LAPORAN TUGAS BESAR
IF3170 INTELIGENSI BUATAN

Pencarian Solusi *Magic Cube* Dengan Algoritma *Local Search*

Disusun oleh:

Yusuf Ardian Sandi	13522015
Ibrahim Ihsan Rasyid	13522018
Zaki Yudhistira Candra	13522031
Tazkia Nizami	13522032

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

DAFTAR ISI

BAB I
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

Pada tugas ini, peserta kuliah akan menyelesaikan permasalahan Diagonal Magic Cube berukuran 5x5x5. 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). Khusus untuk genetic algorithm, boleh dilakukan penukaran posisi lebih dari 2 angka sekaligus dalam satu iterasi (tetap hanya menukar posisi 2 angka saja juga diperbolehkan).

Peserta kuliah akan diminta untuk mengimplementasikan rencana yang telah peserta kuliah buat pada Tugas Kecil 1. Berikut merupakan hal-hal yang perlu dilakukan oleh setiap kelompok:

- Implementasikan 3 algoritma local search dengan rincian sebagai berikut:
    - Salah satu algoritma hill-climbing
    - Simulated Annealing
    - Genetic Algorithm
- Lakukan eksperimen dengan skema sebagai berikut:
    - Jalankan setiap algoritma sebanyak 3 kali, kemudian catat beberapa hal berikut:
        - Berlaku untuk semua algoritma
            - State awal dan akhir dari kubus

        - Nilai objective function akhir yang dicapai
        - Plot nilai objective function terhadap banyak iterasi yang telah dilewati
        - Durasi proses pencarian
    - Berlaku hanya untuk Steepest Ascent Hill-Climbing dan Stochastic Hill-Climbing
        - Banyak iterasi hingga proses pencarian berhenti
    - Berlaku hanya untuk Hill-Climbing with Sideways Move
        - Banyak iterasi hingga proses pencarian berhenti
        - Note: Tambahkan parameter maximum sideways move, dimana ketika banyak sideways move yang dilakukan sudah mencapai maksimum, pencarian dihentikan
    - Berlaku hanya untuk Random Restart Hill-Climbing
        - Banyak restart
        - Banyak iterasi per restart
        - Note: Tambahkan parameter maximum restart, dimana ketika banyak restart sudah mencapai maksimum, pencarian dihentikan.
    - Berlaku hanya untuk Simulated Annealing
        - Plot $e^{\frac{\Delta E}{T}}$ terhadap banyak iterasi yang telah dilewati
        - Frekuensi ‘stuck’ di local optima
- Khusus untuk Genetic Algorithm, lakukan beberapa hal berikut:
    - Terdapat 2 parameter yang dapat diubah, yaitu jumlah populasi dan banyak iterasi.
    - Jadikan jumlah populasi sebagai kontrol, kemudian pilih 3 variasi banyak iterasi yang berbeda. Jalankan program sebanyak masing-masing 3 kali untuk setiap konfigurasi parameter.
    - Jadikan banyak iterasi sebagai kontrol, kemudian pilih 3 variasi jumlah populasi yang berbeda. Jalankan program sebanyak masing-masing 3 kali untuk setiap konfigurasi parameter.
    - Untuk setiap eksperimen, catat beberapa hal berikut:
        - State awal dan akhir dari kubus

        - Nilai objective function akhir yang dicapai
        - Plot nilai objective function terhadap banyak iterasi yang telah dilewati (Cukup plot nilai objective function maksimum dan rata-rata dari populasi terhadap banyak iterasi yang telah dilewati. Jika sudah terlanjur membuat plot untuk tiap individu pada populasi tidak menjadi masalah, silahkan diberikan keterangan saja maksud plotnya apa)
        - Jumlah populasi
        - Banyak iterasi
        - Durasi proses pencarian
- Lakukan analisis terhadap hasil eksperimen, berikut merupakan beberapa pertanyaan yang dapat menjadi acuan untuk analisis yang peserta kuliah lakukan (peserta kuliah boleh menambahkan beberapa pertanyaan tambahan jika dirasa perlu untuk dijelaskan):
    - Seberapa dekat tiap-tiap algoritma bisa mendekati global optima dan mengapa hasilnya demikian?
    - Bagaimana perbandingan hasil pencarian tiap-tiap algoritma dengan algoritma local search yang lain?
    - Bagaimana perbandingan durasi proses pencarian tiap algoritma relatif terhadap algoritma lainnya?
    - Seberapa konsisten hasil akhir yang didapatkan dari tiap-tiap eksperimen yang dilakukan?
    - Bagaimana pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada Genetic Algorithm?
    - dst…
- Program yang dibuat harus bisa memvisualisasikan state awal kubus, state akhir kubus, dan juga hasil eksperimennya (sesuaikan informasi hasil eksperimen yang ditampilkan dengan ketentuan yang telah dijelaskan di poin sebelum ini).
- Cara visualisasi dibebaskan kepada kelompok masing-masing selama seluruh angka pada kubus dan juga hasil eksperimen terlihat dengan jelas.
- Dibebaskan menggunakan bahasa pemrograman apapun.
- Diperbolehkan untuk menggunakan heuristik yang peserta kuliah buat sendiri atau dari referensi lain untuk optimasi pencarian solusi, asalkan masih dalam lingkup local search. Jangan lupa jelaskan heuristik yang peserta kuliah pakai di laporan.

