

Tugas Besar 1 IF3170 Inteligensi Artifisial
Semester I tahun 2024/2025
Pencarian Solusi Diagonal Magic Cube dengan Local Search



Oleh:

Hugo Sabam Augusto (13522129)

Muhammad Zaki (13522136)

Samy Muhammad Haikal (13522151)

Muhammad Roihan (13522152)

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2024

DAFTAR ISI

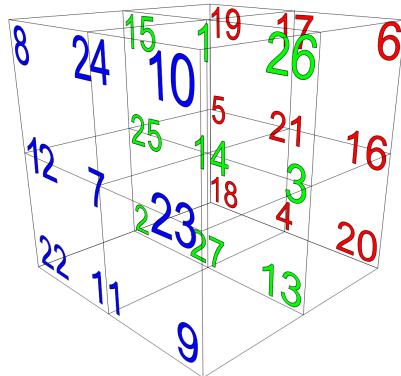
DAFTAR ISI	2
DESKRIPSI PERSOALAN	4
PEMBAHASAN	6
1.1 Pemilihan Objective Function	6
1.2 Penjelasan Implementasi Algoritma Local Search	8
1.2.1 Algoritma Local Search (Server Side)	8
Tabel 1.2.1.1 : Kelas Cube	8
1.2.2 Representasi Magic Cube (Client Side)	22
Tabel 1.2.2.1 : Kelas MagicCube	22
Tabel 1.2.2.2 : Kelas HillClimb	24
Tabel 1.2.2.3 : Kelas SimulatedAnnealing	25
Tabel 1.2.2.4 : Kelas GeneticAlgorithm	25
1.3 Hasil Eksperimen dan Analisis	26
1.3.1 Steepest Ascent Hill-Climbing	26
1.3.1.1 Percobaan 1	26
1.3.1.2 Percobaan 2	28
1.3.1.3 Percobaan 3	30
1.3.2 Stochastic Hill-Climbing	33
1.3.2.1 Percobaan 1	33
1.3.2.2 Percobaan 2	35
1.3.2.3 Percobaan 3	37
1.3.3 Random Restart Hill-Climbing	41
1.3.3.1 Percobaan 1	41
1.3.3.2 Percobaan 2	43
1.3.3.3 Percobaan 3	45
1.3.4 Sideways Move	48
1.3.4.1 Percobaan 1	48
1.3.4.2 Percobaan 2	49
1.3.4.3 Percobaan 3	51
1.3.5 Simulated Annealing	54
1.3.5.1 Percobaan 1	54
1.3.5.2 Percobaan 2	56
1.3.5.3 Percobaan 3	59
1.3.5.4 Percobaan 4	62
1.3.5.5 Percobaan 5	65
1.3.6 Genetic Algorithm (Iterasi beda, Populasi tetap)	67

1.3.6.1 Percobaan 1	67
1.3.6.2 Percobaan 2	70
1.3.6.3 Percobaan 3	72
(Populasi beda,Iterasi tetap)	74
1.3.6.4 Percobaan 4	74
1.3.6.5 Percobaan 5	76
1.3.6.6 Percobaan 6	78
1.3.7 Analisis	81
KESIMPULAN	84
PEMBAGIAN TUGAS	85
REFERENSI	86
LAMPIRAN	87

DESKRIPSI PERSOALAN

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

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



- Terdapat 9 potongan bidang, yaitu:

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

Pada tugas ini, peserta kuliah akan menyelesaikan permasalahan Diagonal Magic Cube berukuran 5x5x5 dengan local search. Initial state dari suatu kubus adalah susunan angka 1 hingga 5^3 secara acak. Kemudian, tiap iterasi pada algoritma local search, langkah yang boleh dilakukan adalah menukar posisi dari 2 angka pada kubus tersebut (2 angka yang ditukar tidak harus bersebelahan).

PEMBAHASAN

1.1 Pemilihan Objective Function

Objective function yang dirancang harus mengukur seberapa "dekat" susunan angka dalam kubus dengan mencapai kondisi ideal di mana semua baris, kolom, tiang, dan diagonal memiliki jumlah yang sama dengan magic number (Berdasarkan referensi magic number untuk kasus ini adalah 315) Atau bisa didapatkan dalam rumus berikut

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

(Sumber :https://en.wikipedia.org/wiki/Magic_cube)

Objective function yang kami gunakan adalah negatif dari total selisih absolut antara angka dalam baris , kolom , tiang dan diagonal dari magic number. Semakin kecil total selisih, semakin baik susunan angka dalam kubus. Nilai maksimum dari objective function ini adalah 0, yaitu ketika semua baris, kolom, tiang, dan diagonal memiliki jumlah yang sama dengan magic number. Alasan penggunaan objective function ini adalah karena menurut kami total selisih absolut memberikan ukuran kesalahan yang lebih linear dan sederhana untuk dinilai, memungkinkan perbaikan bertahap dalam proses iteratif.

Berikut ilustrasi perhitungan objective function kami

Level I				Level II				Level III				Level IV			
a1	a2	a3	a4	a1	a2	a3	a4	a1	a2	a3	a4	a1	a2	a3	a4
b1	b2	b3	b4	b1	b2	b3	b4	b1	b2	b3	b4	b1	b2	b3	b4
c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4
d1	d2	d3	d4	d1	d2	d3	d4	d1	d2	d3	d4	d1	d2	d3	d4

(Sumber :<https://www.magischvierkant.com/three-dimensional-eng/magic-features/>)

1. Σ dari jumlah dalam satu baris di tiap layer, $a1+a2+a3+a4$ dst..(Untuk tiap layer I,II,III,IV) yang diselisihkan dengan dengan magic number serta di absolut agar positif

2. \sum dari jumlah dalam satu kolom di tiap layer, $a_1+b_1+c_1+d_1$ dst..(untuk tiap layer elemen I,II,III,IV) yang diselisihkan dengan dengan magic number serta di absolut agar positif
3. \sum dari jumlah elemen pilar melalui layer ($I\ a_1 + II\ a_1 + III\ a_1 + IV\ a_1 + \dots + I\ d_4 + II\ d_4 + III\ d_4 + IV\ d_4$) yang diselisihkan dengan dengan magic number serta di absolut agar positif
4. \sum dari jumlah diagonal utama dari tiap layer I,II,III,IV ($a_1+b_2+c_3+d_4$) dan ($a_4+b_3+c_2+d_1$) yang diselisihkan dengan magic number serta di absolut
5. \sum dari jumlah diagonal depan ke belakang sisi di KEEMPAT layer ($I\ a_1 + II\ a_2 + III\ a_3 + IV\ a_4$) hingga ($I\ d_1 + II\ d_2 + III\ d_3 + IV\ d_4$) + ($I\ a_4 + II\ a_3 + III\ a_2 + IV\ a_1$) hingga ($I\ d_4 + II\ d_3 + III\ d_2 + IV\ d_1$) yang diselisihkan dengan magic number serta di absolut
6. \sum dari jumlah diagonal kiri ke kanan sisi di KEEMPAT layer ($I\ a_1 + II\ b_1 + III\ c_1 + IV\ d_1$) hingga ($I\ a_4 + II\ b_4 + III\ c_4 + IV\ d_4$) + ($I\ d_1 + II\ c_1 + III\ b_1 + IV\ a_1$) hingga ($I\ d_4 + II\ c_4 + III\ b_4 + IV\ a_4$) yang diselisihkan dengan magic number serta di absolut
7. \sum dari jumlah diagonal ruang yang sudah diselisihkan dengan magic number serta di absolut

$$h = | \sum_{i=0} F(i) - M |$$

{i adalah semua baris/kolom/diagonal/pilar yang telah dibahas di atas, M adalah magic number}

*untuk plotting nanti, nilai h dibuat minus untuk alasan representasi grafik saja

1.2 Penjelasan Implementasi Algoritma Local Search

Untuk implementasi algoritma local search, kami menggunakan bahasa **JavaScript** di bagian server side dengan framework **Express** dan menggunakan library **THREE.js** serta **Chart.js** untuk representasi magic cube dan plotting di bagian client side

1.2.1 Algoritma Local Search (Server Side)

Kelas ini akan merepresentasikan kubus 5x5x5 dan berfungsi sebagai basis untuk menyimpan state kubus serta melakukan operasi-operasi seperti menghitung fitness, melakukan swap angka, menghitung objective function, dan semua algoritma yang akan digunakan nanti

Tabel 1.2.1.1 : Kelas Cube

CLASS		
	Class Cube	Kelas utama dalam tugas ini, berfungsi sebagai controller segala jenis algoritma
ATTRIBUTE		
	<pre>constructor(size, cubeState) { this.n = size; this.magicNumber = (this.n * (Math.pow(this.n, 3) + 1)) / 2; this.cube = cubeState; this.h = 0; this.iterasi = 0; this.maxIterasi = 12000; //buat stochastic this.hValues = []; this.avgHValues = []; this.sequensElement = []; this.hValues = []; this.e_values = []; this.stuck_freq = 0; }</pre>	
METHOD		
1	objectiveFunction() → Integer	Kelas ini menghitung nilai objective function (h) sesuai dengan penjelasan di poin 1.1
	<pre>objectiveFunction() {</pre>	

```

this.h = 0;

// Cek setiap baris di setiap level
for (let i = 0; i < this.n; i++) {
    for (let j = 0; j < this.n; j++) {
        let rowSum = 0;
        let colSum = 0;
        for (let k = 0; k < this.n; k++) {
            rowSum += this.cube[i][j][k];
            colSum += this.cube[i][k][j];
        }
        this.h += Math.abs(rowSum - this.magicNumber);
        this.h += Math.abs(colSum - this.magicNumber);
    }
}

// Cek setiap pilar (melalui level)
for (let j = 0; j < this.n; j++) {
    for (let k = 0; k < this.n; k++) {
        let pillarSum = 0;
        for (let i = 0; i < this.n; i++) {
            pillarSum += this.cube[i][j][k];
        }
        this.h += Math.abs(pillarSum - this.magicNumber);
    }
}

// Cek diagonal di setiap level (2 diagonal per level)
for (let i = 0; i < this.n; i++) {
    let diag1Sum = 0;
    let diag2Sum = 0;
    for (let j = 0; j < this.n; j++) {
        diag1Sum += this.cube[i][j][j];
        diag2Sum += this.cube[i][j][this.n - j - 1];
    }
    this.h += Math.abs(diag1Sum - this.magicNumber);
    this.h += Math.abs(diag2Sum - this.magicNumber);
}

// Cek diagonal vertikal (melalui level)
for (let j = 0; j < this.n; j++) {
    let vertDiag1 = 0;
    let vertDiag2 = 0;
    for (let i = 0; i < this.n; i++) {
        vertDiag1 += this.cube[i][i][j];
        vertDiag2 += this.cube[i][this.n - i - 1][j];
    }
    this.h += Math.abs(vertDiag1 - this.magicNumber);
    this.h += Math.abs(vertDiag2 - this.magicNumber);
}

```


		<pre> improved = true; this.sequensElement.push([[i, j, k], [x, y, z], [this.cube[x][y][z], this.cube[i][j][k]],currentH]); this.hValues.push(currentH * (-1)); this.iterasi++; } else { // newH >= currentH //balikin [this.cube[i][j][k], this.cube[x][y][z]] = [this.cube[x][y][z], this.cube[i][j][k]]; // Selesai 1 iterasi, simpan nilai h } } } } } } } while (improved); return this.cube; </pre>
3	stochasticHillClimbing() → cubeState	Fungsi ini membangkitkan 2 elemen dengan posisi acak dan tukar kedua elemen. Apabila 2 elemen ini ketika ditukar menghasilkan objective function yang lebih baik, maka akan dihitung 1 iterasi dan dijadikan currentH, bila tidak kembalikan dan lakukan ulang hingga this.maxIterasi.
		<pre> let currentH = this.objectiveFunction(); for (let i = 0; i < this.maxIterasi; i++) { let pos1 = Math.floor(Math.random() * (this.n * this.n * this.n)); let pos2; do { pos2 = Math.floor(Math.random() * (this.n * this.n * this.n)); } while (pos1 === pos2); const x1 = Math.floor(pos1 / (this.n * this.n)); const y1 = Math.floor((pos1 % (this.n * this.n)) / this.n); const z1 = pos1 % this.n; const x2 = Math.floor(pos2 / (this.n * this.n)); </pre>

		<pre> const y2 = Math.floor((pos2 % (this.n * this.n)) / this.n); const z2 = pos2 % this.n; [this.cube[x1][y1][z1], this.cube[x2][y2][z2]] = [this.cube[x2][y2][z2], this.cube[x1][y1][z1]]; const newH = this.objectiveFunction(); if (newH < currentH) { currentH = newH; this.sequensElement.push([[i, j, k], [x, y, z], [this.cube[x][y][z], this.cube[i][j][k]],currentH]); this.hValues.push(currentH * (-1)); this.iterasi++; } else { [this.cube[x1][y1][z1], this.cube[x2][y2][z2]] = [this.cube[x2][y2][z2], this.cube[x1][y1][z1]]; } } return this.cube; </pre>
4	simulatedAnnealing() → cubeState	Fungsi ini membangkitkan 2 sel dengan posisi acak. Apabila 2 sel yang dibangkitkan adalah sel yang berbeda, kita akan mengobservasi fungsi objektifnya. Jika fungsi objektif saat kedua sel ditukar lebih besar dari fungsi objektif saat ini, maka kedua sel akan ditukar. Jika tidak, probabilitas penukarannya dihitung berdasarkan fungsi $e^{\frac{\Delta H}{T}}$, dengan $\Delta H = H_0 - H_1$ dan T adalah nilai yang semakin lama semakin turun. Jika nilai fungsi ini lebih besar dari 0.5 maka akan dilakukan penukaran walaupun fungsi objektifnya lebih rendah. Jika tidak, maka skip dan lanjut ke iterasi selanjutnya, lakukan sebanyak maxIterations kali.
		<pre> let currentH = this.objectiveFunction(); let Tvalue = 30000; const coolingRate = 0.99; const maxIterations = 12000; do { let i = Math.floor(Math.random() * 5); let j = Math.floor(Math.random() * 5); let k = Math.floor(Math.random() * 5); let x = Math.floor(Math.random() * 5); let y = Math.floor(Math.random() * 5); </pre>

	<pre> let z = Math.floor(Math.random() * 5); if (i !== x j !== y k !== z) { [this.cube[i][j][k], this.cube[x][y][z]] = [this.cube[x][y][z], this.cube[i][j][k]]; let newH = this.objectiveFunction(); let diffH = currentH - newH; const e_prob = Math.exp(diffH/Tvalue); if (newH < currentH) { currentH = newH; this.sequensElement.push([[i, j, k], [x, y, z], [this.cube[x][y][z], this.cube[i][j][k]],currentH]); } else if(Math.exp(diffH/Tvalue)>0.5){ currentH = newH; this.sequensElement.push([[i, j, k], [x, y, z], [this.cube[x][y][z], this.cube[i][j][k]],currentH]); this.stuck_freq++; }else{ [this.cube[i][j][k], this.cube[x][y][z]] = [this.cube[x][y][z], this.cube[i][j][k]]; } this.iterasi++; this.hValues.push(currentH * (-1)); if(diffH>0){ this.e_values.push(0); }else{ this.e_values.push(Math.exp((diffH)/Tvalue)); } Tvalue *= coolingRate; } } while (this.iterasi < maxIterations Tvalue > 1e-10); return this.cube; </pre>
5	randomRestartHillClimbing(maxRestart) → cubeState <p>Fungsi ini digunakan untuk menyelesaikan permasalahan diagonal magic cube dengan menggunakan algoritma random restart, algoritma hill climb sama seperti dengan steepest ascent akan tetapi dilakukan sebanyak maxRestart dengan cube state awal yang berbeda beda. Fungsi ini akan mengembalikan cube dengan nilai h yang paling optimal diantara cube state yang telah dicoba</p>

	<pre> let best_h = this.getObjective(); let best_cube = this.cube; let tempIterasi = 0; console.log(maxRestart); for (let i = 0; i < maxRestart; i++) { const newCube = new Cube(this.n, this.generateMagicCubeState(this.n)); newCube.steepestAscentHillClimbing(); tempIterasi += newCube.getIterasi(); let currentH = newCube.getObjective(); this.hValues.push(currentH); if (currentH < best_h) { best_h = currentH; best_cube = JSON.parse(JSON.stringify(newCube.cube)); console.log(`Iteration \${i}: Current H = \${currentH}, Best H = \${best_h}`); } } this.iterasi = tempIterasi; this.cube = best_cube; return this.cube; </pre>
6	<p>generateMagicCubeState(n)</p> <p>Fungsi ini digunakan untuk men generate magic cube dengan nilai random yang memenuhi aturan magic number</p> <pre> const totalNumbers = n * n * n; const numbers = Array.from({ length: totalNumbers }, (_, i) => i + 1); for (let i = numbers.length - 1; i > 0; i--) { const j = Math.floor(Math.random() * (i + 1)); [numbers[i], numbers[j]] = [numbers[j], numbers[i]]; } const cubeState = Array.from({ length: n }, () => Array.from({ length: n }, () => Array(n).fill(0))); let index = 0; for (let i = 0; i < n; i++) { </pre>

		<pre> for (let j = 0; j < n; j++) { for (let k = 0; k < n; k++) { cubeState[i][j][k] = numbers[index++]; } } return cubeState; </pre>
7	sidewaysmoveHillClimbing(maxSidewaysMoves)	Fungsi ini digunakan untuk menyelesaikan permasalahan diagonal magic cube dengan algoritma sideways move, algoritma ini memungkinkan untuk menerima nilai h yang sama seperti sebelumnya akan tetapi (sideways move) akan tetapi dibatasi jumlahnya sebanyak maxSidewaysMove dimana jika sudah mencapai batas tersebut akan mengembalikan cube state saat ini
		<pre> let currentH = this.objectiveFunction(); let improved; let sidewaysMoves = 0; console.log(maxSidewaysMoves); do { improved = false; for (let i = 0; i < this.n; i++) { for (let j = 0; j < this.n; j++) { for (let k = 0; k < this.n; k++) { for (let x = 0; x < this.n; x++) { for (let y = 0; y < this.n; y++) { for (let z = 0; z < this.n; z++) { if (i !== x j !== y k !== z) { // Swap elements [this.cube[i][j][k], this.cube[x][y][z]] = [this.cube[x][y][z], this.cube[i][j][k]]; let newH = this.objectiveFunction(); if (newH < currentH) { currentH = newH; improved = true; sidewaysMoves = 0; //Reset counter this.sequensElement.push([[i, j, k], [x, y, z], </pre>

	<pre> [this(cube[x][y][z], this(cube[i][j][k]),currentH)); this.hValues.push(currentH * (-1)); this.iterasi++; } else if (newH === currentH && sidewaysMoves < maxSidewaysMoves) { currentH = newH; sidewaysMoves++; this.sequensElement.push([[i, j, k], [x, y, z], [this(cube[x][y][z], this(cube[i][j][k]),currentH)); this.hValues.push(currentH * (-1)); this.iterasi++; } else { [this(cube[i][j][k], this(cube[x][y][z])] = [this(cube[x][y][z], this(cube[i][j][k]))); } } if (sidewaysMoves >= maxSidewaysMoves) return this(cube; } } } } } } } while (improved); return this(cube; </pre>
8	geneticAlgorithm(populationSize, iterations) Fungsi ini digunakan untuk menyelesaikan magic cube menggunakan algoritma genetic algorithm
	<pre> // Initial Population let population = this.initializePopulation(populationSize); const maxIterations = iterations; do { // Fitness Function let fitnessScores = population.map(cubeState => this.calculateFitness(cubeState)); </pre>

```

let totalFitness = fitnessScores.reduce((sum, score) => sum + score, 0);
let averageFitness = totalFitness / fitnessScores.length;
this.avgHValues.push(averageFitness*(-1));

// Check if we found a solution
let bestFitness = Math.min(...fitnessScores);
this.hValues.push(bestFitness * (-1));

let bestIndex = fitnessScores.indexOf(bestFitness);
if (bestFitness === 0) {
    // Solution found
    this(cube = population[bestIndex];
    return this(cube;
}

// Selection
// Lakukan seleksi untuk mendapatkan parent
let matingPool = this.selection(population, fitnessScores);

// Crossover - Generate new population
let newPopulation = [];
for (let i = 0; i < population.length-1; i+=2){
    let parent1 = matingPool[i];
    let parent2 = matingPool[i + 1];
    let child = this.crossover(parent1, parent2);
    newPopulation.push(child[0]);
    newPopulation.push(child[1]);
}
// kalo ada parent yang gaada pasangan, dijadiin generasi baru
if (matingPool.length % 2 !== 0) {
    newPopulation.push(matingPool[matingPool.length - 1]);
}

// Mutation
let mutationRate = 0.1;
newPopulation = newPopulation.map(individual =>
    Math.random() < mutationRate ? this.mutate(individual) : individual
);

population = newPopulation;
this.iterasi++;
}while (this.iterasi < maxIterations);

// Return best solution found
let finalFitnessScores = population.map(cubeState =>
this.calculateFitness(cubeState));

