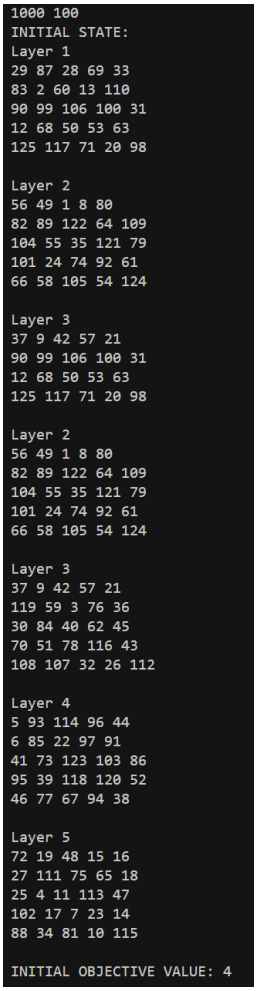
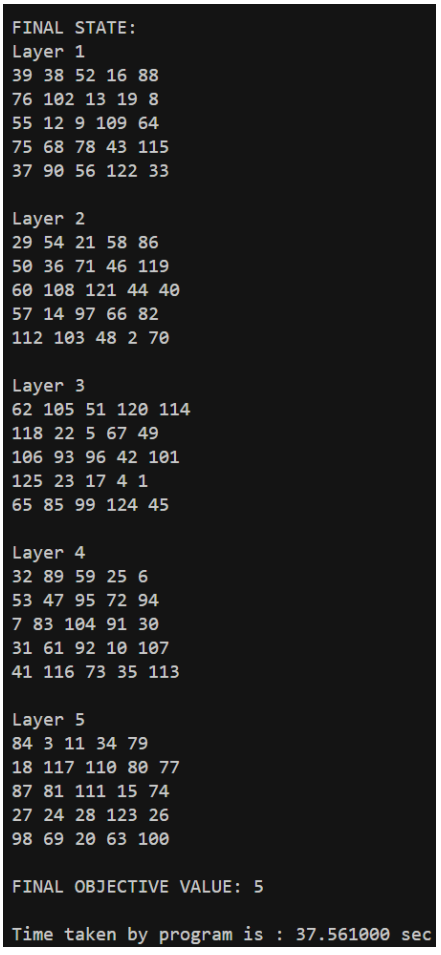
5	 
6	 