BAB II
PEMBAHASAN

1. Pemilihan Objective Function

Fungsi objektif adalah komponen penting dalam optimisasi dan algoritma pencarian solusi yang bertujuan untuk mengevaluasi seberapa baik suatu solusi memenuhi tujuan yang diinginkan. Dalam konteks algoritma pencarian atau optimisasi, fungsi objektif memberikan nilai numerik pada setiap solusi potensial, yang kemudian digunakan untuk membandingkan solusi tersebut satu sama lain. Tujuannya adalah untuk memaksimumkan atau meminimumkan nilai fungsi objektif, tergantung pada jenis masalah yang sedang diselesaikan.

Dalam masalah Magic Cube 5x5x5, tujuannya adalah membuat kubus di mana setiap baris, kolom, tiang (vertikal), diagonal bidang (diagonal pada tiap sisi 2D), dan diagonal ruang (diagonal yang menghubungkan sudut-sudut kubus 3D) memiliki jumlah angka yang sama, yaitu magic number. Magic number ini biasanya dihitung sebagai:

$$magic\ number\ =\ n(n^3\ +\ 1)\ \div\ 2$$

n adalah panjang sisi kubus, dalam hal ini 5. Sehingga diperoleh magic number = 315.

Dalam program penyelesaian permasalahan Magic Cube, fungsi objektif digunakan untuk memaksimalkan total banyak segmen kubus yang jumlahnya sama dengan magic number. Pendefinisian fungsi objektif Anda adalah:

$$Fungsi\ Objektif\ =\ Total\ banyak\ Segmen\ yang\ berjumlah\ sama\ dengan\ magic\ number$$

Banyak segmen yang dibentuk oleh unsur baris, kolom, pilar, diagonal, dan/atau lain-lain ialah sejumlah 109. Oleh karena itu dapat diambil suatu kesimpulan bahwa 0 merupakan state value terburuk dan 109 merupakan state value terbaik (global optima).

2. Penjelasan Implementasi Algoritma Local Search

Dalam implementasi algoritma local search, kami menggunakan bahasa Java dengan mengaplikasikan polymorphism. Berikut kelas/interface yang kami gunakan:

a. Interface IAlgoritma

Java Interface untuk merepresentasikan algoritma yang akan menyelesaikan magic cube.

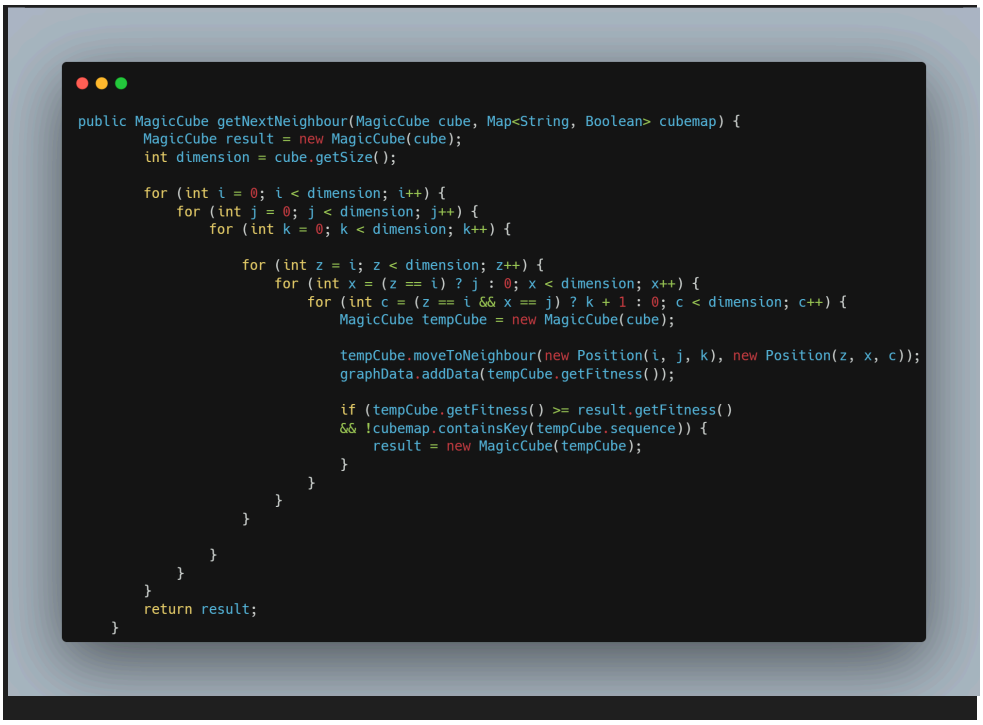
Deskripsi Fungsi	Source Code

Fungsi untuk menyelesaikan magic cube dan mengembalikan objek cube yang telah diselesaikan.	`public interface IAlgorithm {`
	`    MagicCube getSolvedCube(MagicCube cube);`
	
	`    GraphData getGraphData();`
	`}`

Mendapatkan graf data yang berhubungan dengan algoritma tertentu.	

b. Kelas HillClimbingSideMove

HillClimbingSideMove merupakan implementasi interfaceIAlgorithm yang menggunakan algoritma Steepest Ascent dengan gerakan samping (sideways move) untuk memecahkan MagicCube. Ia berupaya menemukan solusi dengan berpindah secara berulang ke tetangga dengan state value yang sama atau lebih baik, yang memungkinkan sejumlah gerakan samping terbatas (bergerak ke state dengan state value yang sama) sebelum berakhir.