```

		<pre>let bestFinalIndex = finalFitnessScores.indexOf(Math.min(...finalFitnessScores)); this.cube = population[bestFinalIndex]; return this.cube;</pre>
9	initializePopulation(size)	Fungsi ini digunakan untuk membuat initial population
		<pre>// Create an array of random cube states const population = []; for (let i = 0; i < size; i++) { let cubeCopy = JSON.parse(JSON.stringify(this.cube)); cubeCopy = this.randomizeCube(cubeCopy); population.push(cubeCopy); } return population;</pre>
10	calculateFitness(cubeState)	Fungsi ini digunakan untuk menghitung nilai fitness suatu populasi
		<pre>const originalCube = this.cube; this.cube = cubeState; const fitness = this.objectiveFunction(); this.cube = originalCube; return fitness;</pre>
11	selection(population,fitnessValues)	Fungsi ini digunakan untuk melakukan selection pada populasi menggunakan algoritma random wheel, dimana populasi yang memiliki nilai paling baik akan memiliki probabilitas terbesar
		<pre>// Seleksi menggunakan random wheel // Fitness score yang semakin dekat dengan 0 probabilitynya semakin besar const minFitness = Math.min(...fitnessValues); const adjustedFitnessValues = fitnessValues.map(fitness => fitness - minFitness + 1); const totalAdjustedFitness = adjustedFitnessValues.reduce((sum, fitness) => sum + fitness, 0);</pre>

		<pre>// Calculate cumulative probabilities const cumulativeProbabilities = []; let cumulativeSum = 0; for (let i = 0; i < adjustedFitnessValues.length; i++) { cumulativeSum += adjustedFitnessValues[i] / totalAdjustedFitness; cumulativeProbabilities[i] = cumulativeSum; } const matingPool = []; for (let i = 0; i < population.length; i++){ // Spin the wheel const randomValue = Math.random(); // Find the individual where cumulative probability is just greater than randomValue for (let i = 0; i < population.length; i++) { if (randomValue <= cumulativeProbabilities[i]) { matingPool.push(population[i]); } } } return matingPool;</pre>
12	crossover(parent1, parent2)	Fungsi ini digunakan untuk melakukan crossover pada 2 buah individu yang akan menghasilkan 2 buah child dengan crossoverpoint random
		<pre>const child1 = JSON.parse(JSON.stringify(parent1)); const child2 = JSON.parse(JSON.stringify(parent2)); const crossoverPoint = Math.floor(Math.random() * 5); for (let i = crossoverPoint; i < 5; i++) { for (let j = 0; j < 5; j++) { for (let k = 0; k < 5; k++) { child1[i][j][k] = parent2[i][j][k]; child2[i][j][k] = parent1[i][j][k]; } } } return [child1, child2];</pre>

13	mutate(cubeState)	Fungsi ini digunakan untuk melakukan mutasi pada sebuah individu
		<pre>// Swap two random elements in the cube let i1 = Math.floor(Math.random() * 5); let j1 = Math.floor(Math.random() * 5); let k1 = Math.floor(Math.random() * 5); let i2 = Math.floor(Math.random() * 5); let j2 = Math.floor(Math.random() * 5); let k2 = Math.floor(Math.random() * 5); [cubeState[i1][j1][k1], cubeState[i2][j2][k2]] = [cubeState[i2][j2][k2], cubeState[i1][j1][k1]]; return cubeState;</pre>
14	randomizeCube(cubeState)	Fungsi ini merupakan fungsi helper untuk membantu melakukan inisialisasi populasi
		<pre>while (true) { // tukar 2 angka let i = Math.floor(Math.random() * 5); let j = Math.floor(Math.random() * 5); let k = Math.floor(Math.random() * 5); let x = Math.floor(Math.random() * 5); let y = Math.floor(Math.random() * 5); let z = Math.floor(Math.random() * 5); if (i !== x j !== y k !== z) { [cubeState[i][j][k], cubeState[x][y][z]] = [cubeState[x][y][z], cubeState[i][j][k]]; return cubeState; } } return cubeState;</pre>

1.2.2 Representasi Magic Cube (Client Side)

Kelas ini akan merepresentasikan kubus 5x5x5 dalam bentuk 3D pada web dengan state yang sudah diolah di server side. Kelas ini mayoritas akan *handle* request dari server dan diolah ke client dengan berbagai method dan operasi. Pada bagian ini semua method yang berhubungan dengan memperbaiki tampilan kubus tidak dituliskan secara detail (pada bagian source code), hanya yang berhubungan dengan logika representasi kubus 3D. Client side akan me-generate initial state dan kirim initial state ke server side, lalu diolah dan dikembalikan ke client side untuk representasi.

Tabel 1.2.2.1 : Kelas MagicCube

CLASS	
Class MagicCube	Parent Class untuk kelas GeneticAlgorithm, HillClimb dan SimmulatedAnnealing
ATTRIBUTE DAN CONSTRUCTOR	
//Bagian Scene this.scene = null; this.camera = null; this.renderer = null; this.replayscene = null; this.replaycamera = null; this.replayrenderer = null; this.solvedScene = null; this.solvedCamera = null; this.solvedRenderer = null; this.controls = null; this.solvedControls = null; this.replaycontrols = null;	//Bagian Logic this.isSolved = false; this(cubeState = []; this.solvedCubeState = []; this.lastRenderTime = 0; this.space = 1; this.nsize = 5; this.x_offset = 1; this.y_offset = 1; this.z_offset = 1; this.nonDeletableObjects = []; this.animationProgress = 0; this.startPosition = new THREE.Vector3(1000, 1000, 1000); this.sequensElement = []; this.replayDurasi = 1000; this.speedUpDurasi = this.replayDurasi; this.nsequence = 0; this.cubePositions = {}; this.isReplayed = false;

		<pre>this.hValues = null; this.avghvalues = null; this.initializeReplayCubeScene(); this.initializeScene(); this.initializeSolvedCubeScene();</pre>
METHOD		
1	initializeScene()	Method ini berguna untuk me-render canvas cube state awal
2	initializeSolvedCubeScene()	Method ini berguna untuk me-render canvas cube state yang sudah di solve
3	initializeReplayCubeScene()	Method ini berguna untuk me-render canvas cube yang akan di-replay
4	animate()	Me-request frame animasi yang sudah disediakan library THREE.js <i>requestAnimationFrame(this.animate.bind(this));</i>
5	animateCubeMovement()	Method ini akan menggunakan memvisualisasikan sekuens dari pertukaran 2 elemen ketika algoritma dijalankan. Variabel yang digunakan yaitu this.SequensElement pada tabel 1.2.1.1
6	setCenterPivotControl(targetControl, targetCamera, targetRenderer)	Method ini mengontrol titik tengah / acuan (pivot) di tengah kubus 3D
7	createTransparentCube(number)	Method yang membuat instansi 1 buah elemen dibungkus dalam 1 cube transparan dengan sebuah sprite angka di dalamnya. Method ini akan digunakan didalam method createCube()
8	createCube(cubeState, targetScene, targetcontrol)	Method menerima parameter state cube, scene dan pivot canvas lalu dibuat instansinya di canvas 3D

9	visualizeCube(cubeState, targetScene, targetControl)	Prosedur yang menganimasikan cube dengan fungsi createCube dan animate() sesuai parameter canvas yang dituju
10	generateInitialState(n)	Method yang membuat state awal cube. State cube ini yang akan dikirim ke server side untuk diproses

Tabel 1.2.2.2 : Kelas HillClimb

CLASS		
Class HillClimb extends MagicCube		Child Class dari kelas MagicCube. Kelas ini berfokus untuk mengirim request untuk algoritma Steepest Ascent, Sideway, Random Restart dan Stochastic. Class ini juga menerima request dari server.
METHOD		
1	solveSteepHC(cubeState)	Menerima parameter cubeState dari method generateInitialState(n) dari kelas parent ke server dan menerima hasil berupa solveState dengan algoritma Steepest Ascent
2	solveSidewayHC(cubeState,maxsidewaysMove)	Menerima parameter cubeState dan maksimal gerakan sideway dari input dan menerima hasil berupa solveState dengan algoritma Sideway Move
3	solveRandomRestartHC(cubeState,maxRestarts)	Menerima parameter cubeState dan maksimal gerakan restart dari input dan menerima hasil berupa solveState dengan algoritma Random Restart
4	solveStochasticHC(cubeState)	Menerima parameter cubeState dari method generateInitialState(n) dari kelas parent ke server dan menerima hasil berupa solveState dengan algoritma Stochastic

Tabel 1.2.2.3 : Kelas SimulatedAnnealing

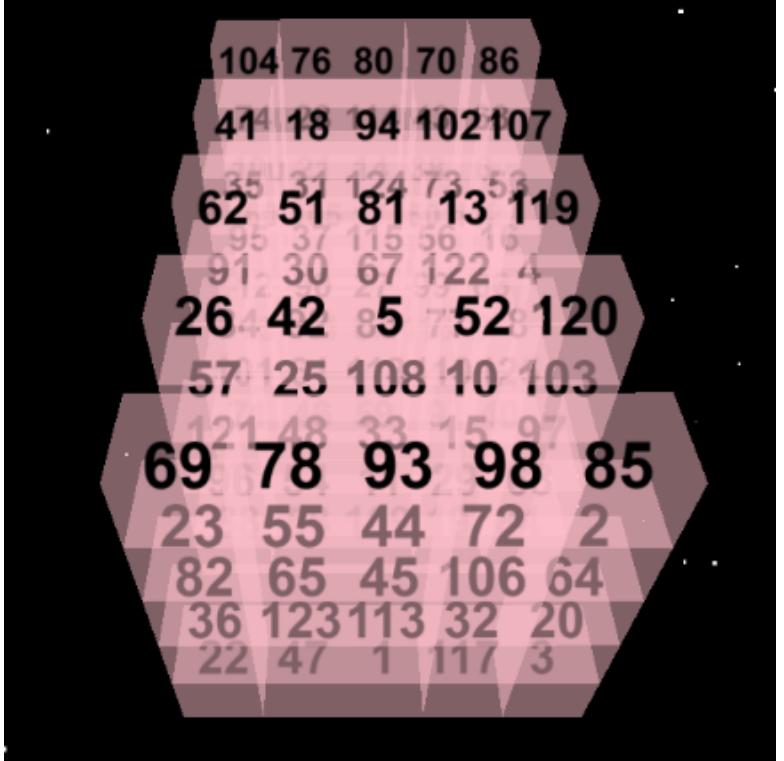
CLASS		
Class SimulatedAnnealing extends MagicCube		Child Class dari kelas MagicCube. Kelas ini berfokus untuk mengirim request untuk algoritma Simulated Annealing. Class ini juga menerima request dari server.
METHOD		
1	solveSA(cubeState)	Menerima parameter cubeState dari method generateInitialState(n) dari kelas parent ke server dan menerima hasil berupa solveState dengan algoritma Simulated Annealing

Tabel 1.2.2.4 : Kelas GeneticAlgorithm

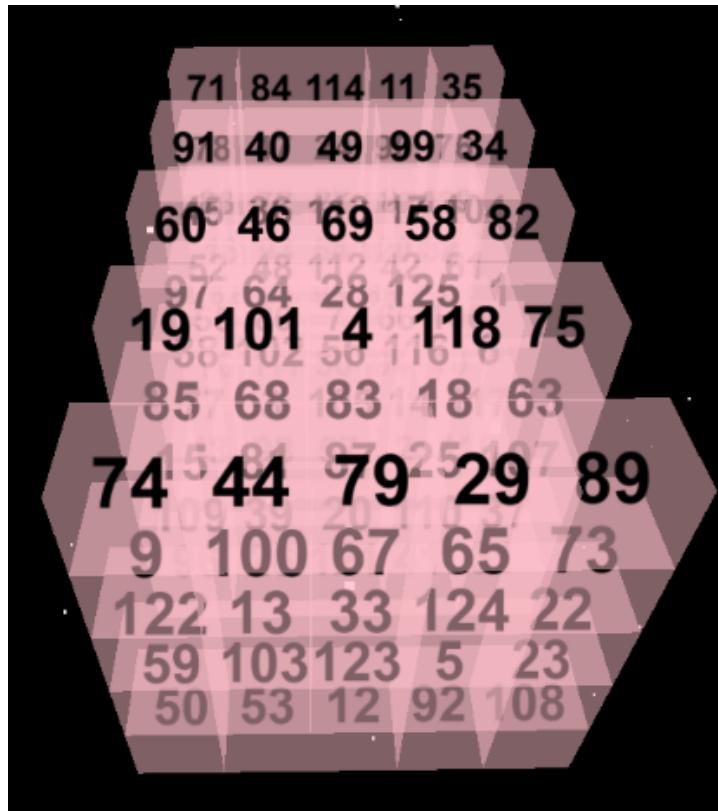
CLASS		
Class GeneticAlgorithm extends MagicCube		Child Class dari kelas MagicCube. Kelas ini berfokus untuk mengirim request untuk algoritma Simulated Annealing. Class ini juga menerima request dari server.
METHOD		
1	solveGA(cubeState)	Menerima parameter cubeState dari method generateInitialState(n) dari kelas parent ke server dan menerima hasil berupa solveState dengan algoritma Genetic Algorithm

1.3 Hasil Eksperimen dan Analisis

1.3.1 Steepest Ascent Hill-Climbing

1.3.1.1 Percobaan 1																																																			
Initial State	 <table border="1"><tr><td>104</td><td>76</td><td>80</td><td>70</td><td>86</td></tr><tr><td>41</td><td>18</td><td>94</td><td>102</td><td>107</td></tr><tr><td>62</td><td>51</td><td>81</td><td>13</td><td>119</td></tr><tr><td>26</td><td>42</td><td>5</td><td>52</td><td>120</td></tr><tr><td>57</td><td>25</td><td>108</td><td>10</td><td>103</td></tr><tr><td>69</td><td>78</td><td>93</td><td>98</td><td>85</td></tr><tr><td>23</td><td>55</td><td>44</td><td>72</td><td>2</td></tr><tr><td>82</td><td>65</td><td>45</td><td>106</td><td>64</td></tr><tr><td>36</td><td>123</td><td>113</td><td>32</td><td>20</td></tr><tr><td>22</td><td>47</td><td>1</td><td>117</td><td>3</td></tr></table>	104	76	80	70	86	41	18	94	102	107	62	51	81	13	119	26	42	5	52	120	57	25	108	10	103	69	78	93	98	85	23	55	44	72	2	82	65	45	106	64	36	123	113	32	20	22	47	1	117	3
104	76	80	70	86																																															
41	18	94	102	107																																															
62	51	81	13	119																																															
26	42	5	52	120																																															
57	25	108	10	103																																															
69	78	93	98	85																																															
23	55	44	72	2																																															
82	65	45	106	64																																															
36	123	113	32	20																																															
22	47	1	117	3																																															

Solved State



Algorithm: Steepest
Hill Climb

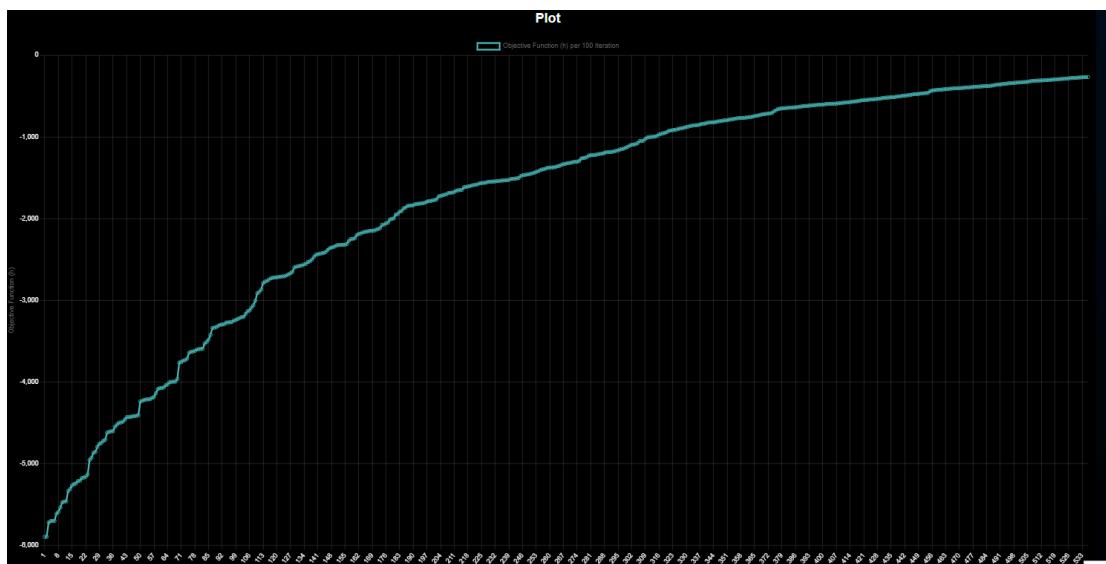
H Before: 5944

H After: 266

Iterations: 536

Waktu Eksekusi: 205
ms

Plot

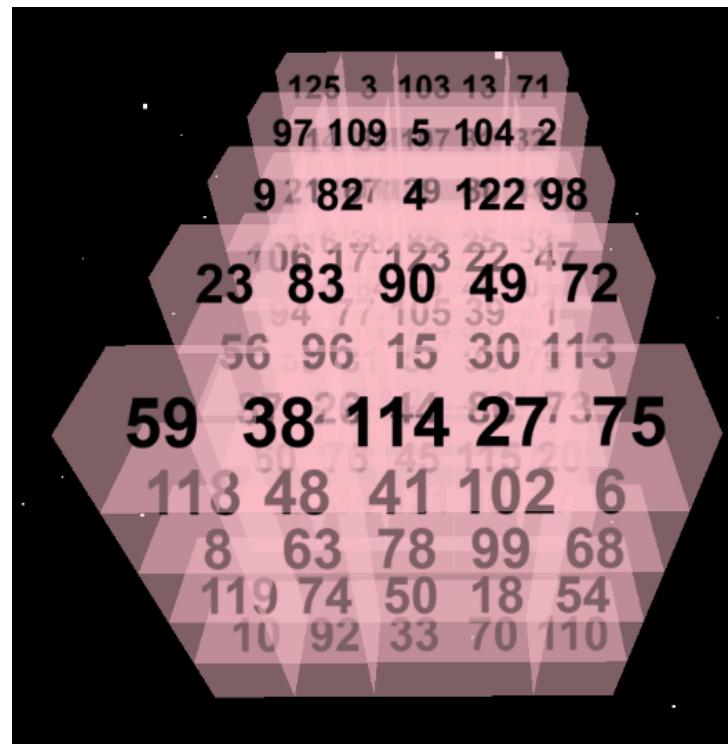


1.3.1.2 Percobaan 2

Initial State

68 3 5 18 2
118 75 37 76 6
8 35 56 5 7 51 70 1 99
29 98 89 13 122
53 36 83 14 59
57 25 24 96 4
12 43 103 115 69
95 49 44 20 107
9 50 67 120 104