Berikut adalah state awal dan akhir dari masing-masing percobaan

Percobaan	**Initial State**	**Final State**

1	 100 100 INITIAL STATE: Layer 1 4 104 86 54 75 46 43 74 10 78 14 68 53 94 63 64 87 32 91 12 116 8 57 73 1 Layer 2 93 123 100 29 115 36 124 98 23 67 82 62 15 37 79 55 35 45 40 112 42 17 70 9 26 Layer 3 30 38 11 109 95 119 61 13 5 56 6 28 60 2 76 108 88 125 114 21 34 117 58 85 19 Layer 4 27 16 44 111 49 89 77 47 92 18 71 41 107 97 39 59 101 69 72 48 80 84 25 52 24 Layer 5 31 66 3 90 83 102 120 96 110 51 99 121 22 113 103 50 81 20 33 122 65 7 105 118 106 INITIAL OBJECTIVE VALUE: 3	 FINAL STATE: Layer 1 61 82 111 125 117 31 116 20 44 3 113 54 104 22 63 25 86 37 40 45 74 68 73 51 46 Layer 2 76 47 34 59 29 123 77 33 119 49 52 39 79 122 115 102 38 18 41 106 7 23 8 94 87 Layer 3 78 14 10 100 107 70 85 91 32 105 121 80 71 96 27 35 16 57 69 62 11 120 89 90 36 Layer 4 26 58 108 1 5 48 75 56 99 84 60 17 21 28 98 88 43 64 67 65 81 4 53 118 2 Layer 5 30 50 19 109 92 6 42 101 112 24 55 12 110 9 83 124 114 93 15 66 13 97 103 72 95 FINAL OBJECTIVE VALUE: 4 Time taken by program is : 0.359000 sec
2	 100 500 INITIAL STATE: Layer 1 92 63 114 17 116 3 64 111 105 24 5 25 80 88 72 38 32 104 44 10 27 75 96 61 69 Layer 2 4 49 45 119 76 41 102 50 52 101 86 81 34 18 70 98 42 121 43 56 58 51 91 95 53 Layer 3 12 73 30 100 82 93 117 66 37 118 7 67 29 60 15 9 2 123 85 21 97 40 84 68 71 83 113 26 110 74 22 19 94 106 54 Layer 5 31 57 59 6 13 115 107 8 39 48 36 77 109 1 125 99 20 120 55 103 87 108 89 78 90 INITIAL OBJECTIVE VALUE: 3	 FINAL STATE: Layer 1 122 121 58 84 93 86 17 32 90 106 87 118 34 33 77 15 37 70 6 8 39 88 30 49 35 Layer 2 51 72 100 42 2 101 27 54 62 107 23 56 24 79 59 21 1 53 109 13 108 114 47 63 94 Layer 3 9 44 19 64 85 25 81 41 78 7 123 96 104 113 120 110 125 83 11 20 10 29 52 82 98 Layer 4 103 18 67 111 46 50 97 40 28 71 75 5 26 73 48 55 92 95 112 14 91 76 12 16 57 Layer 5 102 116 3 38 80 124 117 74 31 68 36 89 69 105 61 43 60 66 45 65 119 115 4 22 99 FINAL OBJECTIVE VALUE: 3 Time taken by program is : 1.483000 sec

3	 100 1000 INITIAL STATE: Layer 1 118 38 87 60 88 11 99 76 26 113 101 112 78 110 2 57 19 14 86 53 100 29 70 79 20 Layer 2 83 54 85 61 91 117 120 3 52 82 45 95 49 98 74 56 92 1 93 44 59 4 102 8 71 Layer 3 28 80 124 43 116 122 94 51 108 64 31 121 27 22 65 107 125 97 10 39 7 84 40 63 35 Layer 4 46 17 67 33 25 62 111 13 24 96 16 119 15 114 90 104 48 109 105 23 81 36 37 72 58 Layer 5 66 89 103 106 32 18 68 73 9 30 69 6 50 123 77 47 55 12 75 5 115 21 34 42 41 INITIAL OBJECTIVE VALUE: 3	 FINAL STATE: Layer 1 122 15 73 56 25 67 78 110 12 13 79 6 102 103 52 70 64 87 121 38 23 66 22 113 96 Layer 2 40 60 16 115 69 35 119 75 5 59 28 57 118 93 54 11 51 17 123 49 8 58 95 30 68 Layer 3 81 88 91 48 3 29 116 112 41 82 100 19 101 37 45 98 92 108 62 106 7 109 125 4 47 Layer 4 63 94 27 120 85 89 61 50 31 21 84 65 74 46 83 10 18 43 44 39 55 14 76 105 72 Layer 5 33 9 53 34 104 99 86 24 26 20 71 111 117 107 80 90 124 32 1 77 97 114 2 42 36 FINAL OBJECTIVE VALUE: 3 Time taken by program is : 3.999000 sec
4	 100 100 INITIAL STATE: Layer 1 57 39 95 120 51 37 52 107 26 45 53 17 117 102 90 32 98 73 4 97 24 11 46 18 64 Layer 2 7 3 105 100 63 101 106 112 94 61 78 27 116 69 85 67 59 122 92 50 125 77 22 76 28 Layer 3 71 30 21 119 66 58 42 123 9 68 103 65 12 48 38 93 20 114 19 79 99 96 16 36 72 Layer 4 14 2 29 109 43 44 35 49 40 80 60 111 8 15 75 84 34 91 62 31 113 74 86 25 33 Layer 5 110 70 115 47 104 13 83 89 121 6 56 108 41 124 55 54 82 10 87 5 81 118 88 23 1 INITIAL OBJECTIVE VALUE: 2	 FINAL STATE: Layer 1 125 70 27 32 71 44 66 83 111 18 50 24 15 77 37 47 60 56 97 116 33 120 12 107 53 Layer 2 52 9 14 93 16 20 86 80 39 119 73 51 102 87 84 10 98 49 114 13 118 41 4 7 42 Layer 3 110 45 104 74 76 123 69 112 1 106 28 61 95 115 2 82 17 36 8 11 40 38 100 117 99 Layer 4 67 29 65 68 43 89 90 85 108 81 75 91 57 94 22 54 31 23 113 21 103 19 30 79 63 Layer 5 58 6 34 96 88 72 122 109 124 92 26 35 55 78 105 48 121 62 59 25 64 3 46 101 5 FINAL OBJECTIVE VALUE: 3 Time taken by program is : 0.375000 sec