Deskripsi Fungsi	Source Code
Inisiasi objek hill-climbing	
Fungsi untuk memperoleh tetangga selanjutnya	

```java
public HillClimbingSideMove(int max_side_moves) {
        this.max_side_moves = max_side_moves;
        this.graphData = new GraphData(false);
    }
```

```java
public MagicCube getNextNeighbour(MagicCube cube, Map<String, Boolean> cubemap) {
        MagicCube result = new MagicCube(cube);
        int dimension = cube.getSize();

        for (int i = 0; i < dimension; i++) {
            for (int j = 0; j < dimension; j++) {
                for (int k = 0; k < dimension; k++) {

                    for (int z = i; z < dimension; z++) {
                        for (int x = (z == i) ? j : 0; x < dimension; x++) {
                            for (int c = (z == i && x == j) ? k + 1 : 0; c < dimension; c++) {
                                MagicCube tempCube = new MagicCube(cube);

                                tempCube.moveToNeighbour(new Position(i, j, k), new Position(z, x, c));
                                graphData.addData(tempCube.getFitness());

                                if (tempCube.getFitness() >= result.getFitness()
                                && !cubemap.containsKey(tempCube.sequence)) {
                                    result = new MagicCube(tempCube);
                                }
                            }
                        }
                    }

                }
            }
        }
        return result;
    }
```

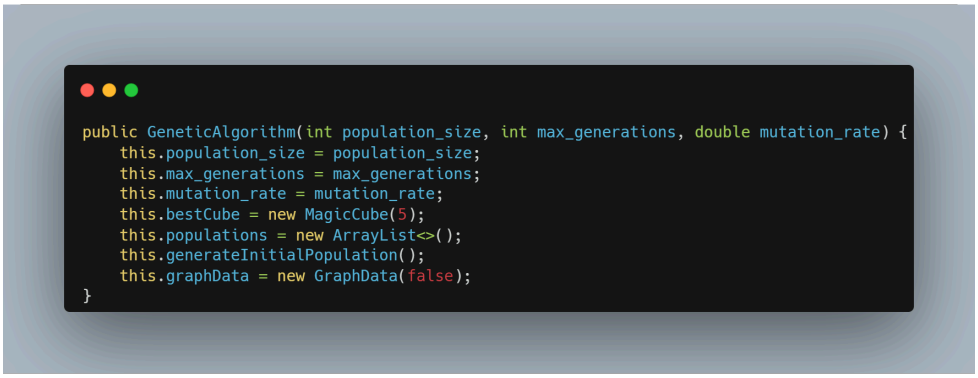
Fungsi utama dari hill climbing algorithm with sideway moves. Fungsi akan menerima kubus dan melakukan while loop hingga diperoleh next neighbour berupa dirinya sendiri atau ketika jumlah iterasi sideway moves yang dilakukan sudah melebihi batas maksimal.	

```
public MagicCube getSolvedCube(MagicCube cube) {
        Map<String, Boolean> cubemap = new HashMap<>();
        graphData.addData(cube.getFitness());

        MagicCube prev = new MagicCube(cube);
        cubemap.put(cube.sequence, true);

        int side_moves = 0; // Counter untuk menghitung sideways moves

        while (true) {
            graphData.finishIteration();
            MagicCube next = new MagicCube(getNextNeighbour(prev, cubemap));
            graphData.addData(next.getFitness());

            if (next.getFitness() > prev.getFitness()) {
                    side_moves = 0; // Reset jika ada peningkatan fitness
                } else if (next.getFitness() == prev.getFitness()) {
                        side_moves++; // Increment jika fitness sama
                        if (side_moves >= max_side_moves) {
                                return prev; // Hentikan pencarian
                            }
            }

            if (next.sequence.equals(prev.sequence)) {
                return next;
            }

            cubemap.put(next.sequence, true);
            prev = new MagicCube(next);
        }
    }
```