Solved State



Algorithm: Steepest
Hill Climb

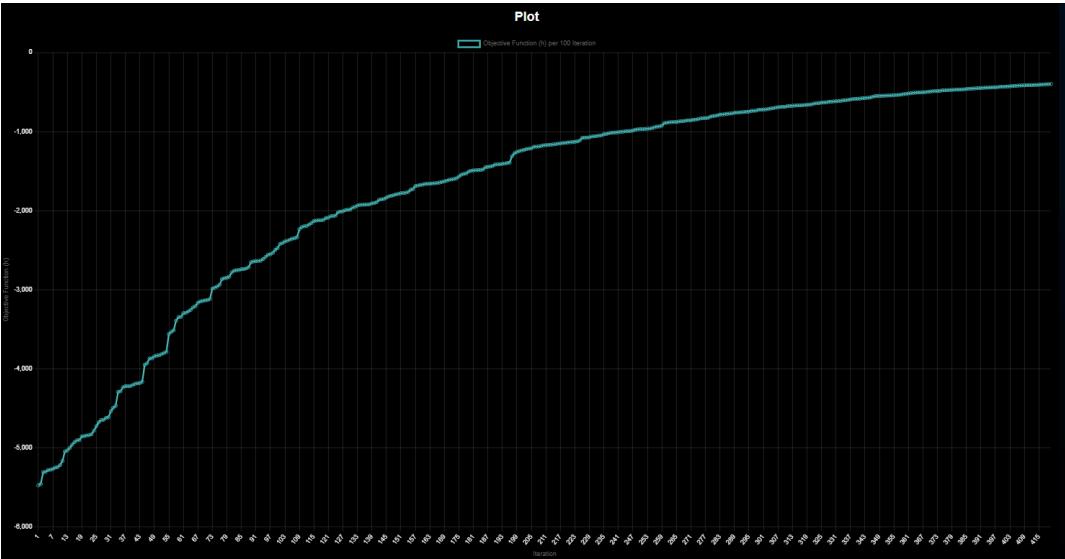
H Before: 5653

H After: 399

Iterations: 420

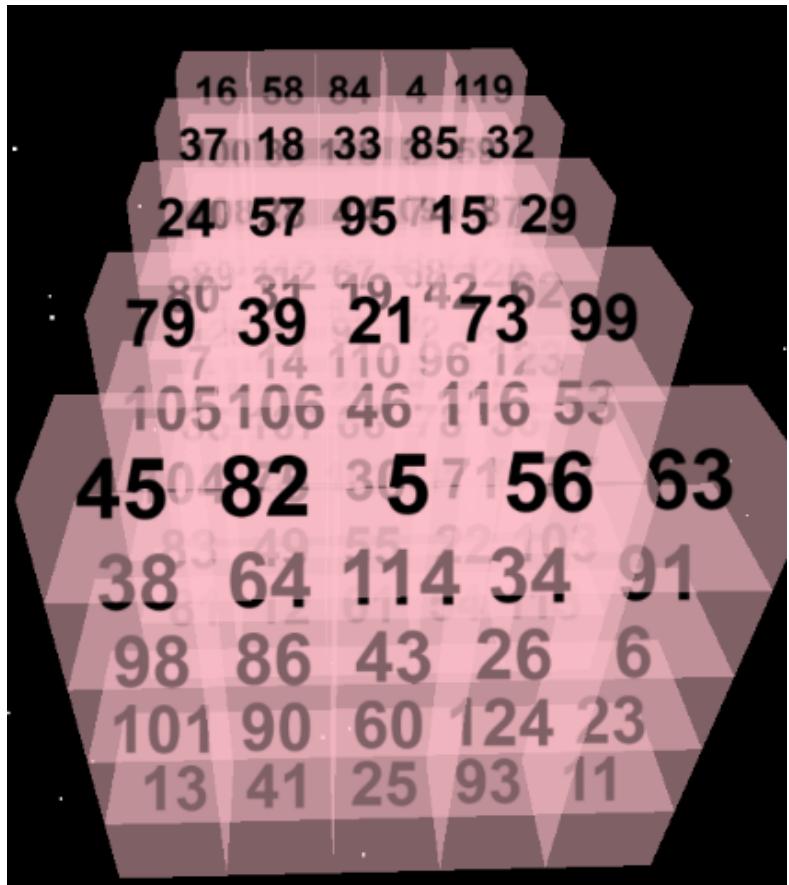
Waktu Eksekusi: 114
ms

Plot

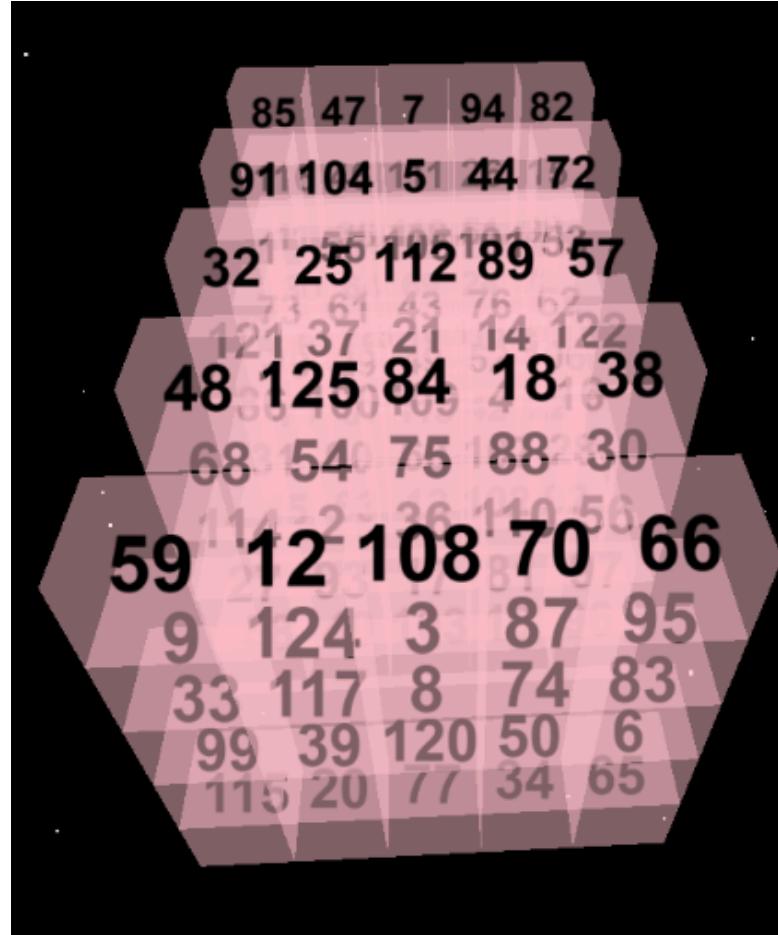


1.3.1.3 Percobaan 3

Initial State



Solved State



Algorithm: Steepest
Hill Climb

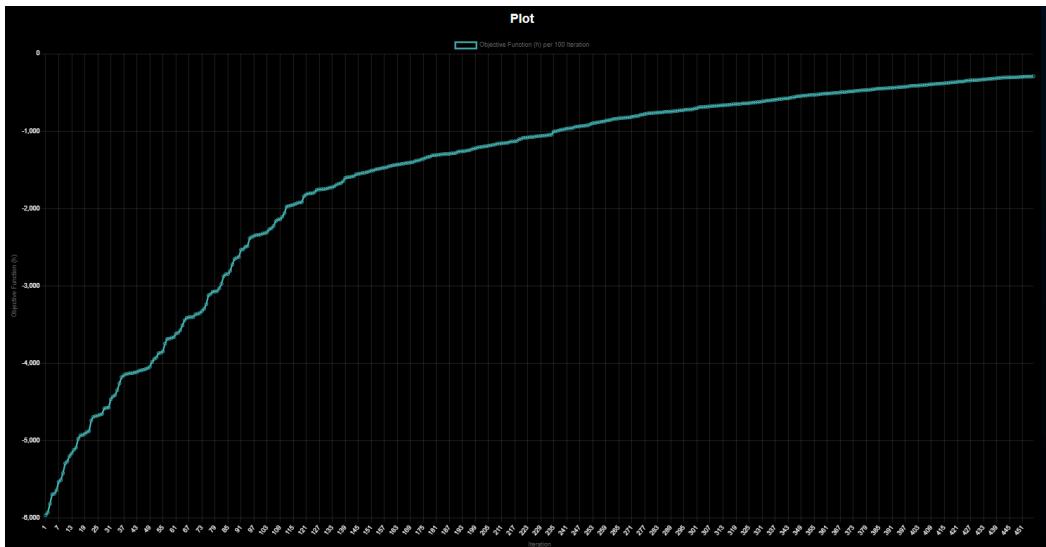
H Before: 5974

H After: 291

Iterations: 456

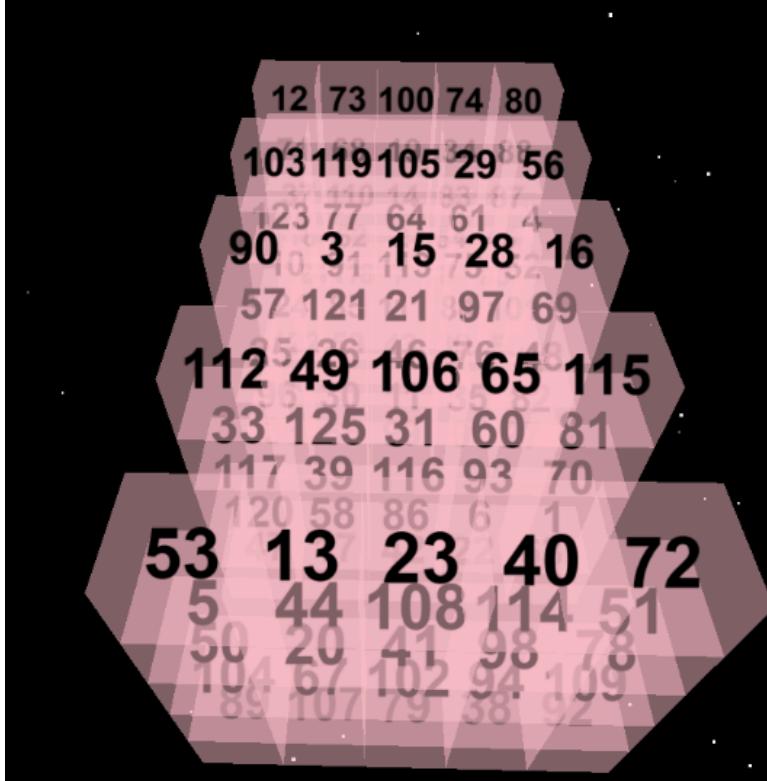
Waktu Eksekusi: 128
ms

Plot

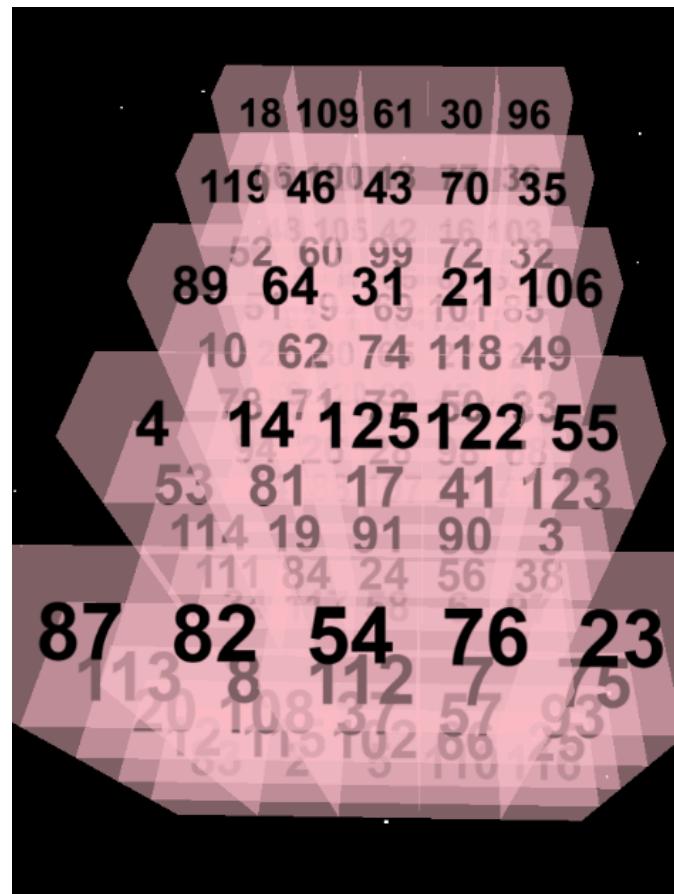


1.3.2 Stochastic Hill-Climbing

Untuk Percobaan HC menggunakan Stochastic, digunakan initial state yang sama untuk ketiga percobaan agar mempermudah perbandingan dan analisis. Selain itu, NMAX iterasi sebesar [12000](#)

1.3.2.1 Percobaan 1	
Initial State	 <p>The image shows a 15x15 grid of numbers from 1 to 225. A subset of these numbers is highlighted in bold black, while others are in a lighter gray. The highlighted numbers form a pattern: 112, 49, 106, 65, 115 in the top row; 53, 13, 23, 40, 72 in the second row; and 12, 73, 100, 74, 80 in the third row. This suggests a specific starting configuration for a search algorithm like Stochastic Hill-Climbing.</p>