5	 500 100 INITIAL STATE: Layer 1 51 19 32 60 30 85 94 53 82 1 99 70 67 121 104 108 125 103 50 117 124 4 3 2 62 Layer 2 76 12 16 63 17 110 56 37 122 22 34 77 118 18 115 86 68 29 14 39 9 109 64 23 100 Layer 3 90 40 65 78 102 46 106 20 43 74 58 91 107 101 79 48 8 33 7 24 31 69 75 123 49 Layer 4 5 35 61 47 27 113 36 83 71 72 26 59 120 54 38 97 112 55 89 25 15 95 21 92 42 Layer 5 52 66 93 57 13 88 44 87 84 28 96 81 45 116 119 41 6 114 10 80 11 111 98 105 73 INITIAL OBJECTIVE VALUE: 4	 FINAL STATE: Layer 1 102 62 115 116 58 66 80 101 37 120 74 72 60 13 49 18 83 38 108 12 19 94 43 2 54 Layer 2 56 48 77 50 22 65 63 87 26 11 23 59 52 20 96 86 111 15 90 46 81 95 84 124 36 Layer 3 64 106 93 24 14 8 35 99 7 113 30 103 109 5 70 73 91 88 55 1 92 104 41 32 107 Layer 4 122 16 100 25 75 51 61 33 123 31 71 47 39 29 69 118 79 112 121 27 42 45 117 110 105 Layer 5 9 85 53 6 114 67 34 44 119 40 97 68 4 28 98 21 3 89 78 76 10 125 17 57 82 FINAL OBJECTIVE VALUE: 4 Time taken by program is : 9.092000 sec
6	 1000 100 INITIAL STATE: Layer 1 29 87 28 69 33 83 2 60 13 110 90 99 106 100 31 12 68 50 53 63 125 117 71 20 98 Layer 2 56 49 1 8 80 82 89 122 64 109 104 55 35 121 79 101 24 74 92 61 66 58 105 54 124 Layer 3 37 9 42 57 21 90 99 106 100 31 12 68 50 53 63 125 117 71 20 98 Layer 2 56 49 1 8 80 82 89 122 64 109 104 55 35 121 79 101 24 74 92 61 66 58 105 54 124 Layer 3 37 9 42 57 21 119 59 3 76 36 30 84 40 62 45 70 51 78 116 43 108 107 32 26 112 Layer 4 5 93 114 96 44 6 85 22 97 91 41 73 123 103 86 95 39 118 120 52 46 77 67 94 38 Layer 5 72 19 48 15 16 27 111 75 65 18 25 4 11 113 47 102 17 7 23 14 88 34 81 10 115 INITIAL OBJECTIVE VALUE: 4	 FINAL STATE: Layer 1 39 38 52 16 88 76 102 13 19 8 55 12 9 109 64 75 68 78 43 115 37 90 56 122 33 Layer 2 29 54 21 58 86 50 36 71 46 119 60 108 121 44 40 57 14 97 66 82 112 103 48 2 70 Layer 3 62 105 51 120 114 118 22 5 67 49 106 93 96 42 101 125 23 17 4 1 65 85 99 124 45 Layer 4 32 89 59 25 6 53 47 95 72 94 7 83 104 91 30 31 61 92 10 107 41 116 73 35 113 Layer 5 84 3 11 34 79 18 117 110 80 77 87 81 111 15 74 27 24 28 123 26 98 69 20 63 100 FINAL OBJECTIVE VALUE: 5 Time taken by program is : 37.561000 sec