c. Kelas GeneticAlgorithm

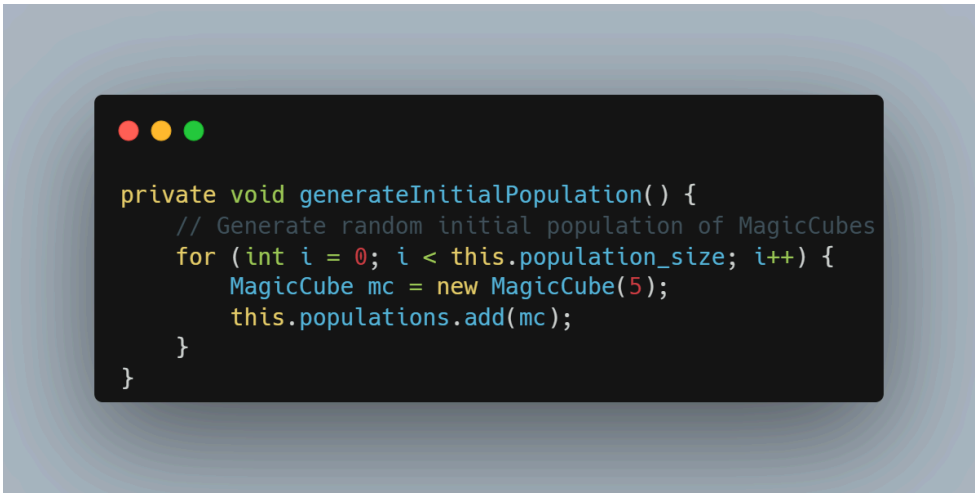
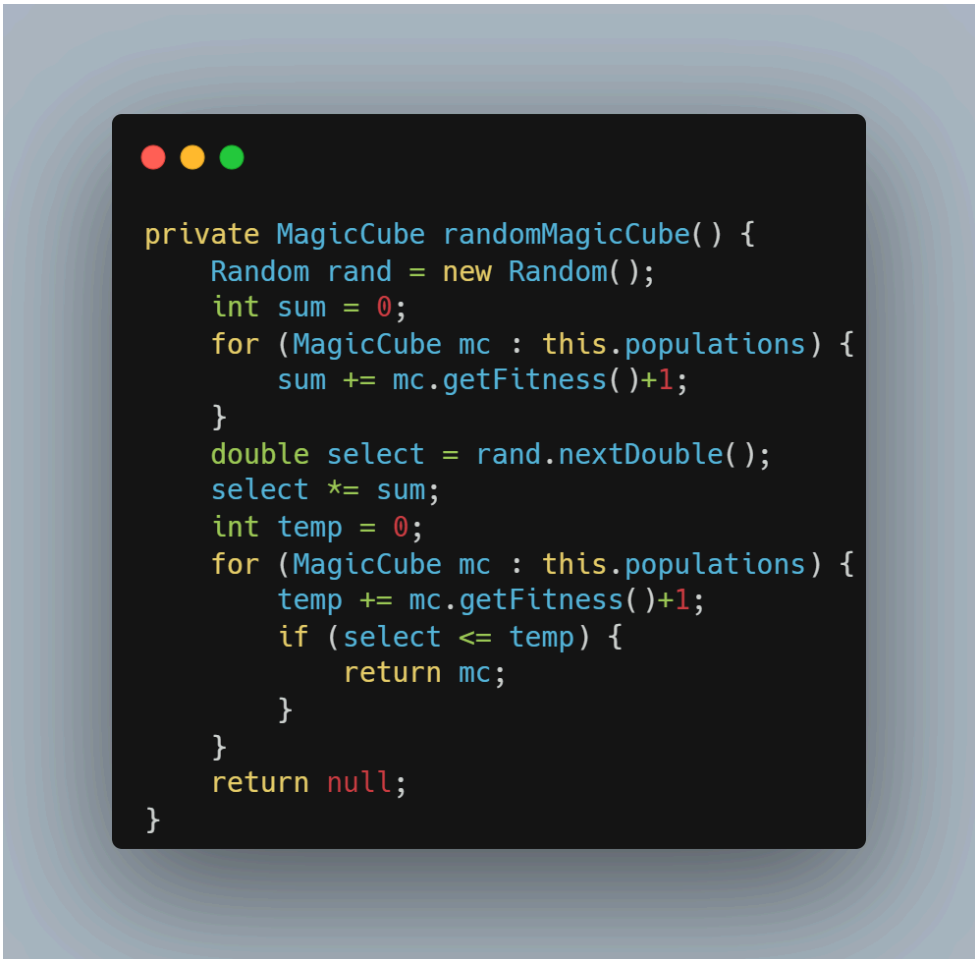
Kelas GeneticAlgorithm mengimplementasikan interface IAlgorithm dengan algoritma genetik untuk memecahkan masalah MagicCube. Kelas ini mengelola populasi MagicCube dan mengembangkannya dari generasi ke generasi untuk menemukan solusi terbaik. Kelas ini mencakup metode untuk menghasilkan populasi awal, melakukan operasi persilangan dan mutasi, serta memilih solusi terbaik.

Deskripsi Fungsi	**Source Code**

Inisiasi objek GeneticAlgorithm	
Fungsi utama untuk mencari solusi menggunakan genetic algorithm	

```java
public GeneticAlgorithm(int population_size, int max_generations, double mutation_rate) {
    this.population_size = population_size;
    this.max_generations = max_generations;
    this.mutation_rate = mutation_rate;
    this.bestCube = new MagicCube(5);
    this.populations = new ArrayList<>();
    this.generateInitialPopulation();
    this.graphData = new GraphData(false);
}
```

```java
public MagicCube getSolvedCube(MagicCube cube) {
    // Find best solution from population
    this.populations.set(0, cube);
    int i = 0;
    int best_eval;
    do {
        List<MagicCube> new_mcs = new ArrayList<>();
        for (int j = 0; j < this.population_size / 2; j++) {
            MagicCube cube1 = this.randomMagicCube();
            MagicCube cube2 = this.randomMagicCube();
            List<MagicCube> children = this.crossover(cube1, cube2);
            for (MagicCube child : children) {
                MagicCube mutated_child = this.mutate(child);
                graphData.addData(mutated_child.getFitness());
                new_mcs.add(mutated_child);
            }
        }
        this.populations = new_mcs;
        i++;
        best_eval = this.getBestFitness();
        graphData.finishIteration();
        this.mutation_rate -= 0.005;
    } while (i < this.max_generations && best_eval < 90);
    return this.bestCube;
}
```

Fungsi untuk menghasilkan populasi awal	
Fungsi untuk melakukan random selection dan mengembalikan MagicCube yang terpilih	

```java
private void generateInitialPopulation() {
    // Generate random initial population of MagicCubes
    for (int i = 0; i < this.population_size; i++) {
        MagicCube mc = new MagicCube(5);
        this.populations.add(mc);
    }
}
```

```java
private MagicCube randomMagicCube() {
    Random rand = new Random();
    int sum = 0;
    for (MagicCube mc : this.populations) {
        sum += mc.getFitness()+1;
    }
    double select = rand.nextDouble();
    select *= sum;
    int temp = 0;
    for (MagicCube mc : this.populations) {
        temp += mc.getFitness()+1;
        if (select <= temp) {
            return mc;
        }
    }
    return null;
}
```