Solved State



Algorithm: Stochastic
Hill Climb

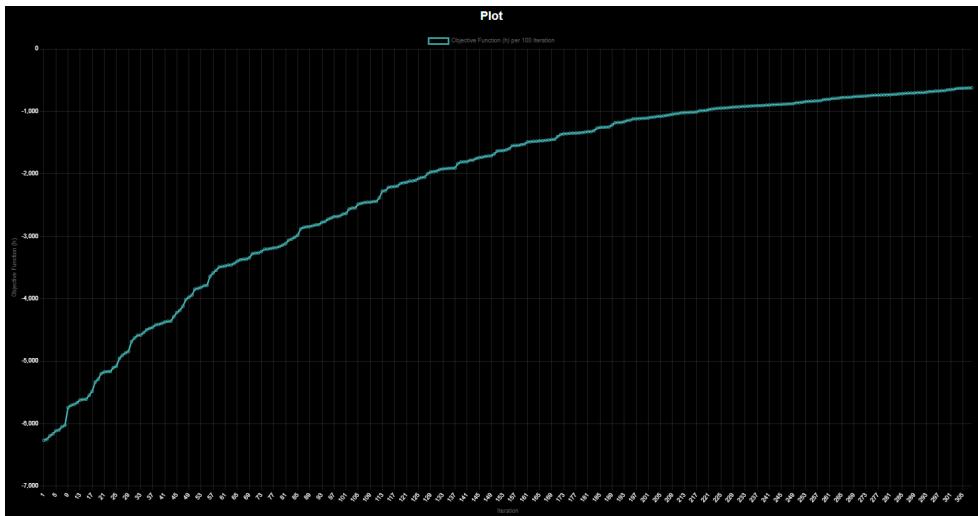
H Before: 6274

H After: 623

Iterations: 308

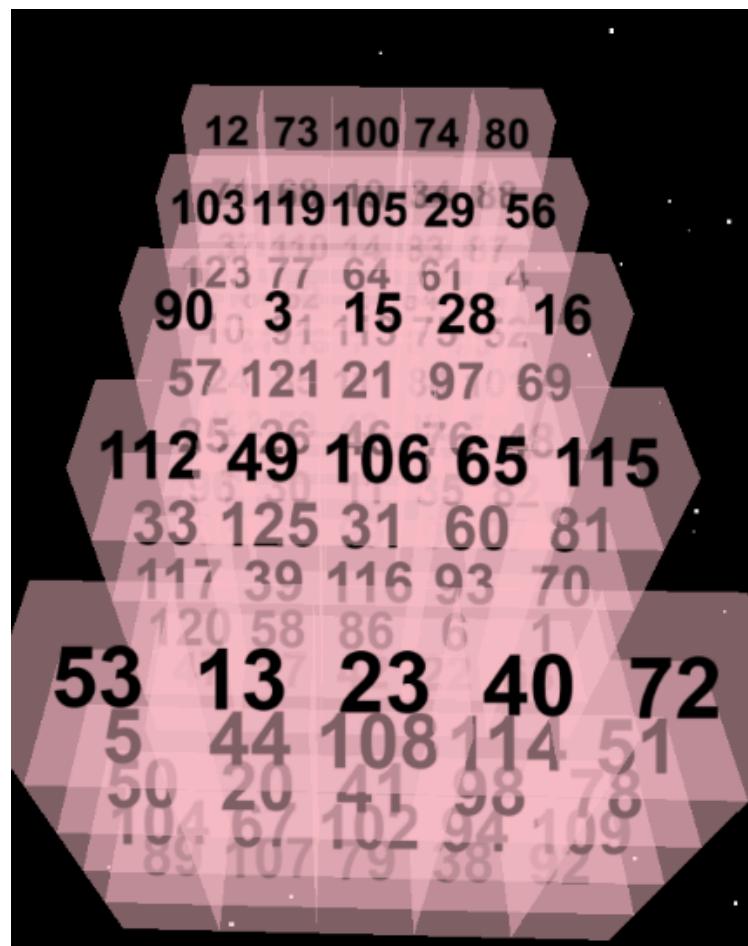
Waktu Eksekusi: 42 ms

Plot

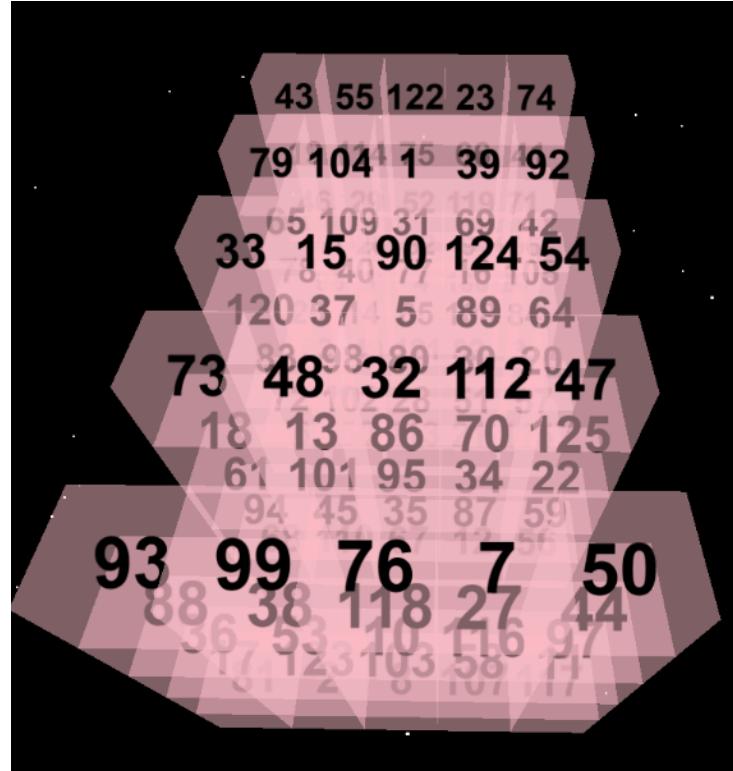


1.3.2.2 Percobaan 2

Initial State



Solved State



Algorithm: Stochastic Hill Climb

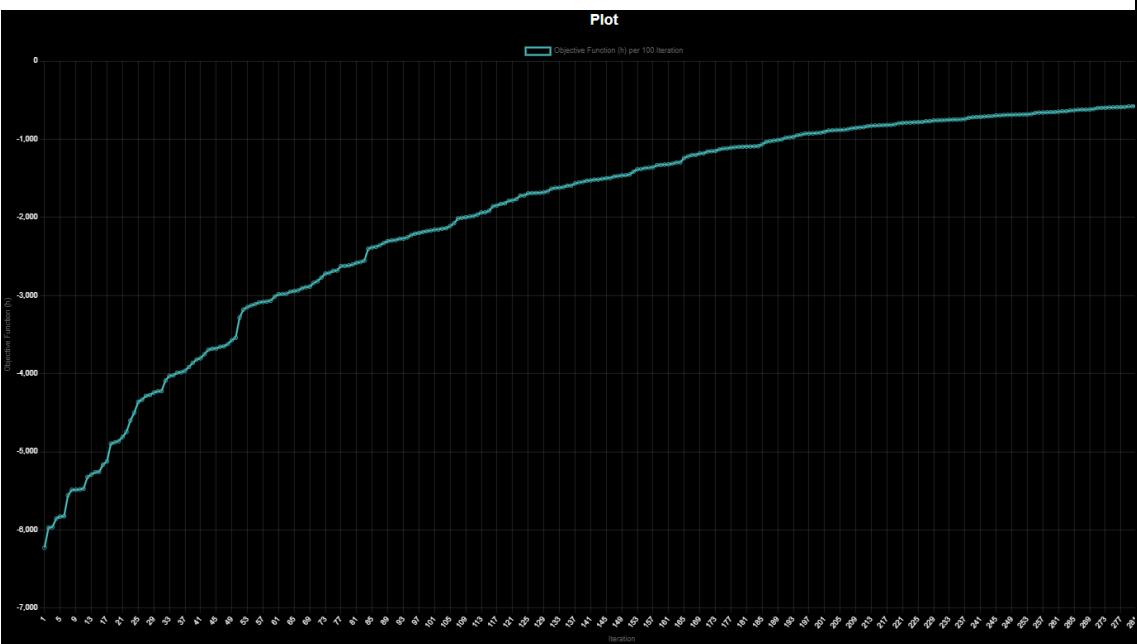
H Before: 6274

H After: 520

Iterations: 297

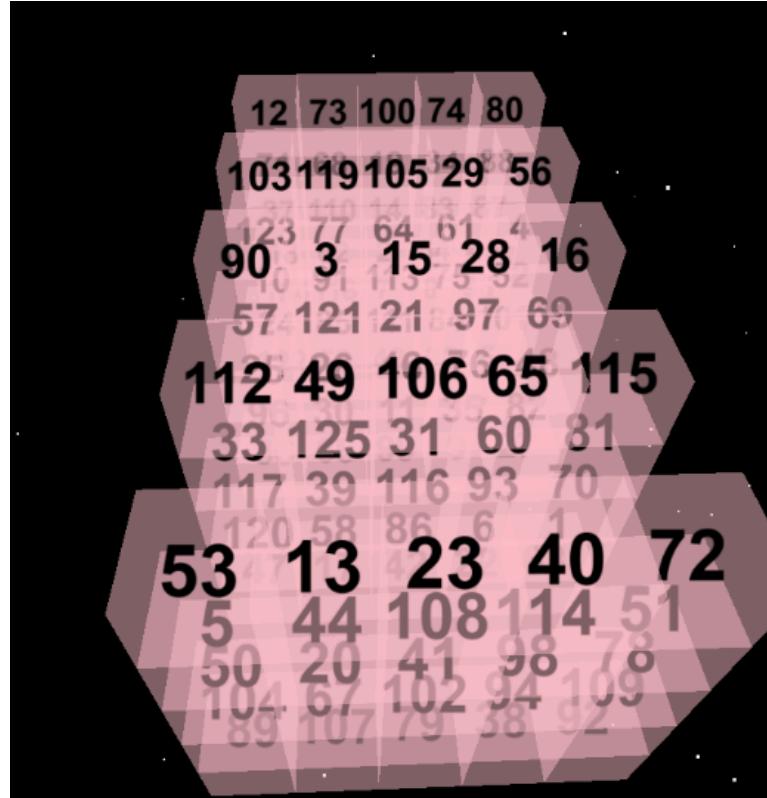
Waktu Eksekusi: 34 ms

Plot

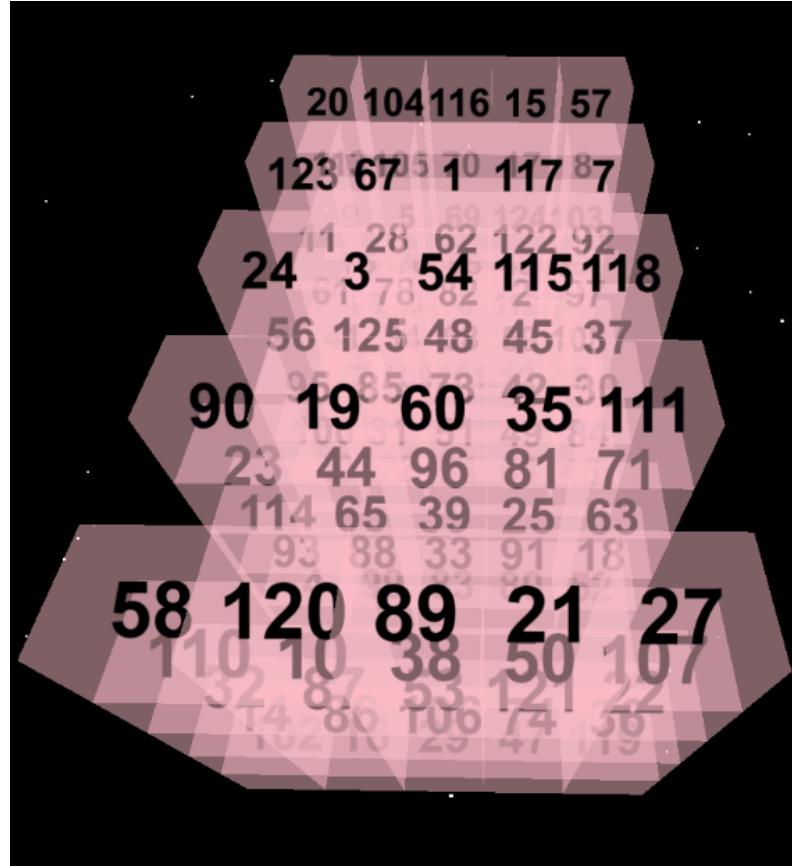


1.3.2.3 Percobaan 3

Initial State



Solved State



Algorithm: Stochastic Hill Climb

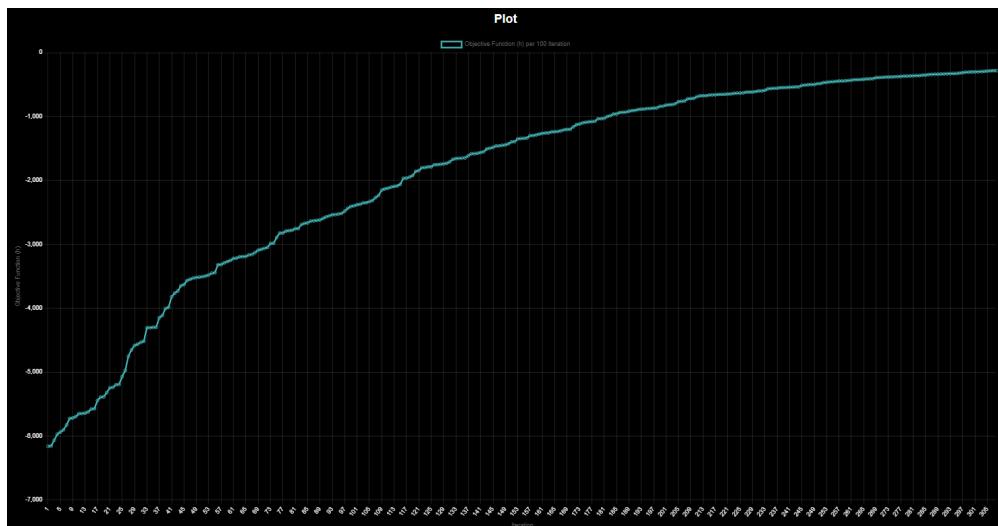
H Before: 6274

H After: 281

Iterations: 308

Waktu Eksekusi: 28 ms

Plot

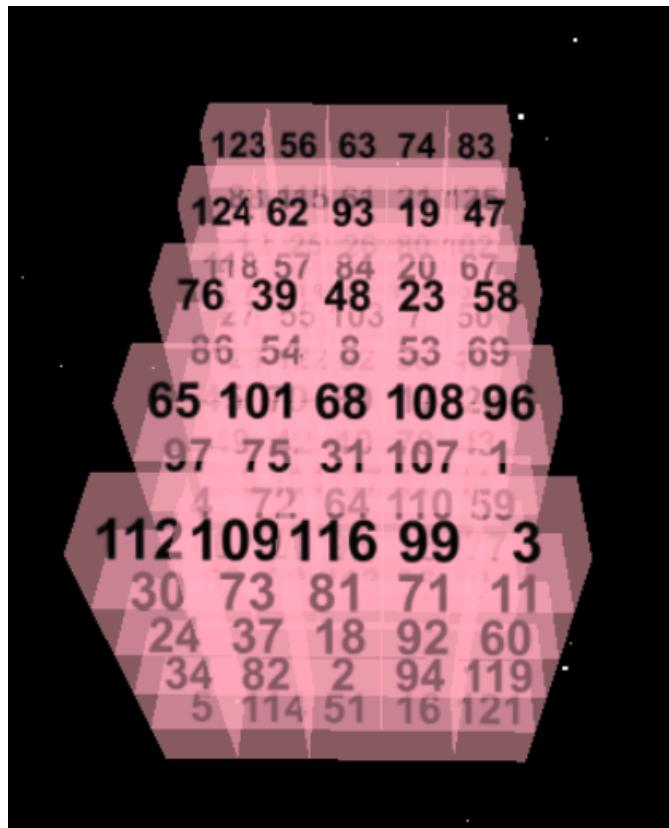


1.3.3 Random Restart Hill-Climbing

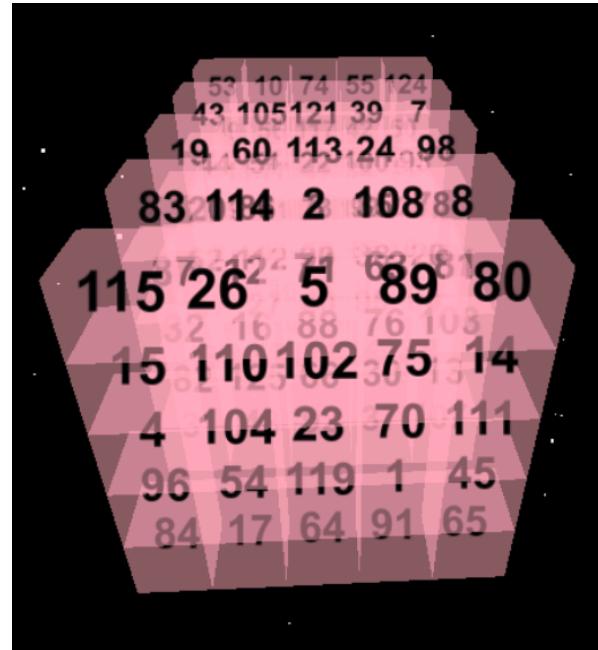
1.3.3.1 Percobaan 1

Max restart : 100

Initial State



Solved State



Algorithm: Random
Restart Hill Climb

H Before: 6129

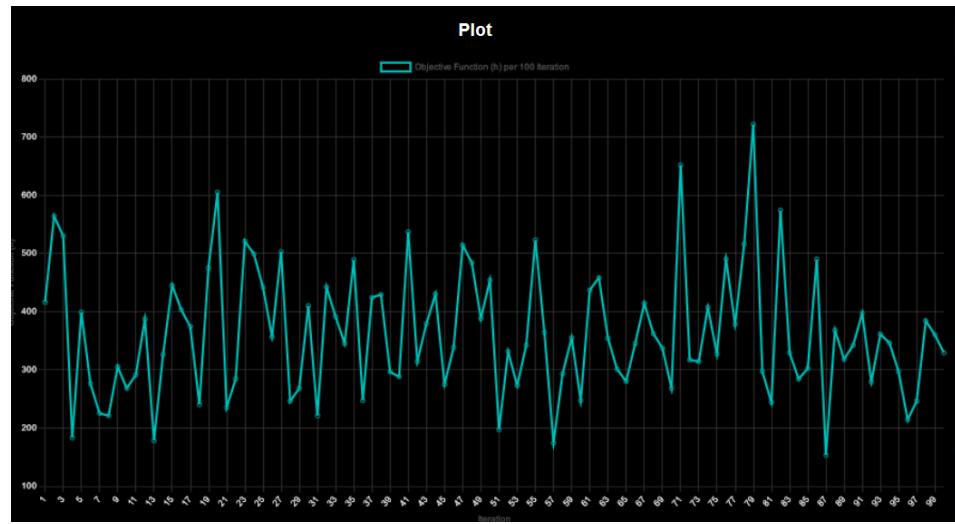
H After: 153

Iterations: 50145

Waktu Eksekusi:
10529 ms

Jumlah Restart: 100

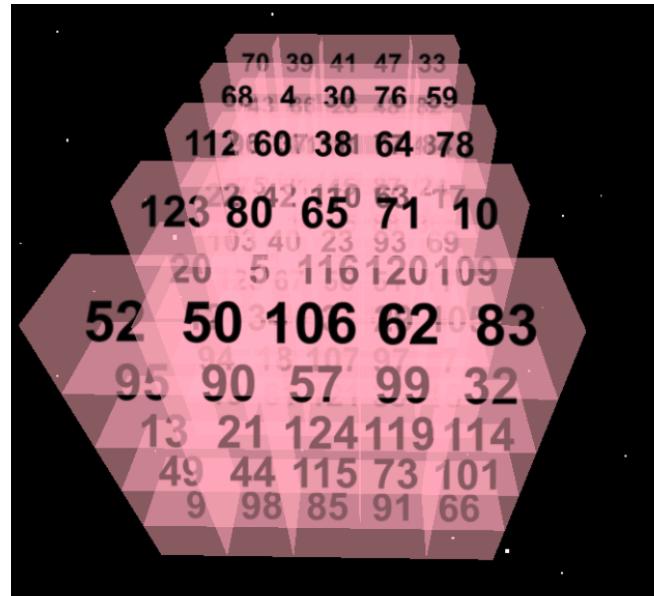
Plot



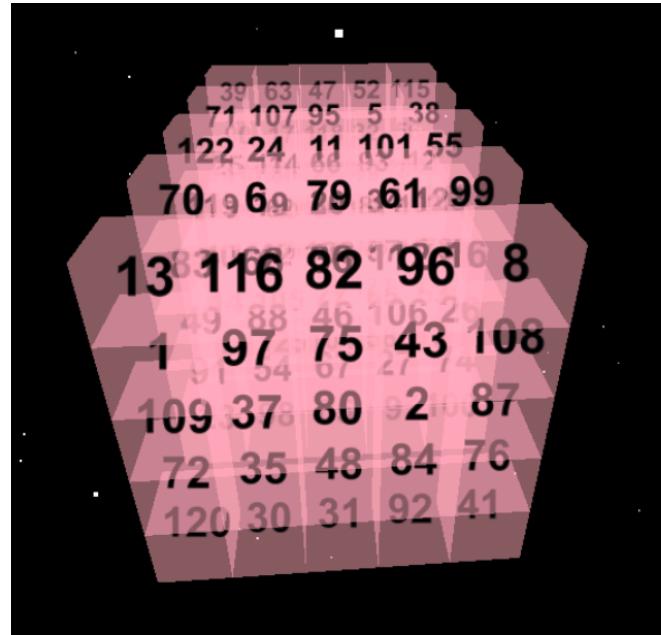
1.3.3.2 Percobaan 2

Max restart : 20

Initial State



Solved State



Algorithm: **Random Restart Hill Climb**

H Before: **6469**

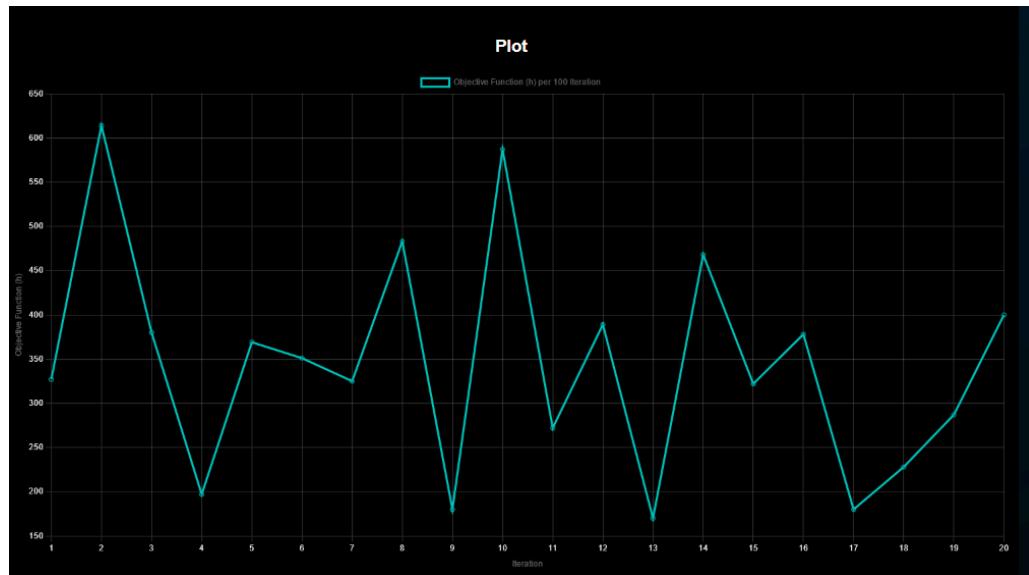
H After: **170**

Iterations: **10113**

Waktu Eksekusi: **2200**
ms

Jumlah Restart: **20**

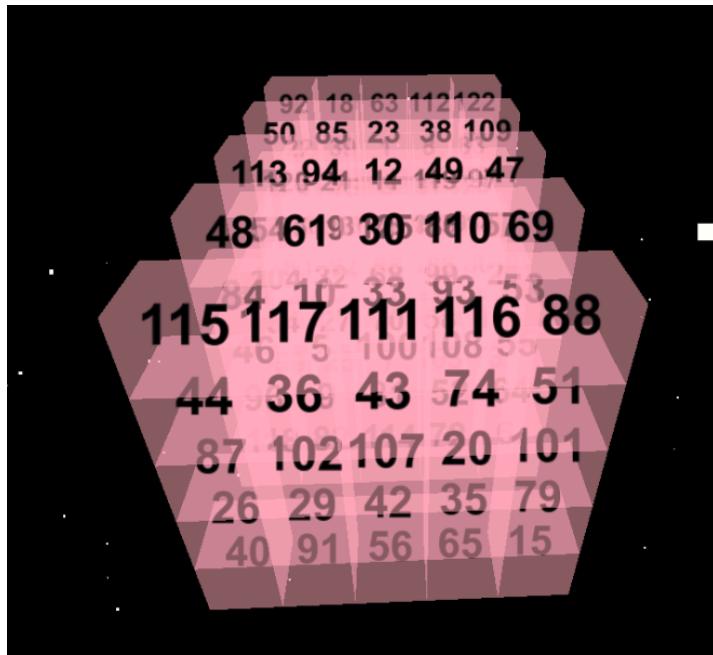
Plot



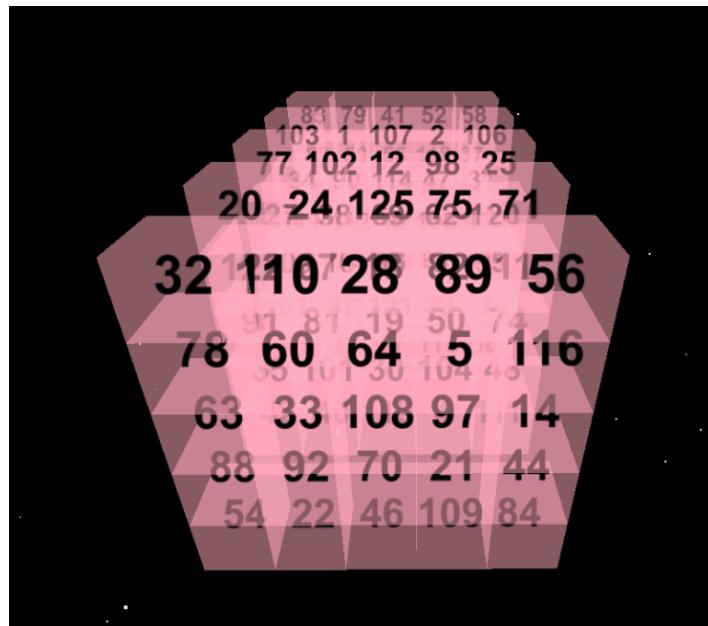
1.3.3.3 Percobaan 3

Max restart: 50

Initial State



Solved State



Algorithm: **Random Restart Hill Climb**

H Before: **6631**

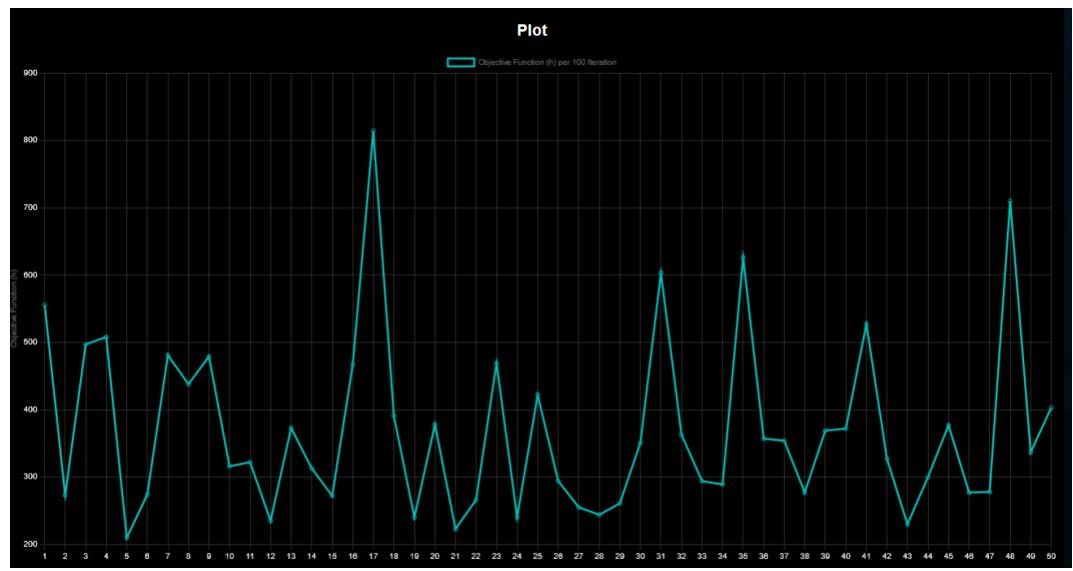
H After: **210**

Iterations: **25131**

Waktu Eksekusi: **5949**
ms

Jumlah Restart: **50**

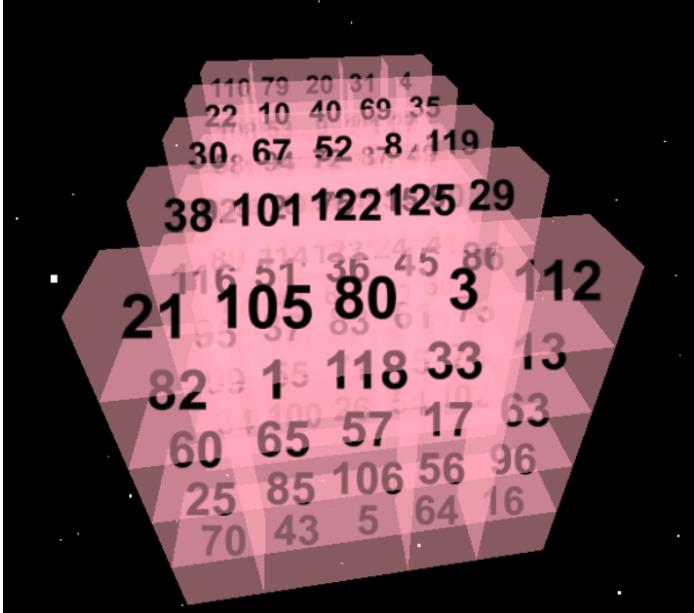
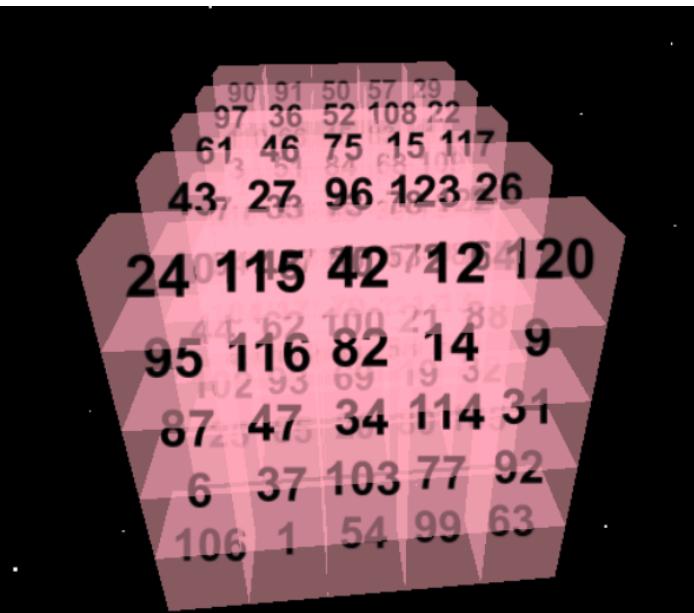
Plot



1.3.4 Sideways Move

1.3.4.1 Percobaan 1

Max Sideways : 100

Initial State	 A 4x4 grid of numbered tiles from 1 to 15, with an empty space at position (3,3). The tiles are arranged as follows: Row 1: 110, 79, 20, 31, 4 Row 2: 22, 10, 40, 69, 35 Row 3: 30, 67, 52, 8, 119 Row 4: 38, 101, 122, 125, 29 The empty space is located at the bottom-right position.
Solved State	 A 4x4 grid of numbered tiles from 1 to 15, with an empty space at position (3,3). The tiles are arranged as follows: Row 1: 90, 91, 50, 57, 39 Row 2: 97, 36, 52, 108, 22 Row 3: 61, 46, 75, 15, 117 Row 4: 43, 27, 96, 123, 26 The empty space is located at the bottom-right position.