Parameter	Percobaan 1	Percobaan 2	Percobaan 3	Percobaan 4	Percobaan 5	Percobaan 6
Durasi (s)	0.359	1.483	3.999	0.375	9.092	37.561
Banyak Populasi	100	100	100	100	500	1000
Jumlah Iterasi	100	500	1000	100	100	100
Initial State Value	3	3	3	2	4	4
Final State Value	4	3	3	3	4	5

Idealnya, meskipun waktu pemrosesan menjadi jauh lebih lama, jumlah populasi dan banyaknya iterasi akan relatif berpengaruh positif terhadap hasil value yang didapat dengan algoritma Genetic Algorithm. Hal ini disebabkan bertambahnya probabilitas menemukan state dengan *value* yang jauh lebih baik di tiap iterasinya. Akan tetapi, pada kode pemrograman yang dibuat, terdapat banyak faktor yang tidak dapat diprediksi dan tidak pasti dengan Genetic Algorithm ini. Dimulai dari *generate initial state* yang memungkinkan banyak terjadi kemungkinan, hingga proses-proses upaya pengoptimalan yang dilakukan.

Proses *selection* dengan metode *tournament* tidak melibatkan seluruh state yang menjadi kandidat untuk pemilihan *value* terbaik. Pemilihan metode ini dilakukan atas dasar ada banyaknya *state* yang memiliki fitness function nol. Hal ini dapat dilihat pada grafik yang menunjukkan bahwa persebaran *state value* populasi di tiap generasi cenderung sangat rendah. Jika melakukan *selection* dengan *roulette wheel,* tentu akan menghilangkan kesempatan sebagian besar state sejak awal di-*generate*.

Kemudian, proses *crossover* yang melakukan *swapping* secara acak dan diikuti mutasi yang dilakukan secara acak juga. Segala bentuk ketidakpastian inilah yang menjadikan Genetic Algorithm tidak begitu efektif dalam menyelesaikan masalah dengan kasus seperti Magic Number ini. Tak seperti algoritma lainnya, Genetic Algorithm cenderung lebih sulit untuk mempertahankan *value* terbaik yang pernah didapat.

3.2. Analisis

Dari hasil implementasi berbagai algoritma local search, tampak bahwa masing-masing algoritma berhenti pada **final state** sebelum mencapai global optima. Hal ini dikarenakan algoritma local search cenderung terperangkap pada local optima. Berikut adalah nilai state terbaik yang dicapai oleh masing-masing algoritma dalam eksperimen yang dilakukan:

Algoritma	State Value Terbaik	Durasi (s)
Hill-Climbing Steepest Ascent	-65	0.297749
Hill-Climbing with Sideways Move	-69	0.595917
Random Restart Hill-Climbing	-68	2.35006
Stochastic Hill Climbing	-73	21.4581
Simulated Annealing	-56	22,383
Genetic Algorithm	5 (-104)	37.561

Performa masing-masing algoritma berbeda karena mereka menggunakan pendekatan yang beragam untuk mencapai global optima.