Fungsi untuk mentranslasikan MagicCube ke array	
Fungsi untuk mentranslasikan array ke MagicCube	

```java
private int[] translateCubetoArray(MagicCube mc) {
    int size = mc.getSize();
    int[] ret = new int[125];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            for (int k = 0; k < size; k++) {
                Position pos = new Position(i, j, k);
                int index = i * 25 + j * 5 + k;
                ret[index] = mc.getCubeElement(pos);
            }
        }
    }
    return ret;
}
```

```java
private MagicCube translateArraytoCube(int[] arr) {
    MagicCube ret = new MagicCube(5);
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            for (int k = 0; k < 5; k++) {
                Position pos = new Position(i, j, k);
                int index = i * 25 + j * 5 + k;
                ret.setCubeElement(pos, arr[index]);
            }
        }
    }
    return ret;
}
```

Fungsi untuk melakukan crossover	

```java
private List<MagicCube> crossover(MagicCube parent1, MagicCube parent2) {
    // Combine two MagicCubes to create a new MagicCube
    Random rand = new Random();
    int[] arr_parent1 = this.translateCubetoArray(parent1);
    int[] arr_parent2 = this.translateCubetoArray(parent2);

    int[] child1 = new int[125];
    int[] child2 = new int[125];
    Arrays.fill(child1, -1);
    Arrays.fill(child2, -1);

    int crossover_point1 = rand.nextInt(0, 124);
    int crossover_point2 = rand.nextInt(0, 124);
    if (crossover_point1 > crossover_point2) {
        int temp = crossover_point1;
        crossover_point1 = crossover_point2;
        crossover_point2 = temp;
    }

    for (int i = crossover_point1; i <= crossover_point2; i++) {
        child1[i] = arr_parent1[i];
        child2[i] = arr_parent2[i];
    }

    Map<Integer, Integer> mapping1 = new HashMap<>();
    Map<Integer, Integer> mapping2 = new HashMap<>();
    for (int i = crossover_point1; i <= crossover_point2; i++) {
        mapping1.put(arr_parent2[i], arr_parent1[i]);
        mapping2.put(arr_parent1[i], arr_parent2[i]);
    }

    for (int i = 0; i < 125; i++) {
        if (child1[i] == -1) {
            int gene = arr_parent2[i];
            while (mapping1.containsKey(gene)) {
                gene = mapping1.get(gene);
            }
            child1[i] = gene;
        }
    }

    for (int i = 0; i < 125; i++) {
        if (child2[i] == -1) {
            int gene = arr_parent1[i];
            while (mapping2.containsKey(gene)) {
                gene = mapping2.get(gene);
            }
            child2[i] = gene;
        }
    }

    MagicCube mc_child1 = this.translateArraytoCube(arr_parent1);
    MagicCube mc_child2 = this.translateArraytoCube(arr_parent2);

    List<MagicCube> ret = new ArrayList<>();
    if (mc_child1.getFitness() > parent1.getFitness()) {
        ret.add(mc_child1);
    } else {
        ret.add(parent1);
    }
    if (mc_child2.getFitness() > parent2.getFitness()) {
        ret.add(mc_child2);
    } else {
        ret.add(parent2);
    }

    return ret;
}
```

Fungsi untuk melakukan mutate	
Fungsi untuk menentukan nilai fitness function terbaik dari populasi saat itu	

```java
private MagicCube mutate(MagicCube cube) {
    // Mutate the cube based on mutation rate
    Random rand = new Random();

    int[] arr_chromosome = this.translateCubetoArray(cube);

    if (rand.nextDouble() < mutation_rate) {
        int pos1 = rand.nextInt(arr_chromosome.length);
        int pos2 = rand.nextInt(arr_chromosome.length);
        while (pos1 == pos2) {
            pos2 = rand.nextInt(arr_chromosome.length);
        }
        int temp = arr_chromosome[pos1];
        arr_chromosome[pos1] = arr_chromosome[pos2];
        arr_chromosome[pos2] = temp;
    }
    MagicCube mutated = this.translateArraytoCube(arr_chromosome);
    return mutated;
}
```

```java
private int getBestFitness() {
    int ret = 0;
    for (MagicCube mc : this.populations) {
        if (mc.getFitness() > ret) {
            ret = mc.getFitness();
            this.bestCube.copyFrom(mc);
        }
    }
    return ret;
}
```

Fungsi-fungsi getter dan setter pada kelas	

d. Kelas SimulatedAnnealing

Kelas untuk implementasi algoritma simulated annealing untuk menyelesaikan magic cube. Menerima masukkan initial temperatur yang akan diturunkan secara perlahan sekaligus menjelajahi tetangga magic cube secara acak. Kelas ini juga menyimpan riwayat probabilitas yang dihasilkan untuk ditampilkan ke grafik. Selain itu, algoritma ini juga terdapat sebuah modifikasi (heuristic) dimana daripada mengembalikan solusi berupa current state, algoritma ini akan mengembalikan solusi terbaik yang pernah dihasilkan saja. Threshold peluang yang digunakan pada algoritma ini adalah 0,95 dengan suhu awal dan *cooling rate* secara berturut-turut adalah 10 dan 0,000001.