**Algorithm: Sideways
Hill Climb**

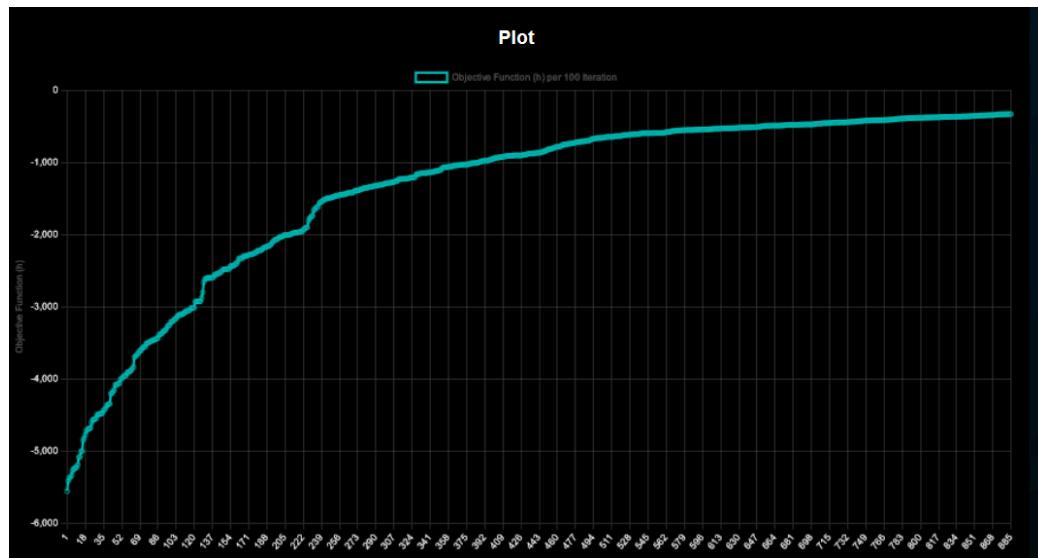
H Before: 5682

H After: 333

Iterations: 885

**Waktu Eksekusi: 275
ms**

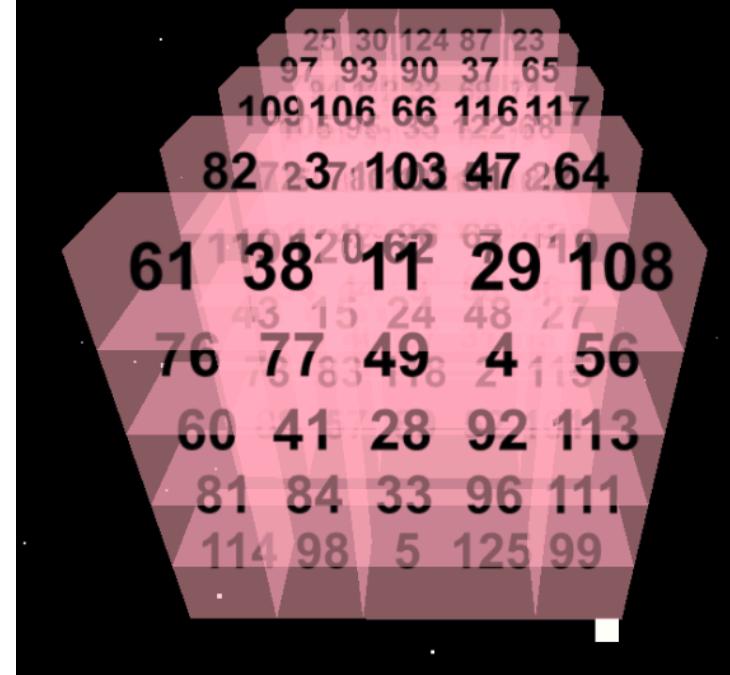
Plot



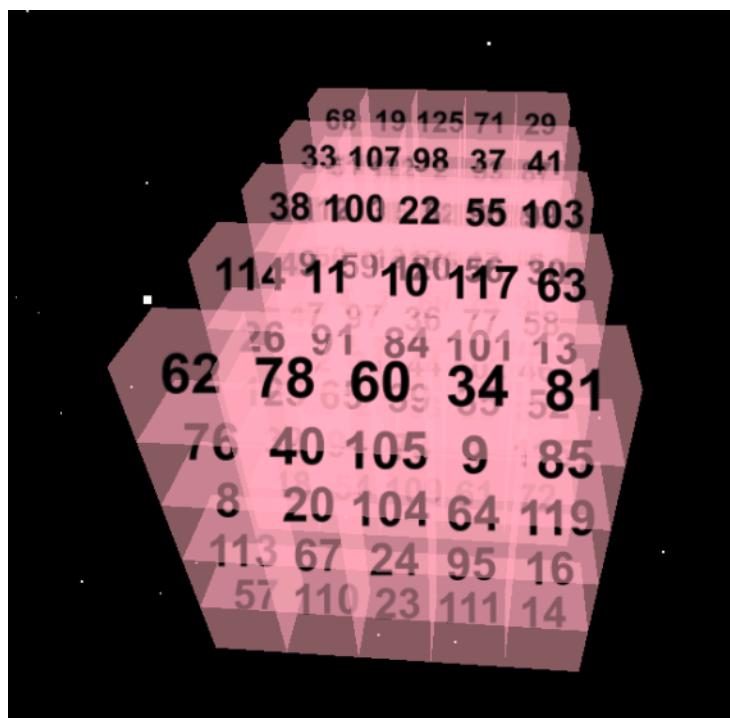
1.3.4.2 Percobaan 2

Max sideways : 51

Initial State



Solved State



Algorithm: **Sideways
Hill Climb**

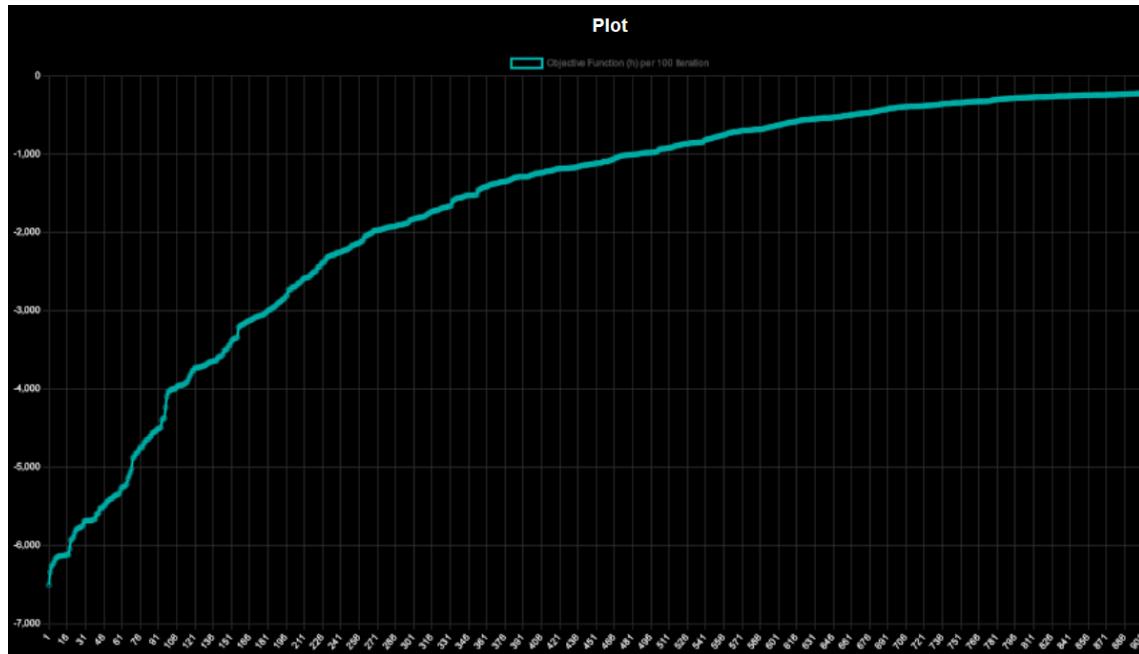
H Before: **6524**

H After: **204**

Iterations: **967**

Waktu Eksekusi: **288**
ms

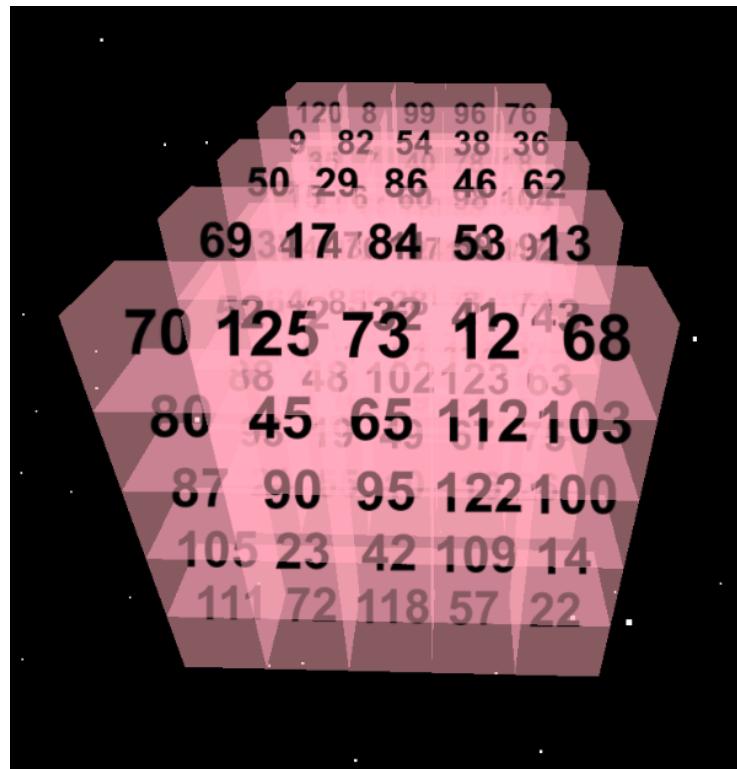
Plot



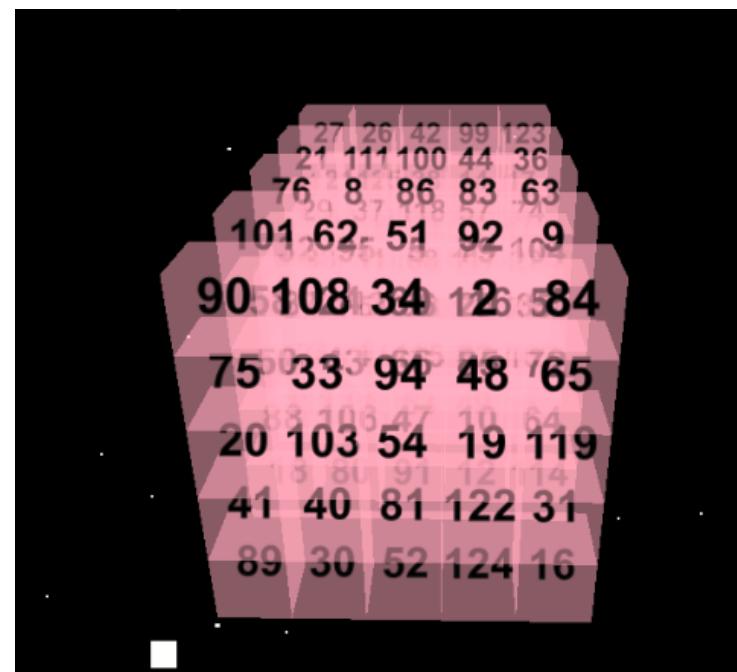
1.3.4.3 Percobaan 3

Max Sideways : 10

Initial State



Solved State



Algorithm: **Sideways
Hill Climb**

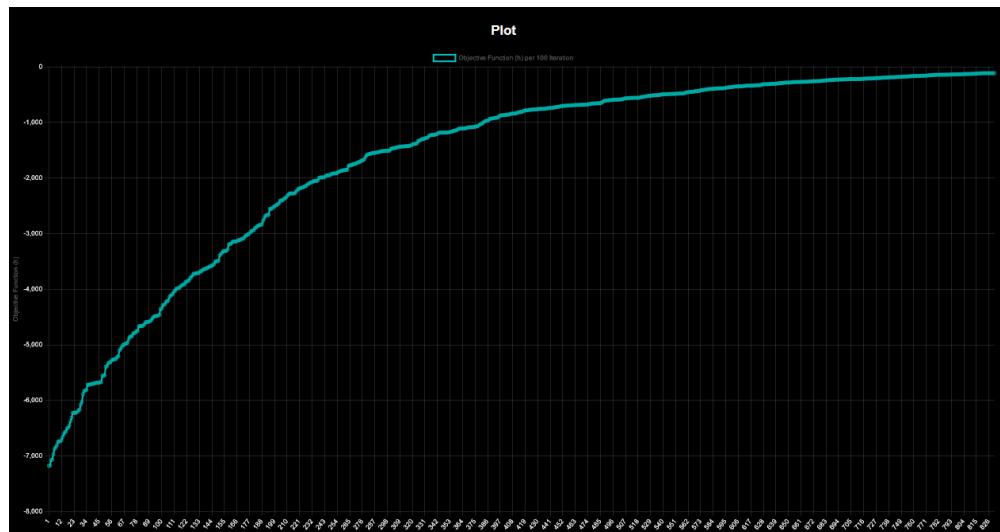
H Before: **7270**

H After: **115**

Iterations: **831**

Waktu Eksekusi: **161**
ms

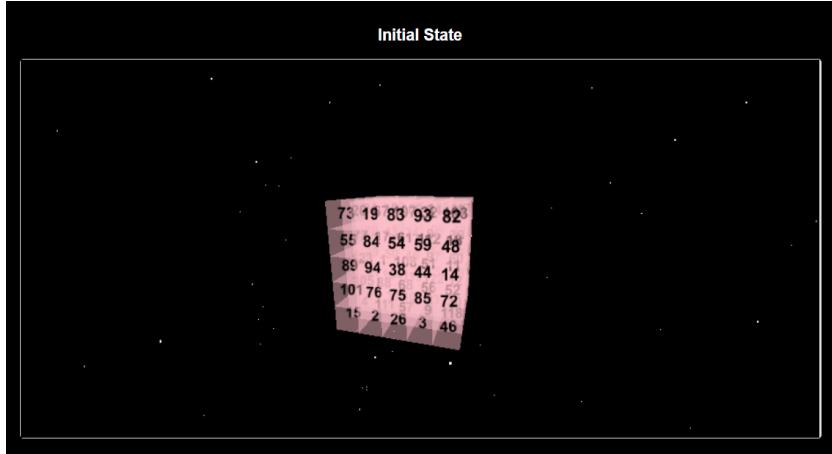
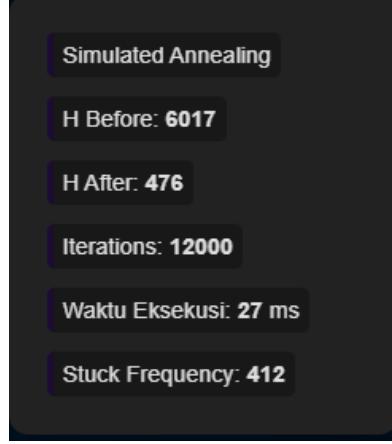
Plot

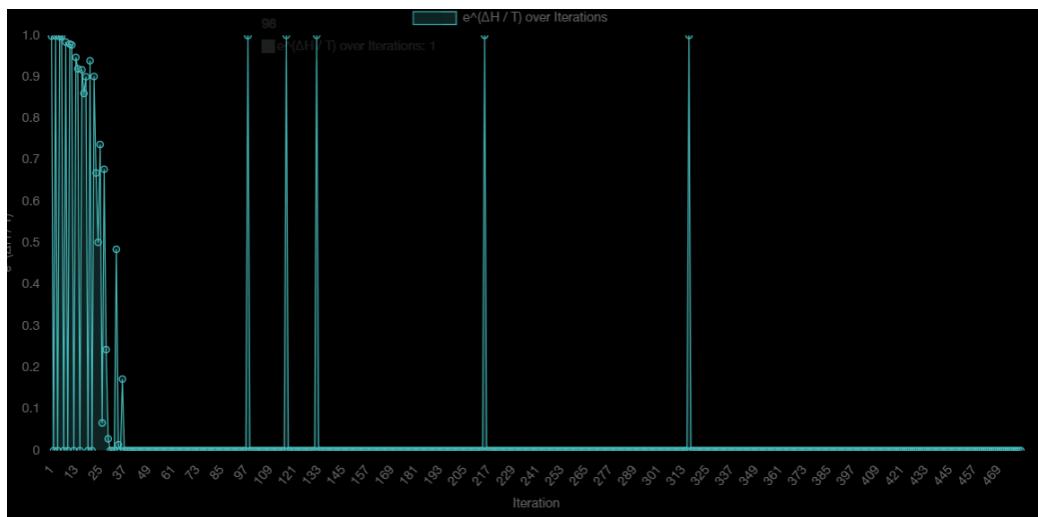
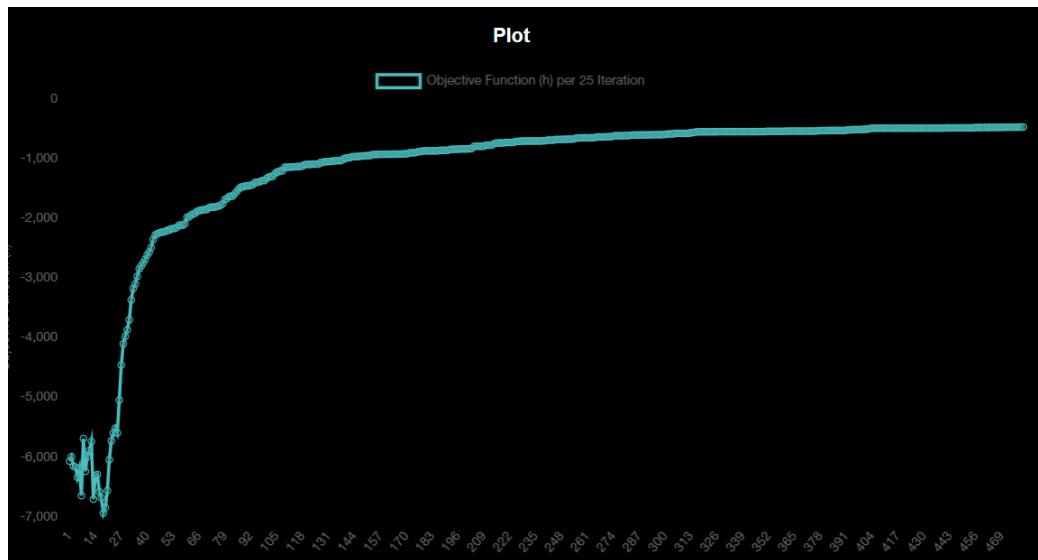


1.3.5 Simulated Annealing

1.3.5.1 Percobaan 1

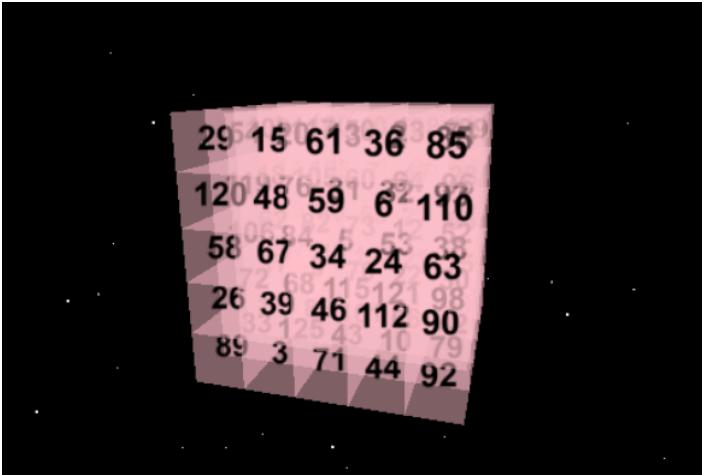
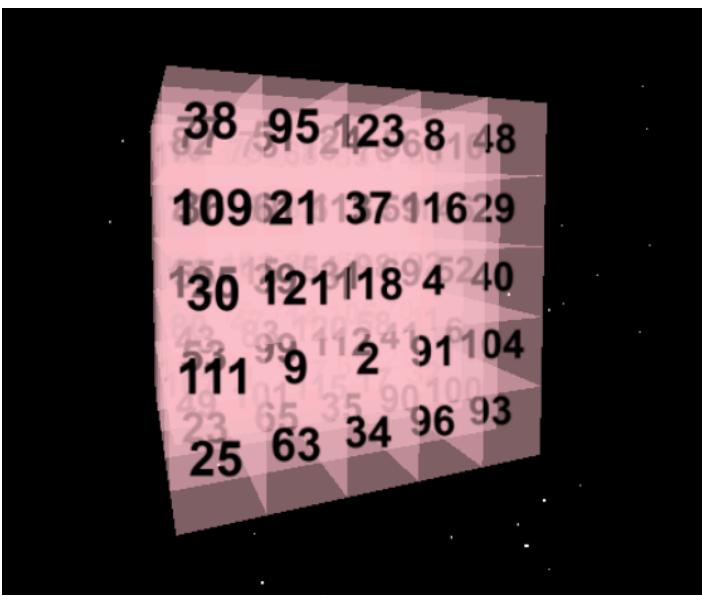
(TValue 30000, coolingRate 0.99, maximumIteration 12000)

Initial State	
Solved State	
Plot	



1.3.5.2 Percobaan 2

(TValue 30000, coolingRate 0.99, maximumIteration 12000)