- Hill Climbing Steepest Ascent terjebak di local maximum karena algoritma berhenti ketika tidak menemukan neighbor yang lebih baik.
- Hill Climbing with Sideways Move sedikit lebih baik dengan mengeksplorasi "flat" area atau plateu (neighbor dengan nilai yang sama), berupaya mencapai shoulder yang bisa mengarah ke global optima.
- Random Restart Hill-Climbing memanfaatkan mekanisme restart, memberikan peluang untuk memulai pencarian dari titik baru dan memperluas area eksplorasi.
- Stochastic Hill Climbing menghasilkan performa terendah karena sangat bergantung pada pemilihan neighbor secara acak, membuatnya kurang prediktif. Walau begitu, karena pemilihan 1 neighbour secara acak, waktu yang dibutuhkan jauh lebih cepat dibandingkan rata-rata algoritma lain (contoh : untuk melakukan 1000 iterasi hanya perlu 0.4 detik)
- Simulated Annealing mencapai nilai terbaik karena kemampuannya untuk menggabungkan random walk dan stochastic hill climbing, memungkinkan pendekatan bertahap menuju solusi global.
- Dibandingkan algoritma lainnya, Genetic Algorithm menghasilkan performa yang kurang optimal dan bahkan paling buruk dalam menyelesaikan permasalahan ini. Hal ini disebabkan banyaknya proses dan elemen yang melibatkan pendekatan *randomness* secara pola pemrosesannya sulit diatur untuk menuju ke *state* dengan *value* lebih baik.

Konsistensi solusi yang ditemukan berbeda-beda di setiao algoritma yang ada, hal ini terutama dikarenakan adanya faktor random yang mempengaruhi hasil pencarian beberapa algoritma. Algoritma yang lebih bergantung pada pendekatan acak (seperti Random Restart Hill-Climbing, Stochastic Hill Climbing, Simulated Annealing, dan Genetic Algorithm) menunjukkan konsistensi

yang lebih rendah dibandingkan yang tidak menggunakan pendekatan acak (seperti Steepest Ascent dan Hill Climbing with Sideways Move).

BAB 4
KESIMPULAN DAN SARAN

4.1. Kesimpulan

Pada tugas ini, kelompok kami melakukan percobaan menggunakan berbagai algoritma *local search* untuk menyelesaikan masalah *magic cube*. Algoritma yang diimplementasikan meliputi beberapa variasi *hill climbing*, *simulated annealing*, dan *genetic algorithm*. Berdasarkan hasil eksperimen, algoritma yang paling mendekati global optima adalah *simulated annealing*. Selain itu, menurut kami, mekanisme kerja dari Simulated Annealing secara keseluruhan paling baik untuk diterapkan pada masalah magic cube ini guna menghasilkan solusi terbaik.

Percobaan yang dilakukan juga memungkinkan kami untuk mengevaluasi konsistensi dari setiap algoritma. Kami menemukan bahwa beberapa algoritma lebih konsisten dalam menemukan solusi, sedangkan yang lain kurang konsisten karena adanya elemen acak (*randomness*) dalam pemilihan *neighbor* atau perpindahan *state*.

Dengan demikian, solusi pencarian untuk *diagonal magic cube* telah ditemukan menggunakan berbagai algoritma *local search*, yaitu *Steepest Ascent Hill-Climbing*, *Hill-Climbing with Sideways Move*, *Random Restart Hill-Climbing*, *Stochastic Hill-Climbing*, *Genetic Algorithm*, dan *Simulated Annealing*. Semua algoritma tersebut diimplementasikan dalam bahasa Python.