Deskripsi Fungsi	**Source Code**

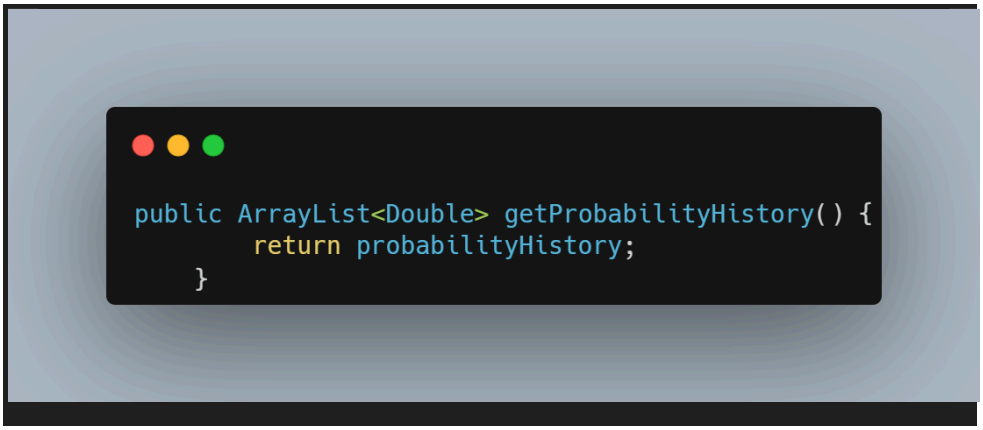
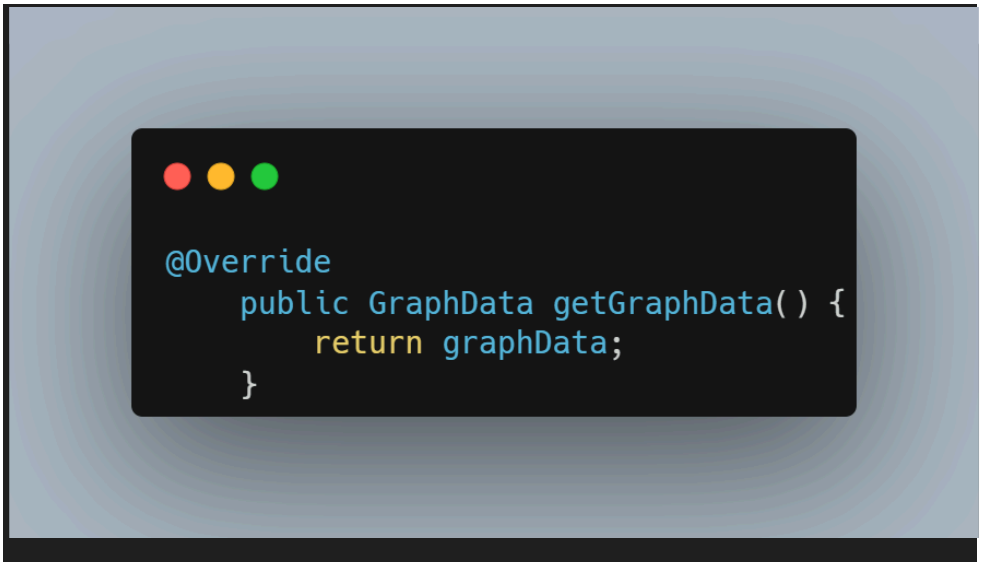
Menghasilkan tetangga acak dari Magic Cube yang diberikan dengan menukar dua posisi acak.	

```java
public MagicCube getRandomNeighbour(MagicCube cube) {
        MagicCube neighbor = new MagicCube(cube);


        // Get random two Positions
        Random rand = new Random();
        int x1 = rand.nextInt(cube.getSize());
        int y1 = rand.nextInt(cube.getSize());
        int z1 = rand.nextInt(cube.getSize());
        Position el1 = new Position(x1, y1, z1);


        int x2 = rand.nextInt(cube.getSize());
        int y2 = rand.nextInt(cube.getSize());
        int z2 = rand.nextInt(cube.getSize());
        Position el2 = new Position(x2, y2, z2);


        // Swap the two elements
        neighbor.moveToNeighbour(el1, el2);
        return neighbor;
    }
```

Menyelesaikan Magic Cube yang diberikan. Menghapus riwayat probabilitas sebelumnya dan menambahkannya dengan yang baru dari algoritma yang telah dijalankan.	
Menghitung probabilitas perpindahan ke solusi tetangga dalam algoritma simulated annealing.	

```java
@Override
    public MagicCube getSolvedCube(MagicCube cube) {
        double temperature = initial_temperature;
        MagicCube currentCube = new MagicCube(cube);
        MagicCube bestCube = new MagicCube(cube);
        int iteration = 0;


        probabilityHistory.clear();


        int localOptimumFrequency = 0;


        while (temperature > 1) {
            iteration++;


            MagicCube neighbour = getRandomNeighbour(currentCube);
            int currentFitness = currentCube.getFitness();
            int neighbourFitness = neighbour.getFitness();


            double accProbability = acceptanceProbability(currentFitness, neighbourFitness, temperature);


            graphData.addData(neighbourFitness, temperature);
            // probabilityHistory.add(accProbability);

            // Hitung frekuensi stuck local optimum
            if (!(neighbourFitness > currentFitness)) {
                localOptimumFrequency++;
            }


            if (accProbability > 0.95) {
                graphData.finishIteration();
                currentCube = new MagicCube(neighbour);
            }


            if (currentFitness > bestCube.getFitness()) {
                bestCube = new MagicCube(currentCube);
                // System.out.println("Best Fitness: " + bestCube.getFitness());
                // System.out.println("Temperature: " + temperature);
            }
            temperature *= 1 - cooling_rate;
        }


        System.out.println("Number of iterations: " + iteration);
        System.out.println("Local Optimum Stuck Frequency: " + localOptimumFrequency);
        return bestCube;
    }
```