Initial State	
Solved State	

Simulated Annealing

H Before: **6236**

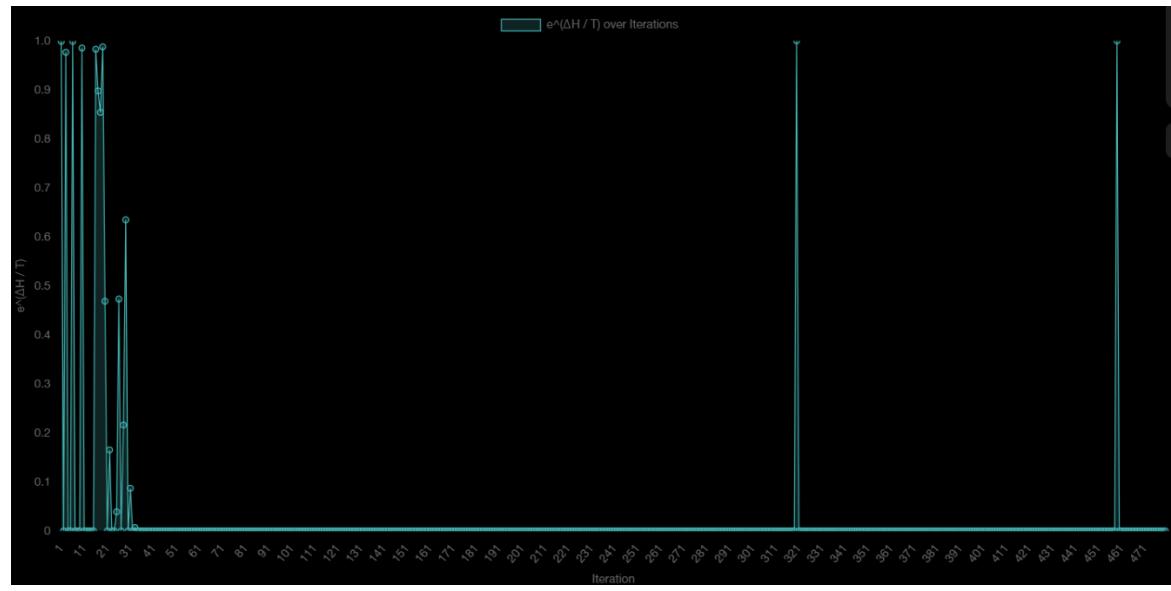
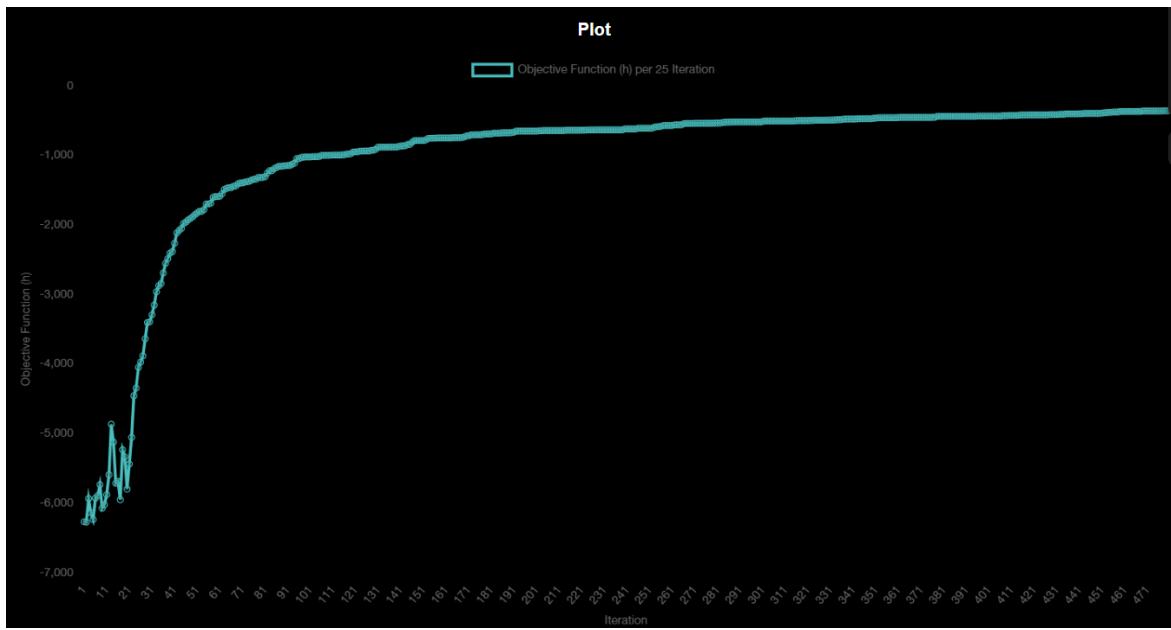
H After: **361**

Iterations: **12000**

Waktu Eksekusi: **32 ms**

Stuck Frequency: **393**

Plot



1.3.5.3 Percobaan 3

(TValue 100, coolingRate 0.99, maximumIteration 12000)

Initial State	<p>The initial state of a 5x5 number puzzle. The numbers are arranged in a specific pattern. The center cell contains the value 66. Other visible values include 111, 73, 59, 69, 117, 48, 30, 79, 106, 32, 114, 107, 31, 45, 90, 17, 121, 14, 70, 91, 108, 103, 66, 28, 74, 109, 67, 42, 62, 9, 1, 29, 89, 39, 10, 120, 44, 78, 93, 5, 19, 55, 20, 118, 92, 23, 16, 107, 111, 60, 55, 38, 51, 103, 74, 99, 97, 14, 2, 106, 123, 82, 41, 58, 10, 43, 39, 125, 31, 57, 28, 87, 48, 101, 54, 30, 89, 81, 1, 116, 122, 5, 32, 85, 69, 72, 95, 27, 102, 19.</p>
Solved State	<p>The solved state of a 5x5 number puzzle. The numbers are arranged in a specific pattern. The center cell contains the value 66. Other visible values include 94, 121, 15, 64, 21, 23, 16, 107, 111, 60, 55, 38, 51, 103, 74, 99, 97, 14, 2, 106, 123, 82, 41, 58, 10, 43, 39, 125, 31, 57, 28, 87, 48, 101, 54, 30, 89, 81, 1, 116, 122, 5, 32, 85, 69, 72, 95, 27, 102, 19.</p>

Simulated Annealing

H Before: **6105**

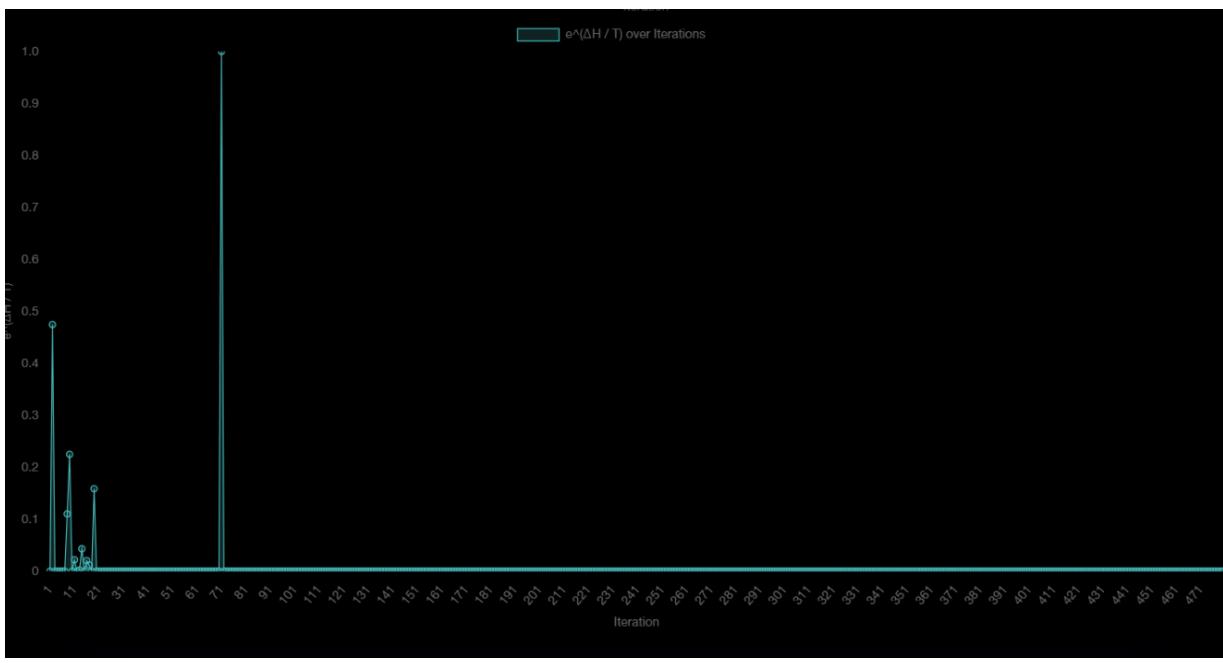
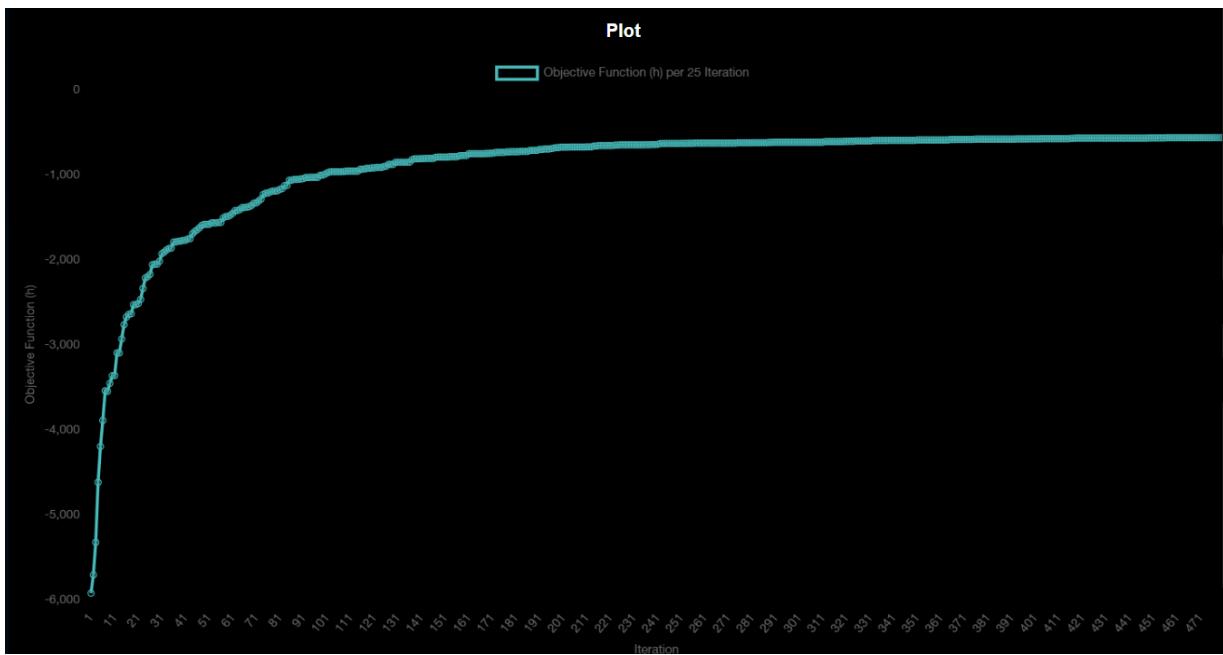
H After: **564**

Iterations: **12000**

Waktu Eksekusi: **94 ms**

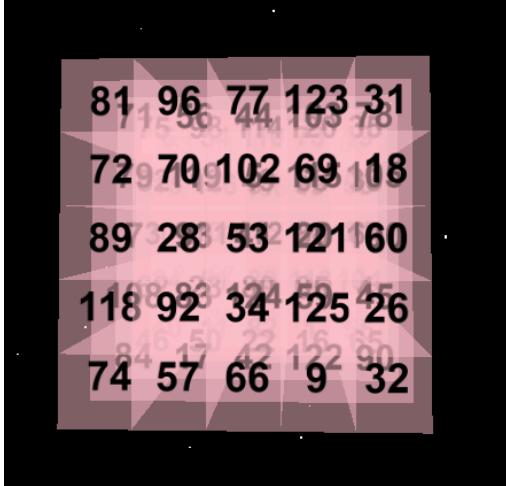
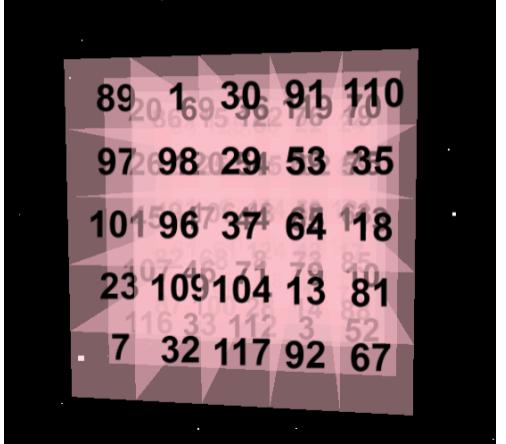
Stuck Frequency: **133**

Plot



1.3.5.4 Percobaan 4

(TValue 30000, coolingRate 0.99, maximumIteration 12000)

Initial State	
Solved State	

Simulated Annealing

H Before: 6136

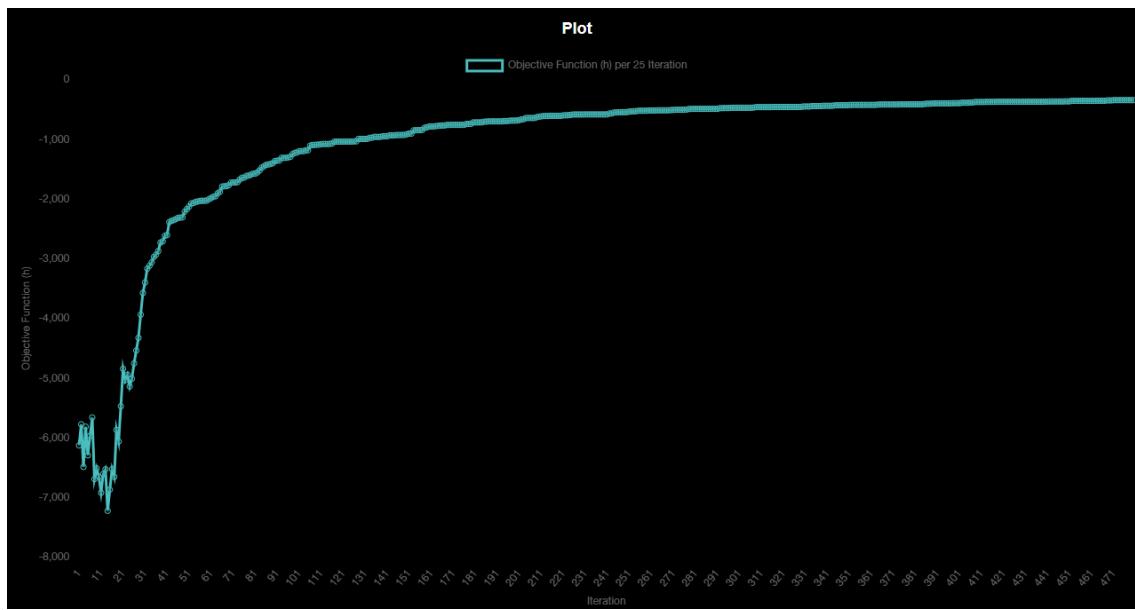
H After: 334

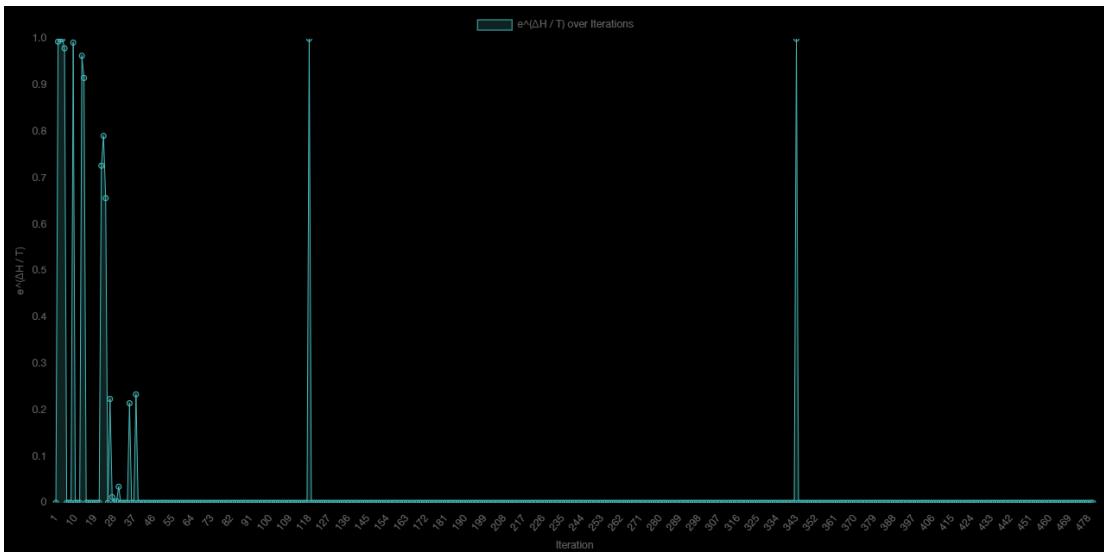
Iterations: 12000

Waktu Eksekusi: 30 ms

Stuck Frequency: 448

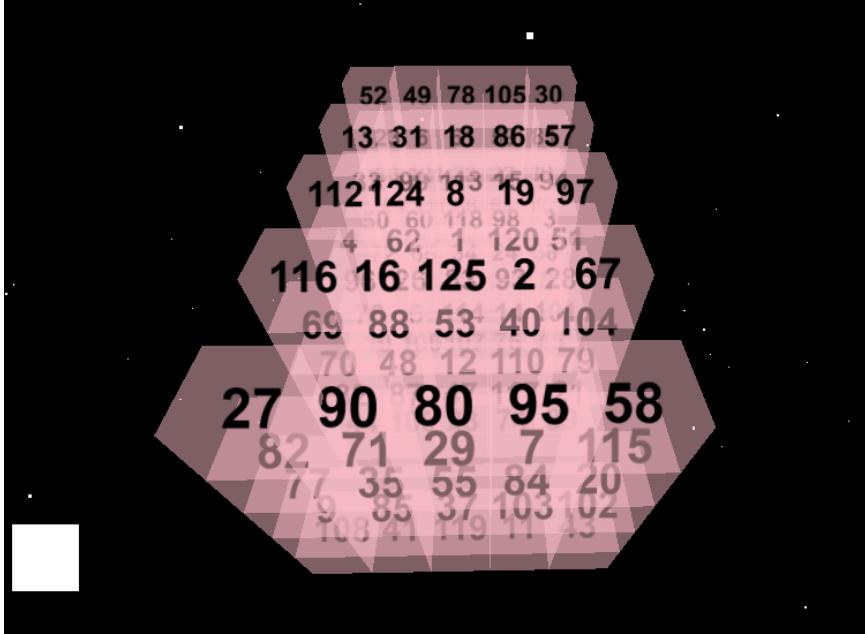
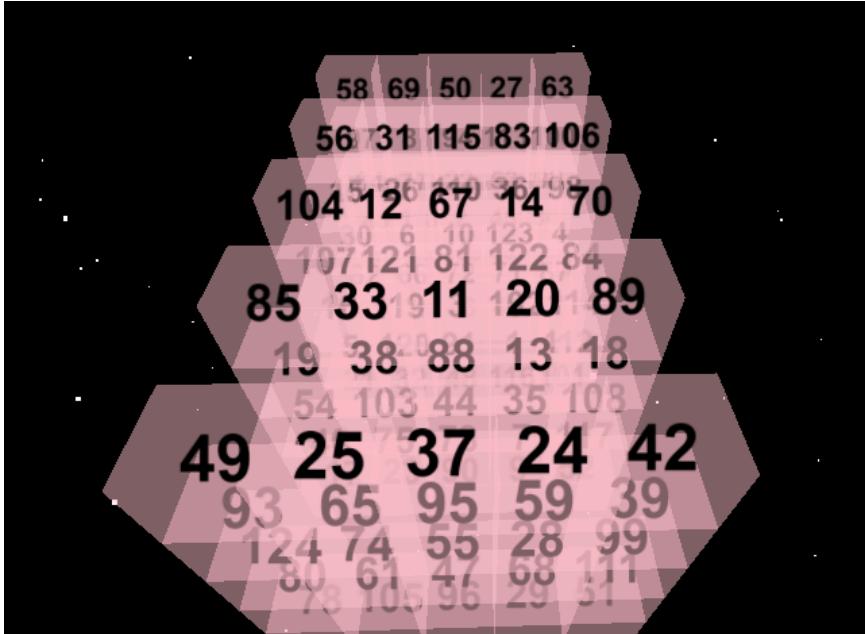
Plot





1.3.5.5 Percobaan 5

(TValue 30000, coolingRate 0.99, maximumIteration 1000)