4.2. Saran

- Dilakukan optimalisasi program sehingga faktor *random* yang cenderung membawa ke *state* yang lebih buruk dapat dikurangi.
- Menerapkan konsep *concurrency* pada program sehingga waktu eksekusi dapat dioptimalkan
- Memperhatikan durasi waktu pengerjaan sehingga dapat lebih optimal dalam mengatur manajemen waktu pengerjaan tugas
- Mendalami pengetahuan bahasa pemrograman terkait agar lebih mudah dalam mengerjakan tugas

PEMBAGIAN TUGAS

NIM	Nama	Pembagian Tugas
18222008	Abel Apriliani	Implementasi - Algoritma Simulated Annealing - Visualisasi objective function - Visualisasi probabilitas Simulated Annealing Laporan - Deskripsi persoalan - Pembahasan Local Search - Pembahasan Simulated annealing - Hasil dan Analisis Simulated Annealing - Merapikan dokumen
18222036	Olivia Christy L.	Implementasi - Hill Climbing Steepest Ascent - Hill Climbing Sideways Move - Random Restart Hill Climbing - Stochastic Hill Climbing Laporan - Deskripsi Persoalan - Pembahasan (Hill Climbing Steepest Ascent, Hill Climbing Sideways Move, Random Restart Hill Climbing, Stochastic Hill Climbing) - Hasil dan Analisis (Pembahasan (Hill Climbing Steepest Ascent, Hill Climbing Sideways Move, Random Restart

		Hill Climbing, Stochastic Hill Climbing, Analisis Keseluruhan)
18222044	Khansa Adilla Reva	Algoritma - Membuat Genetic Algorithm - Membuat Local Search - Membantu mendebug dan merevisi Simulated Annealing Laporan - Deskripsi Persoalan - Pembahasan Local Search - Kesimpulan - Saran
18222062	Nafisha Virgin	Algoritma - Membuat Genetic Algorithm Laporan - Pembahasan Genetic Algorithm - Analisa dan Hasil Genetic Algorithm - Saran - Kesimpulan - Deskirpsi Persoalan

REFERENSI

Magic Square. https://en.wikipedia.org/wiki/Magic_square

Magic Cube. https://en.wikipedia.org/wiki/Magic_cube

Erdiwansyah E. (2016). Analisis Perbandingan Metode Local Search dan Population Based Dalam Algoritma Berevolusi untuk Penyelesaian Travelling Salesman Problem (TSP). https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://ojs.serambimekkah.ac.id/jse/article/view/307/287&ved=2ahUKEwjFnKfY7u2IAxV-2DgGHawzO7UQFnoECBQQAQ&usg=AOvVaw2ofPqFlAz70fqP-bHL6OfH

S. Russell dan P. Norvig (2010). *Artificial Intelligence A Modern Approach* Edisi Ketiga.

Sneha Kothari (2023). *Local Search Algorithms in AI: A Comprehensive Guide*. https://www.simplilearn.com/local-search-algorithms-in-ai-article

Rob Womersley (2008). *Local and Global Optimization Formulation, Methods and Applications.* http://web.maths.unsw.edu.au/~rsw/lgopt.pdf

JC Ang (2014). *Global search vs local search.* https://www.researchgate.net/post/May-I-know-what-is-the-difference-of-global-and-local-search-in-term-of-machine-learning-and-computer-science

Global Search Method. https://bookdown.org/max/FES/global.html

Quora (2023). *Why do algorithms become entrapped in local optima?* https://www.quora.com/Why-do-algorithms-become-entrapped-in-local-optima

Quora (2020). *What is a complete search algorithm?* http://www.quora.com/What-is-a-complete-search-algorithm

Slide Kuliah IF3070 - Dasar Inteligensi Artificial 2024.

Magic Square | Urutan Ganjil. https://www.geeksforgeeks.org/magic-square/

David (2012). *Magic Cube*. https://github.com/davidwhogg/MagicCube/blob/master/code/cube.py

GeeksForGeeks. 2024. https://www.geeksforgeeks.org/genetic-algorithms/.

Viescinski, Amanda. 2024. *Tournament Selection in Genetic Algorithms*. https://www.baeldung.com/cs/ga-tournament-selection#:~:text=Tournament%20Selection%20is%20a%20popular%20method%20for%20choosing%20individuals%20from,the%20algorithm%20and%20understand%20why.