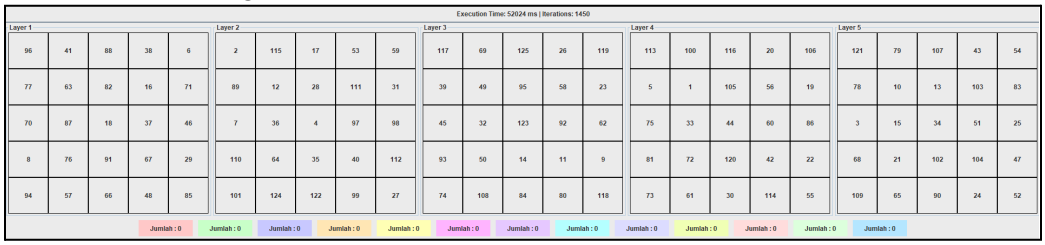
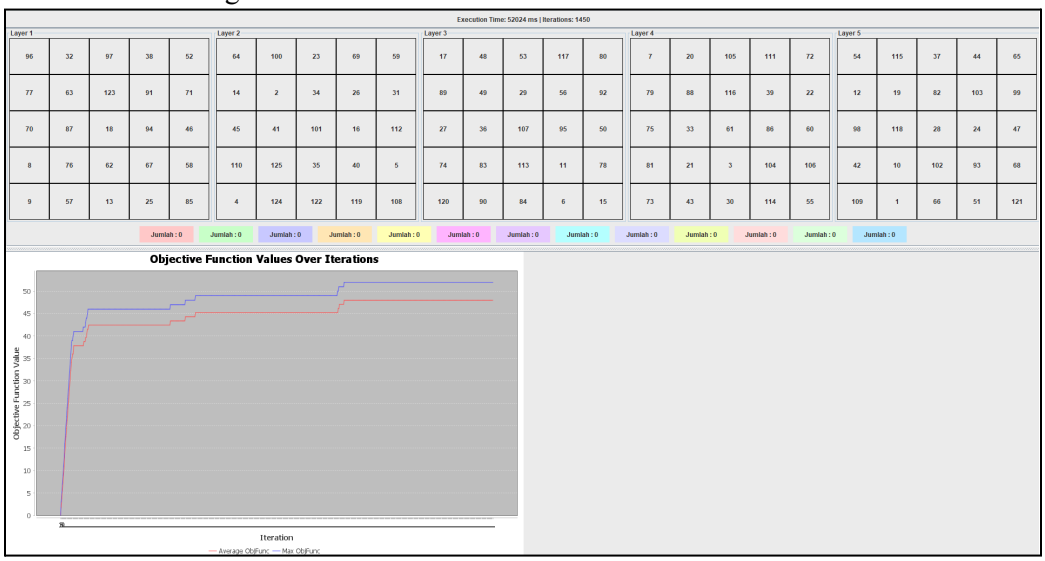
```java
private double acceptanceProbability(int currentFitness, int neighbourFitness, double temperature) {
        if (neighbourFitness > currentFitness) {
            // counter++;
            return 1.0;
        }
        return Math.exp((neighbourFitness - currentFitness) / temperature);
    }
```

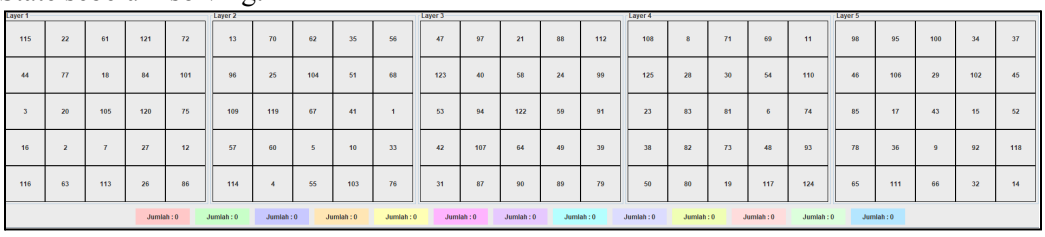
Mengambil riwayat probabilitas yang terekam selama proses algoritma.	
Mendapatkan data graf	