Initial State	
Solved State	

Simulated Annealing

H Before: 6533

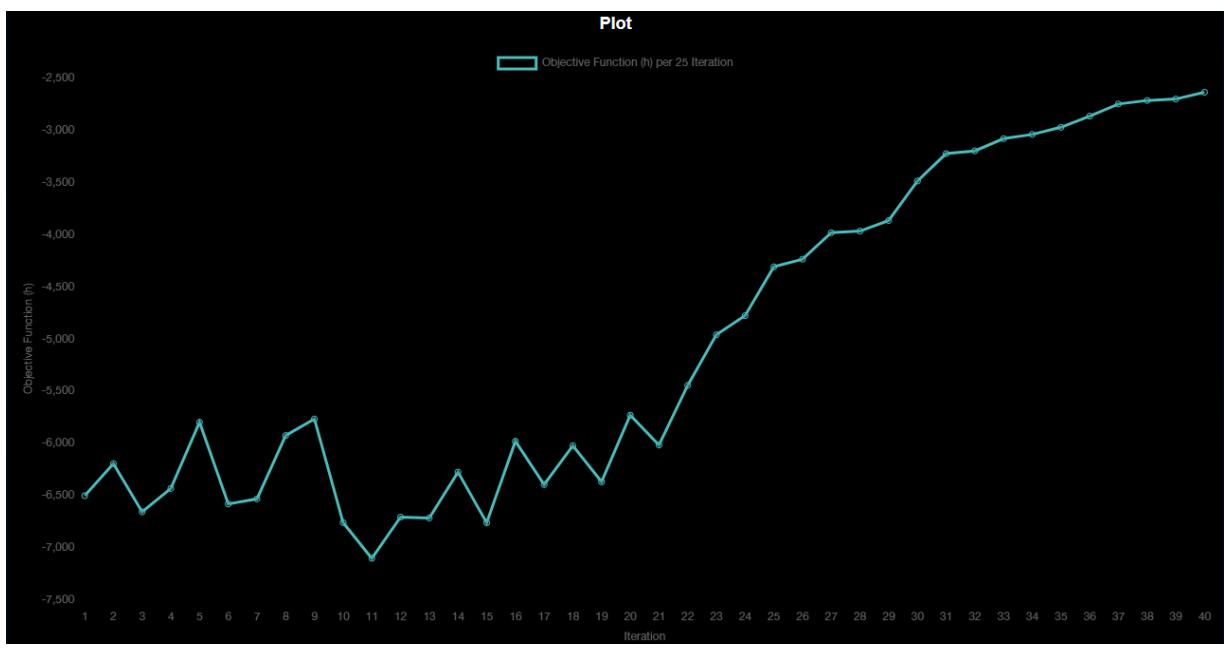
H After: 2563

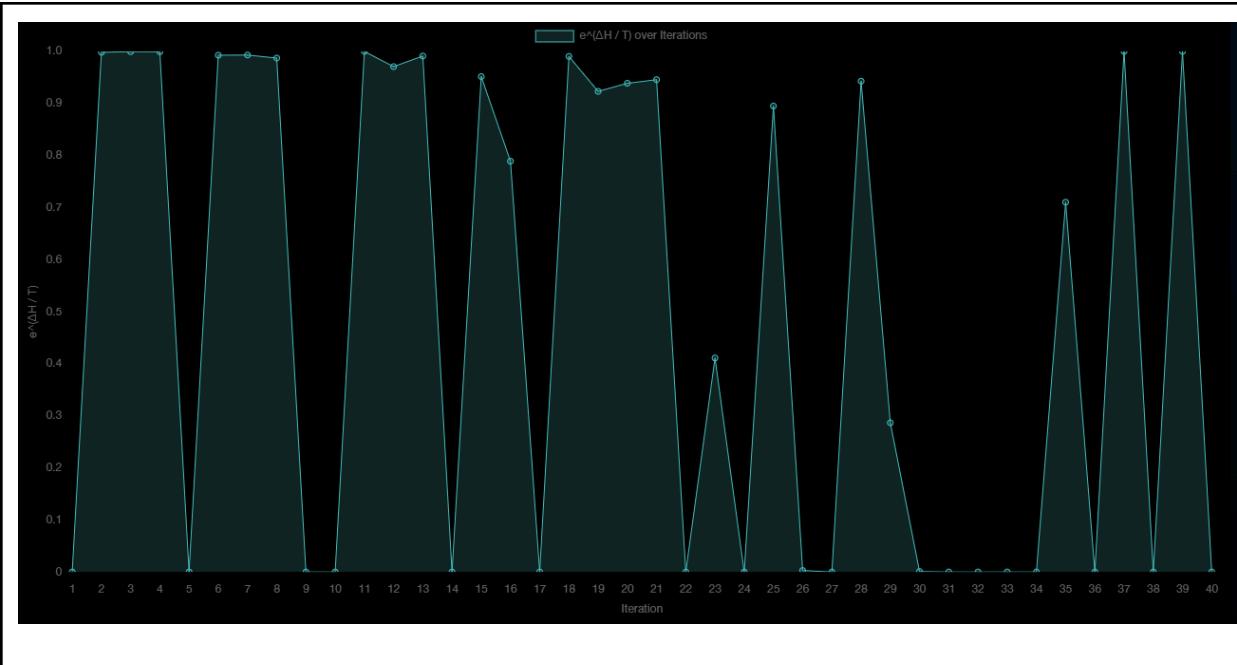
Iterations: 1000

Waktu Eksekusi: 14 ms

Stuck Frequency: 346

Plot



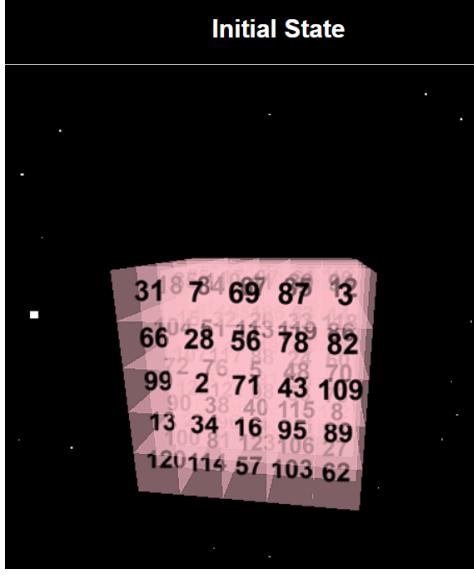


1.3.6 Genetic Algorithm

(Iterasi beda, Populasi tetap)

1.3.6.1 Percobaan 1

Iterasi = 10 populasi = 50

Initial State	<p style="text-align: center;">Initial State</p> 
Solved State	<p style="text-align: center;">Solved State</p> 

**Algorithm: Genetic
Algorithm**

H Before: 5744

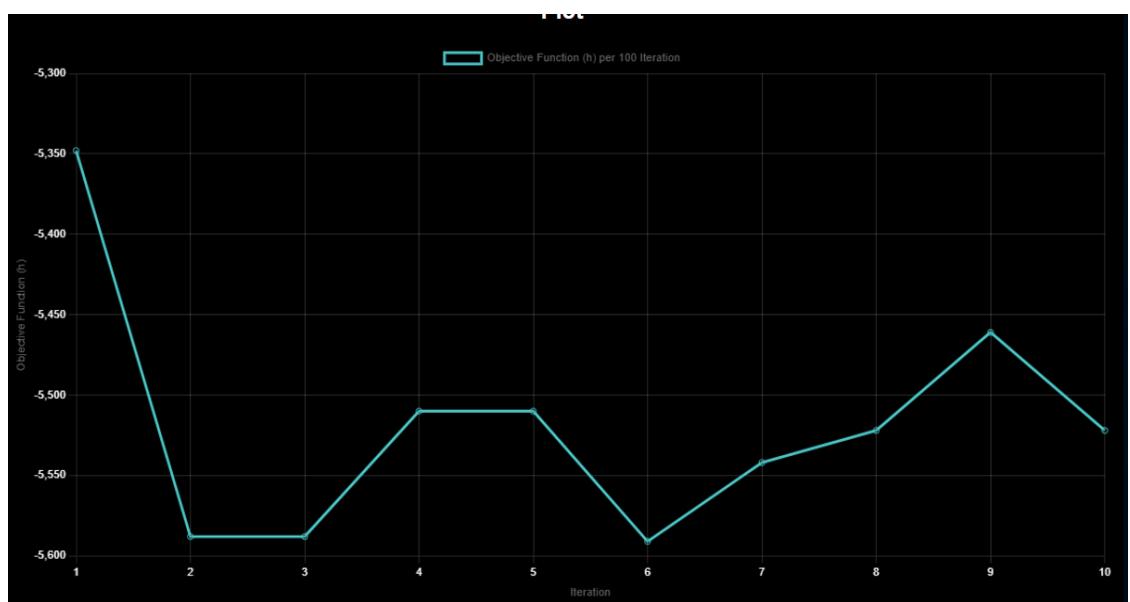
H After: 5522

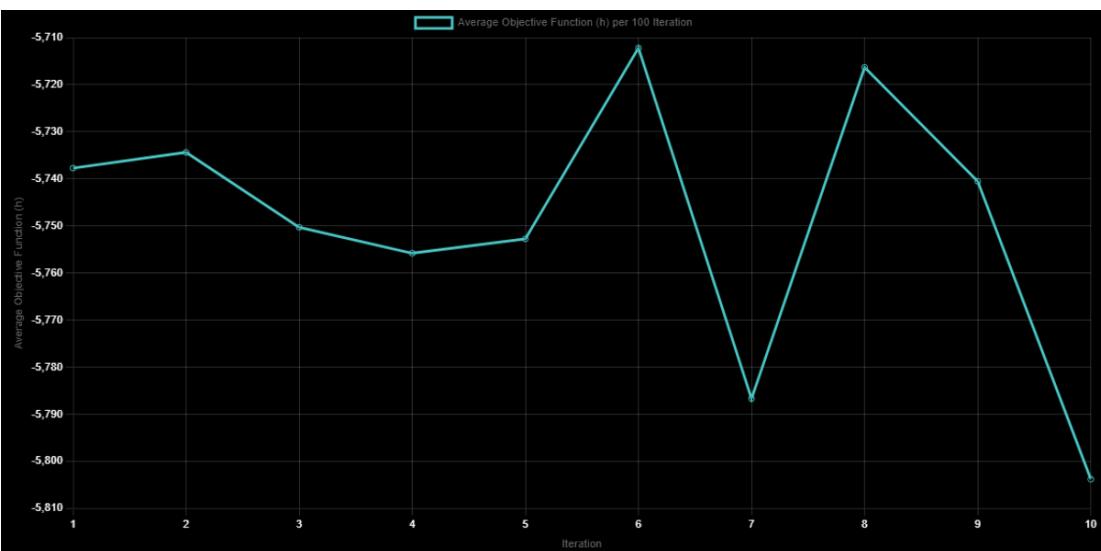
Iterations: 10

Populations: 50

Waktu Eksekusi: 22 ms

Plot

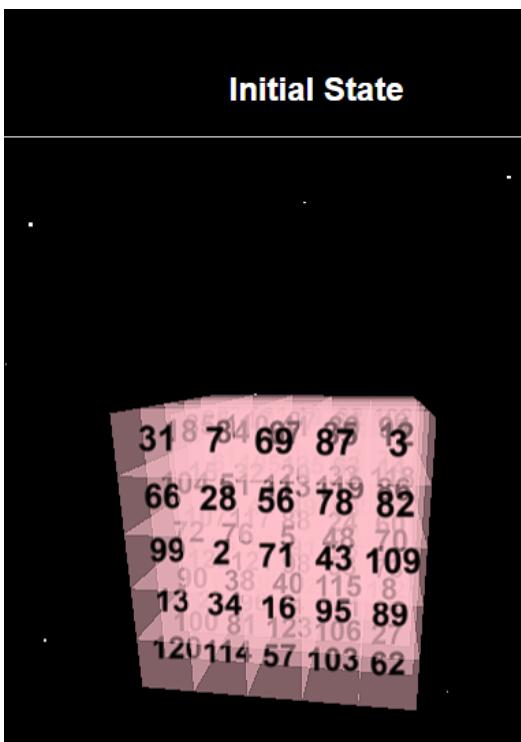




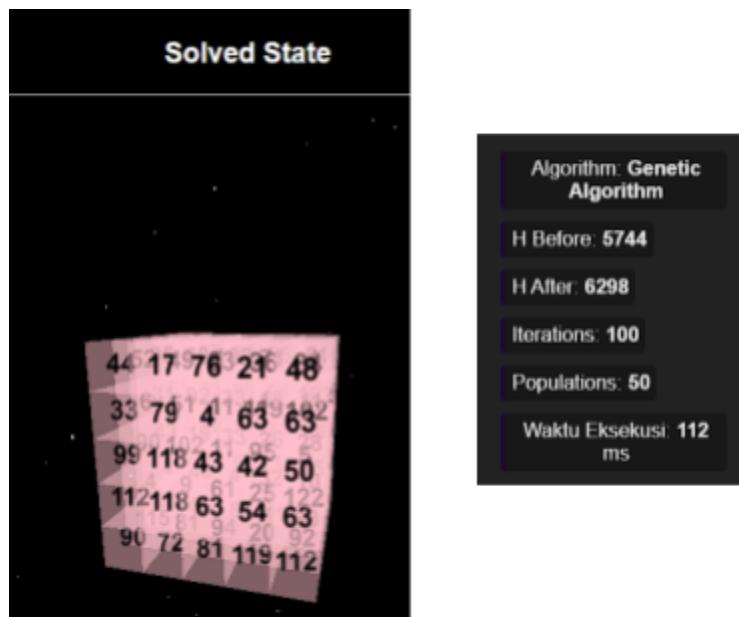
1.3.6.2 Percobaan 2

Iterasi = 100 populasi = 50

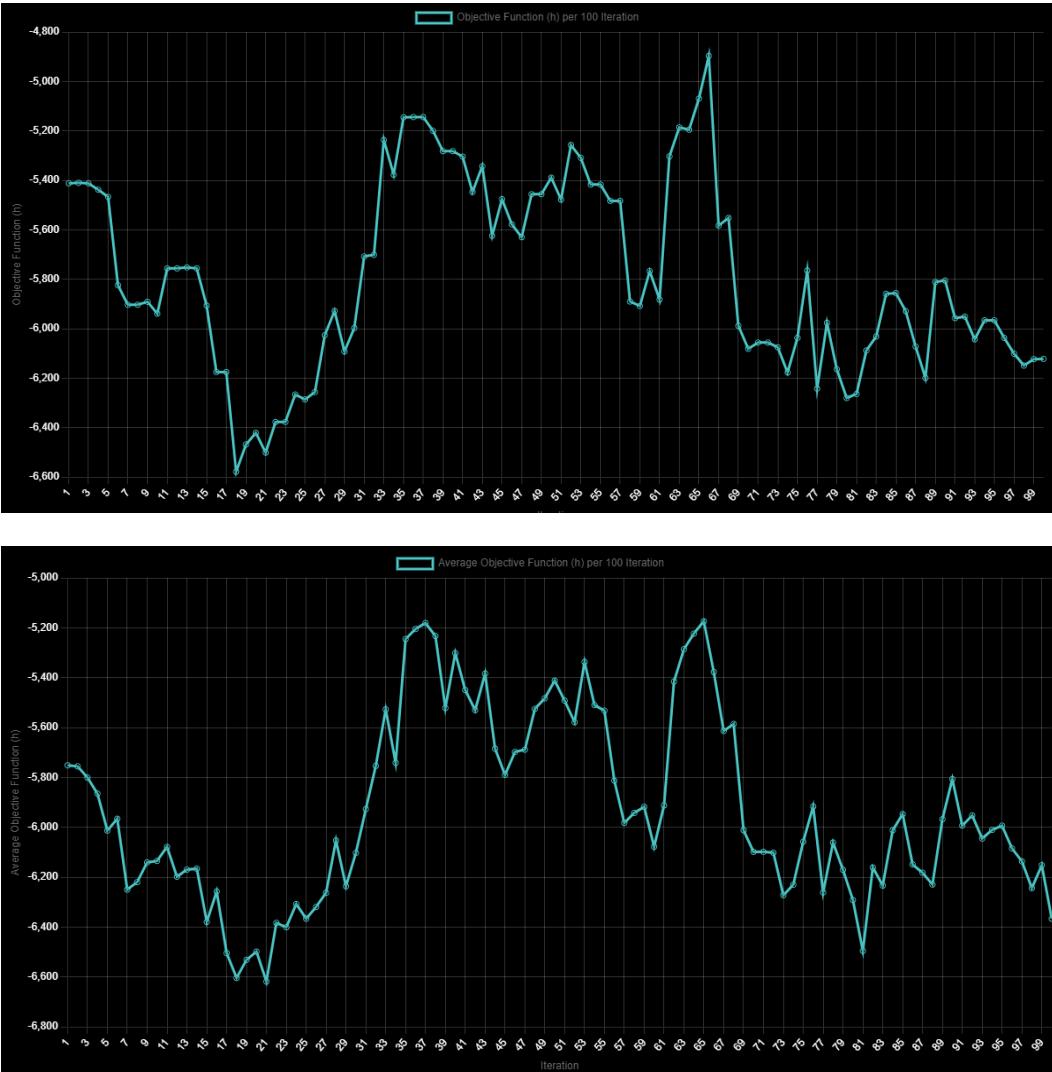
Initial State



Solved State



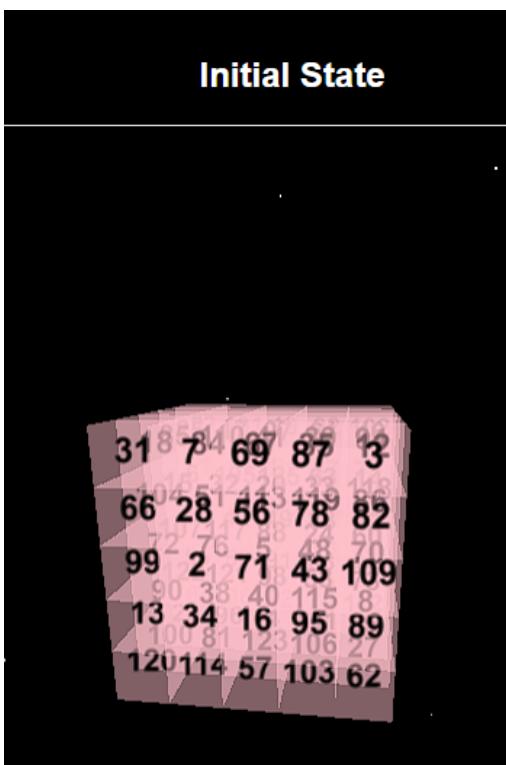
Plot



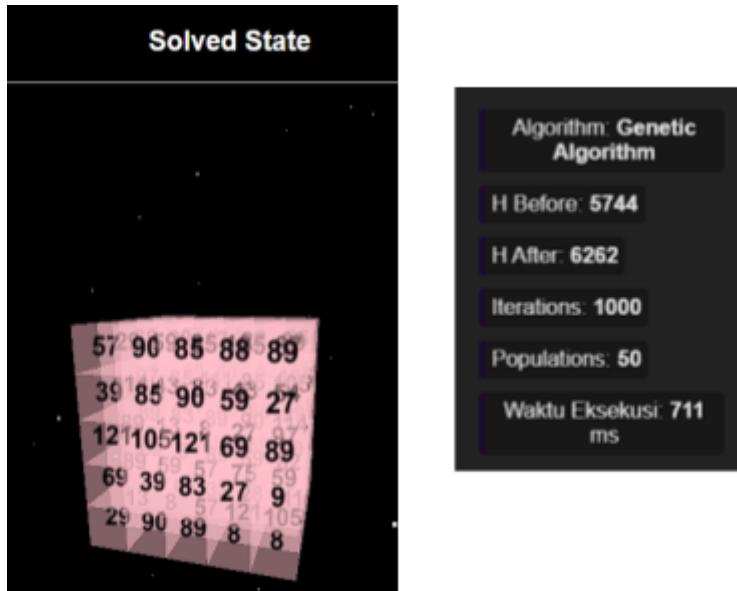
1.3.6.3 Percobaan 3

Iterasi = 1000 populasi = 50

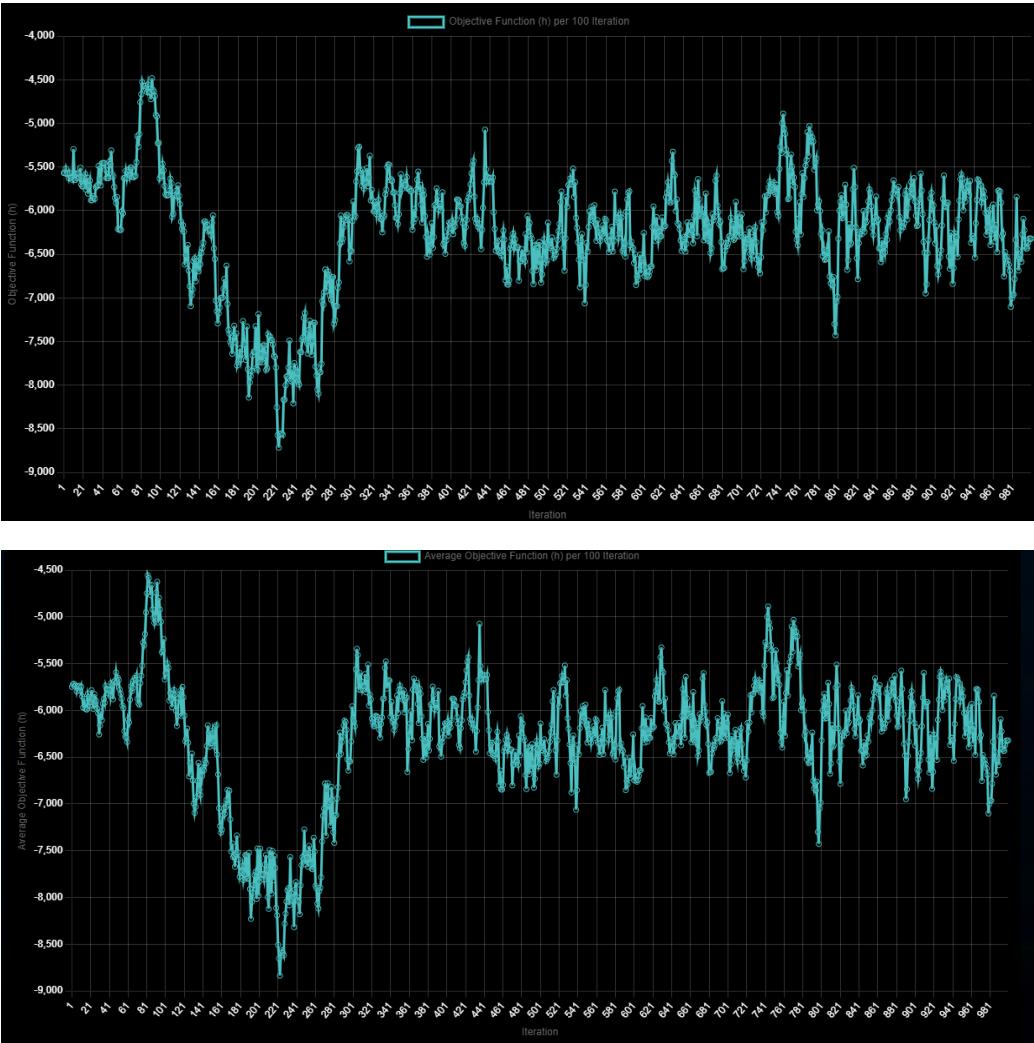
Initial State



Solved State



Plot



(Populasi beda, Iterasi tetap)

1.3.6.4 Percobaan 4

Iterasi = 50 populasi = 10

Initial State

Initial State

31 7 69 87 3
66 28 56 78 82
99 2 71 43 109
13 34 16 95 89
120 114 57 103 62

Solved State

Solved State

90 29 18 75 3
66 99 56 78 49
26 2 71 43 109
14 34 13 95 89
120 114 12 41 62

Algorithm: Genetic Algorithm

H Before: 5744

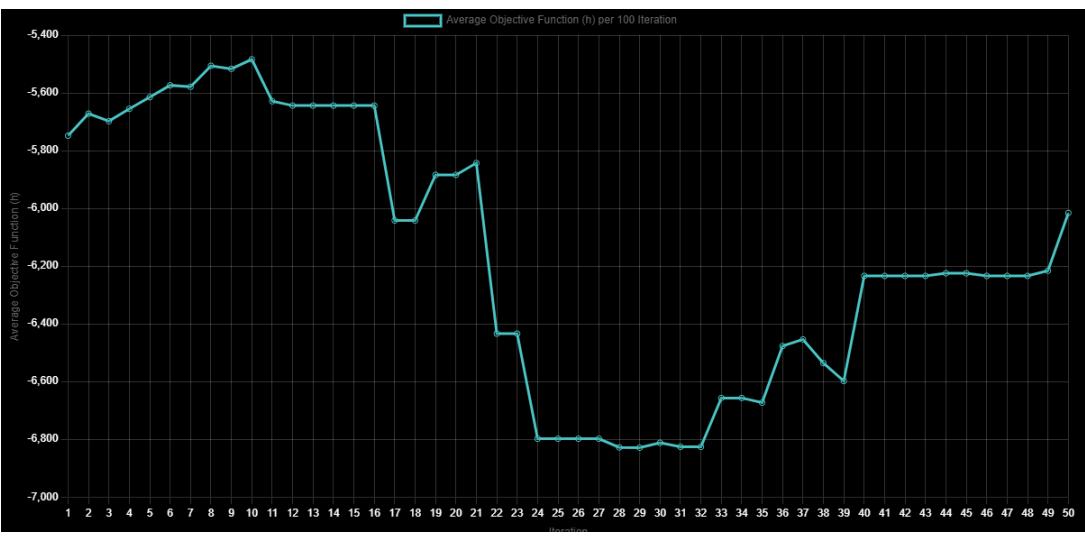
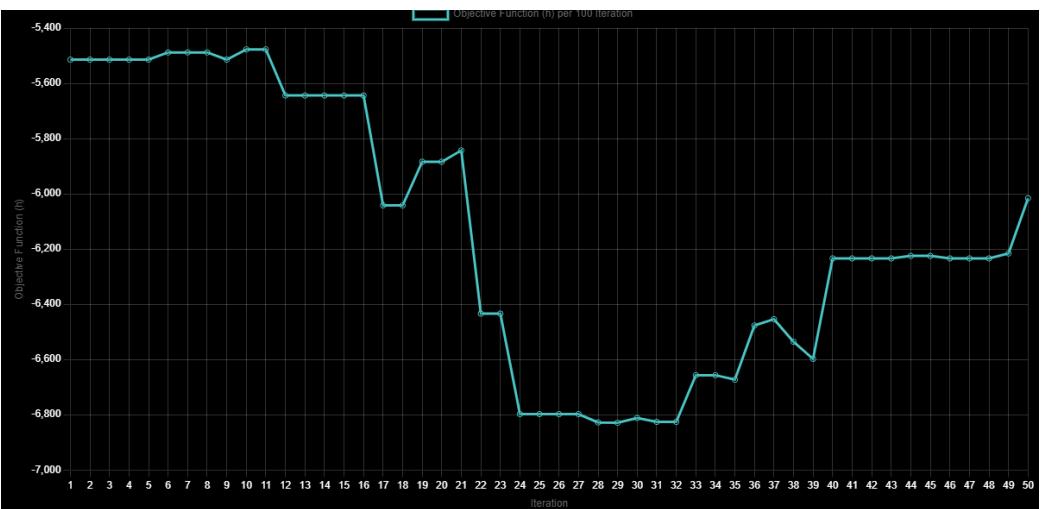
H After: 6016

Iterations: 50

Populations: 10

Waktu Eksekusi: 38 ms

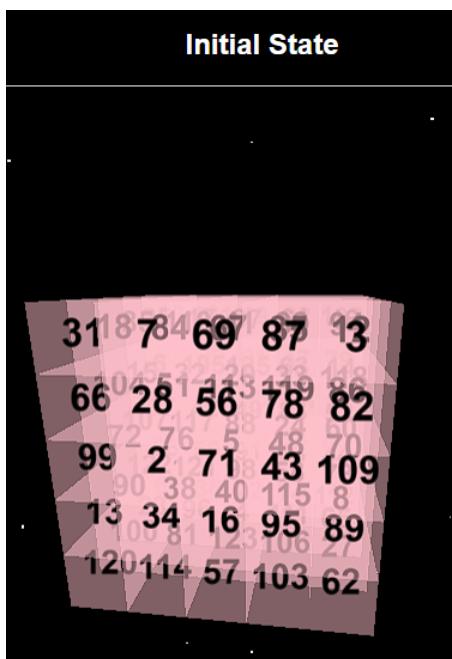
Plot



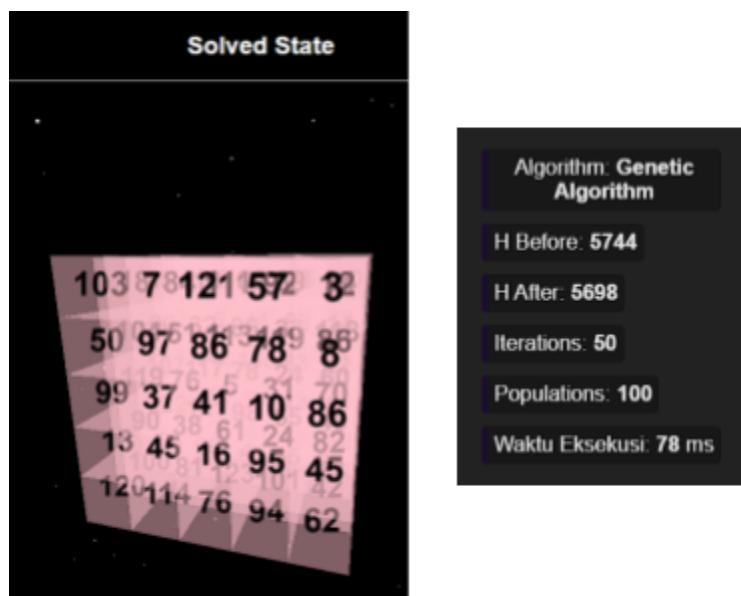
1.3.6.5 Percobaan 5

Iterasi = 50 populasi = 100

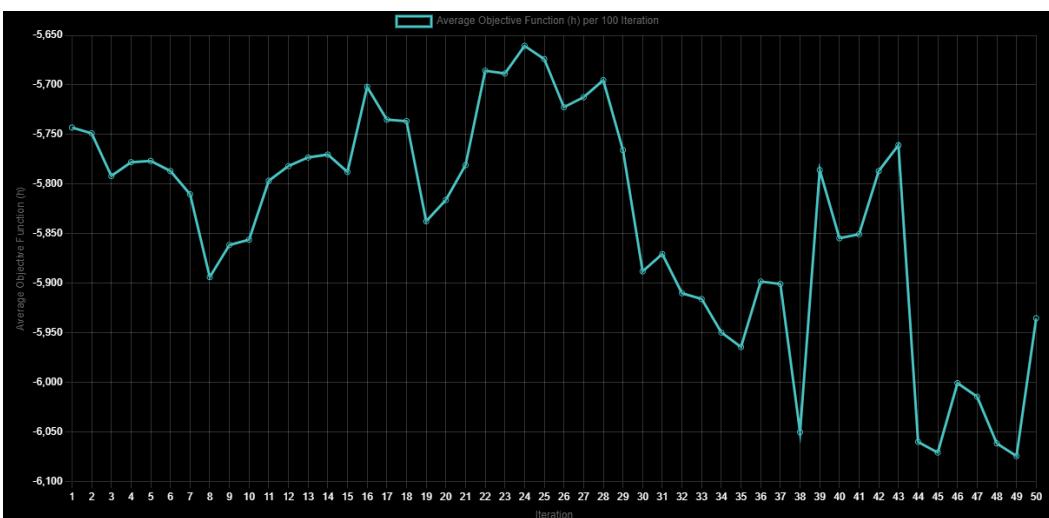
Initial State



Solved State



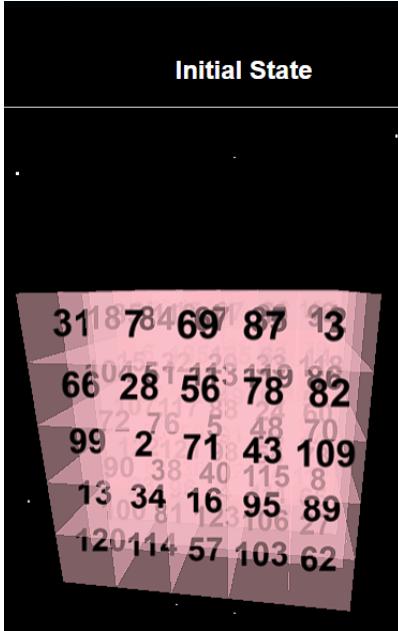
Plot



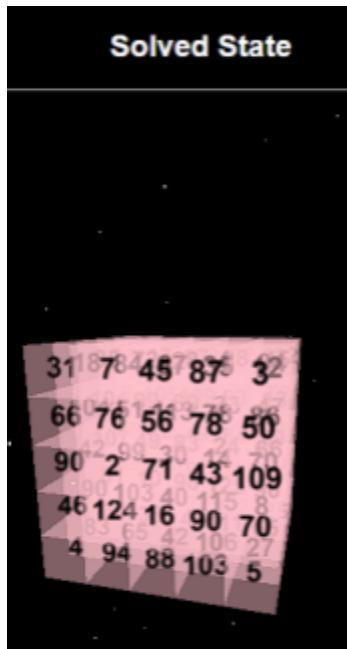
1.3.6.6 Percobaan 6

Iterasi = 50 populasi = 1000

Initial State



Solved State



Algorithm: Genetic Algorithm

H Before: 5744

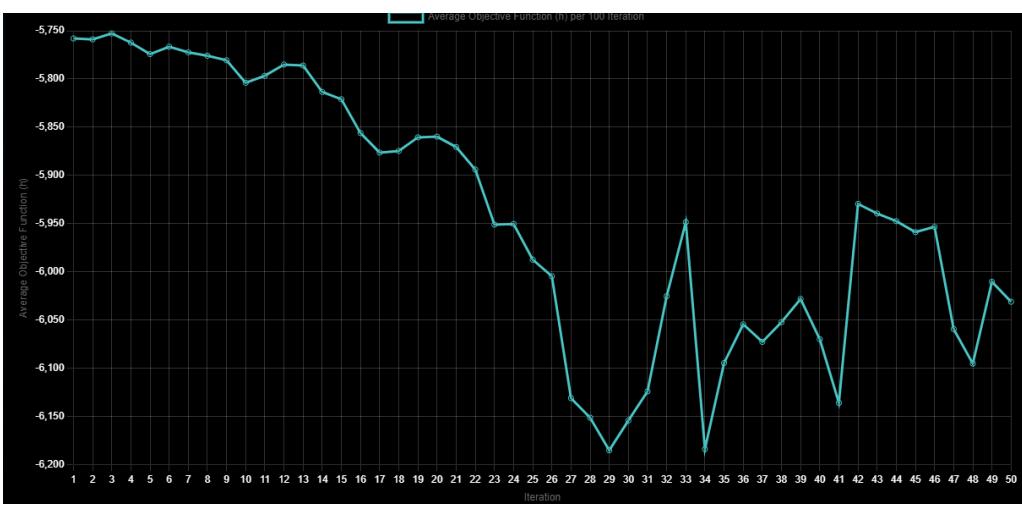
H After: 5421

Iterations: 50

Populations: 1000

Waktu Eksekusi: 2299 ms

Plot



1.3.7 Analisis

Tabel hasil awal dan hasil akhir nilai objective untuk tiap algoritma dan percobaan

Percobaan	HC				GA		SA $T = 30000$ $e = 0.99$
	Steep	Random Restart	Sideways	Stochastic	Iterasi beda	Populasi beda	
1	266/5944	153/6129	333/5682	623/6274	5522 /5744	6016 /5744	476/6017
2	399/5653	170/6469	204/6524	520/6274	6298 /5744	5698 /5744	361/6236
3	291/5974	210/6631	115/7270	281/6274	6262 /5744	5421 /5744	334/6136

Tabel durasi

Percobaan	HC				GA		SA $T = 30000$ $e = 0.99$
	Steep	Random Restart	Sideways	Stochastic	Iterasi beda	Populasi beda	
1	205 ms	10529 ms	275 ms	42 ms	22 ms	38 ms	27 ms
2	114 ms	2200 ms	288 ms	34 ms	112 ms	78 ms	32 ms
3	128 ms	5949 ms	161 ms	28 ms	711 ms	2299 ms	30 ms

Dari kedua tabel diatas, kita bisa membandingkan efisiensi dari algoritma-algoritma tersebut. Untuk membandingkan diperlukan rasio nilai h karena state awal tiap percobaan dan algoritma berbeda-beda. Rasio dibagi dengan waktu dalam satuan detik. Rumusnya seperti berikut:

$$\frac{Hawal - Hakhir}{Hawal}$$

$$t(s)$$

Tabel perbandingan rasio h terhadap waktu (detik) dengan pembulatan

Percobaan	HC				GA		SA T = 30000 e = 0.99
	Steep	Random Restart	Sideways	Stochastic	Iterasi beda	Populasi beda	
1	4.66	0.09	3.42	21.45	1.76	-1.25	34.11
2	8.15	0.44	3.36	26.97	-0.86	0.10	29.44
3	7.43	0.16	6.11	34.11	-0.13	0.02	31.52

Dilihat dari tabel data diatas meskipun hasil akhir nilai h untuk SA tidak seoptimal RR, jika dibandingkan rasio h terhadap waktu, SA lebih unggul dikarenakan waktu yang dibutuhkan untuk menyelesaikan random restart sangat lama jika dibandingkan waktu yang dibutuhkan oleh SA. Dari tabel rasio juga bisa dilihat konsistensi laju nya. Algoritma Hill Climb Sideway Move dan Random Restart (Percobaan 1 dan 2) cenderung lebih konsisten dibanding algoritma yang lain. Hill Climb Stochastic memiliki laju yang tidak konsisten karena cara kerja algoritma tersebut yang random. Selanjutnya, kami akan membahas setiap algoritma yang digunakan secara lebih rinci

Pertama untuk Steepest Ascent HC. Algoritma ini sangat cepat untuk gagal dan berhasil. Hal ini terjadi karena algoritma ini terminate begitu memasuki posisi tertinggi(dalam kasus ini h terendah) atau ketika kondisi datar (h tertinggi yang ditemukan sama dengan h saat ini) dari hasil.

Kedua, Sideway Step HC. Algoritma ini bisa dikatakan lebih baik dibanding Steepest Ascent karena algoritma ini tidak terminate ketika memasuki kondisi datar, jadi masih ada kemungkinan untuk menemukan suatu elemen dengan tetangga yang lebih rendah. Secara komputasi algoritma Sideway Step lebih lama dibanding Steepest Ascent karena masih harus mencari di kondisi datar, namun dengan ini bisa menghindari local optima lebih baik.

Ketiga, Random Restart HC. Algoritma ini pada dasarnya Steepest Ascent HC, namun diulang berkali kali hingga didapatkan nilai terbaik. Secara komputasi pasti lebih mahal dibanding kedua algoritma sebelum ini, namun eksplorasi dari algoritma ini lebih jauh mudah mendapat nilai optimum terutama dalam kasus diagonal magic cube 5x5 ini skalanya besar.

Keempat, Stochastic HC. Algoritma ini susah dibandingkan dengan algoritma lainnya karena hasilnya yang terlalu random, jadi menurut kami algoritma ini lebih ke arah keberuntungan (hoki).

Kelima ada Simulated Annealing. Algoritma ini memiliki kemungkinan mendapat optimum terbaik dari keenam algoritma(termasuk GA). Algoritma ini seperti Stochastic, namun dimodifikasi untuk bisa mundur ke nilai h yang lebih buruk agar bisa menghindari local optimum. Secara tidak langsung algoritma ini pasti lebih baik daripada Sideway Step karena tidak hanya flat, namun algoritma ini bisa turun untuk menghindari local optima hingga T mencapai 0 (kondisi terminate).

Terakhir GA. Berdasarkan hasil percobaan kami, algoritma ini kurang cocok untuk kasus ini. Algoritma ini memilih nilai fitness terbaik dari sejumlah initial state, kemudian individu terbaik memiliki peluang besar untuk terpilih sebagai parent yang akan mengalami proses crossover dan mutasi, sehingga terbentuk populasi baru. Namun, karena terdapat terlalu banyak kemungkinan dalam pembentukan populasi baru, hasilnya cenderung acak dan kurang terarah.Kami juga menganalisis pengaruh iterasi dan jumlah populasi pada algoritma ini. Didapati bahwa semakin banyak iterasi, semakin lama algoritma berproses untuk menemukan solusi yang lebih baik. Hal ini memberi kesempatan bagi populasi untuk berkembang melalui proses seleksi, crossover, dan mutasi. Dengan lebih banyak iterasi, GA memiliki peluang yang lebih besar untuk menjelajahi ruang solusi. Dengan populasi yang lebih besar, variasi genetik dalam populasi juga lebih tinggi. Ini memberi GA lebih banyak kandidat solusi dalam satu iterasi, sehingga meningkatkan peluang menemukan solusi yang lebih baik di setiap iterasi. Namun populasi yang banyak membuat waktu komputasi untuk setiap iterasi menjadi lebih lambat. Kami merasa agar algoritma ini mencapai hasil yang sangat optimal dan tidak terjebak di local optima, diperlukan banyak tuning terutama di bagian mutasi dan crossover. Tak hanya itu, komputasinya juga pasti sangat besar dibandingkan Simulated Annealing karena algoritma banyaknya step mulai dari membangkitkan banyak initial state, selection, cross dan mutation, dan setelah itu diulangi kembali.

KESIMPULAN DAN SARAN

Kesimpulannya, Algoritma local search terbaik untuk kasus ini adalah Simulated Annealing. Algoritma ini unggul dalam mencari nilai optimum karena keahliannya dalam menghindari local optima yang lebih baik daripada Sideway Step, dan eksplorasi elemen yang baik dari pemilihan elemennya yang random (Stochastic).

Saran dari kelompok kami adalah untuk memperbanyak percobaan pada setiap algoritma guna menganalisis persebaran data secara lebih komprehensif. Sebaiknya, semua percobaan dilakukan pada satu perangkat yang sama agar hasilnya tidak terpengaruh oleh perbedaan spesifikasi perangkat yang digunakan. Dengan demikian, kami dapat memastikan konsistensi dan validitas data yang diperoleh.

Selain itu, penting untuk mendokumentasikan setiap langkah percobaan dengan baik, termasuk parameter yang digunakan, hasil yang diperoleh, dan analisis yang dilakukan. Ini akan memudahkan evaluasi dan perbandingan hasil di masa depan.

PEMBAGIAN TUGAS

NIM	Tugas
13522129	3D cube,Steepest,Stochastic,Laporan
13522136	Random Restart, Sideways, Laporan
13522151	Simulated Annealing, Laporan
13522152	Genetic Algorithm, Laporan

REFERENSI

- [Features of the magic cube - Magisch vierkant](#)
- [Perfect Magic Cubes \(trump.de\)](#)
- [Magic cube - Wikipedia](#)

LAMPIRAN

Pranala GitHub: <https://github.com/miannetopokki/DiagonalMagicCubeSolver>