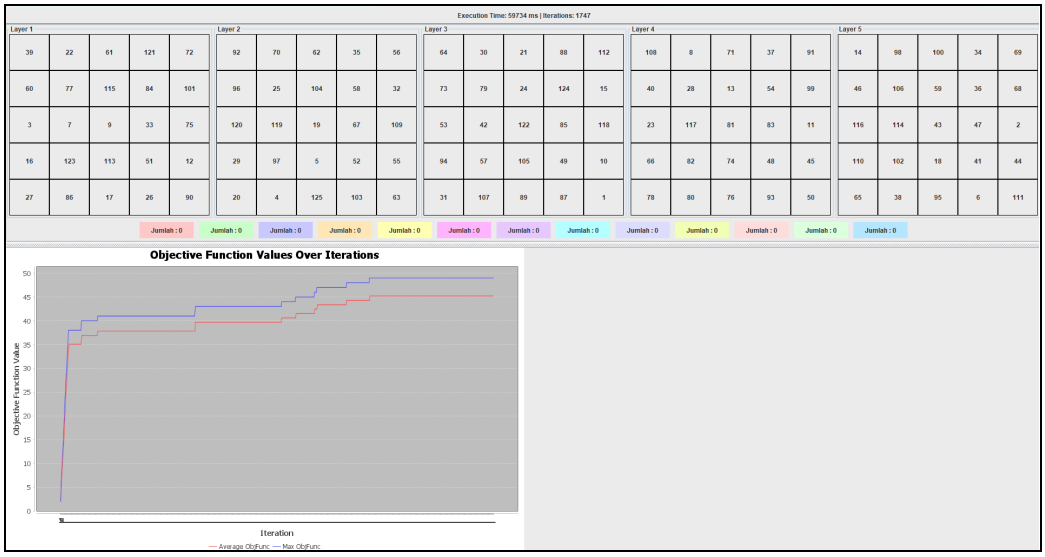
3. Hasil Eksperimen dan Analisis

a. Algoritma Hill Climbing With Sideways Move

Percobaan	Hasil

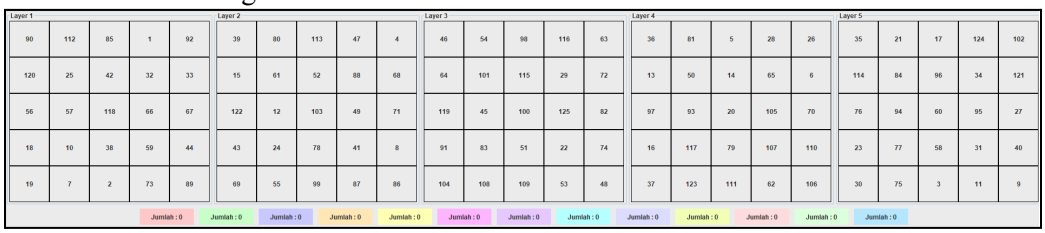
1	State sebelum solving:
	 
	State setelah solving:
	 
	
	Keterangan:
	`Fitness before solving: 0`
	`Fitness after solving: 52`
	`iterasi: 1450`
	`Waktu eksekusi: 52024 ms`

2	State sebelum solving:  State setelah solving:

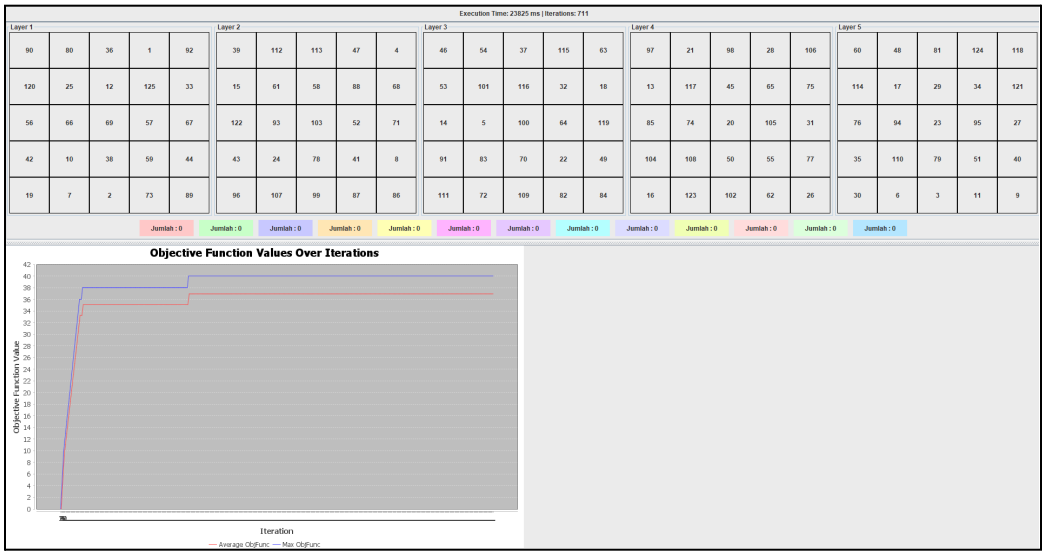
<table>
<tr>
<td></td>
<td>


Keterangan:

</td>
</tr>
<tr>
<td>3</td>
<td>
State sebelum solving:



State setelah solving:

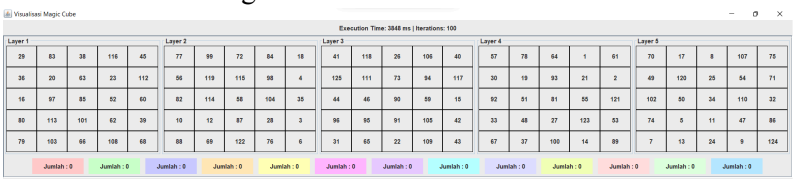
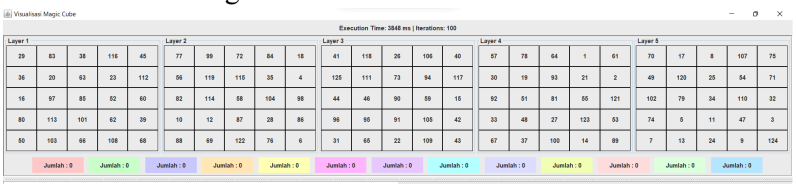
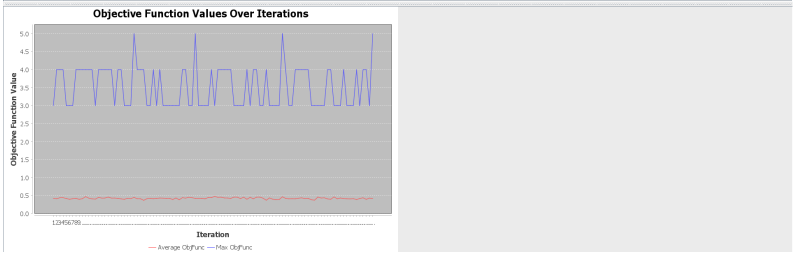


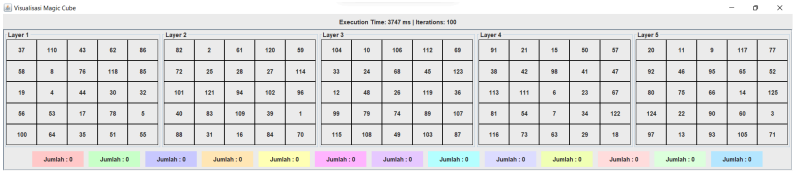
</td>
</tr>
</table>

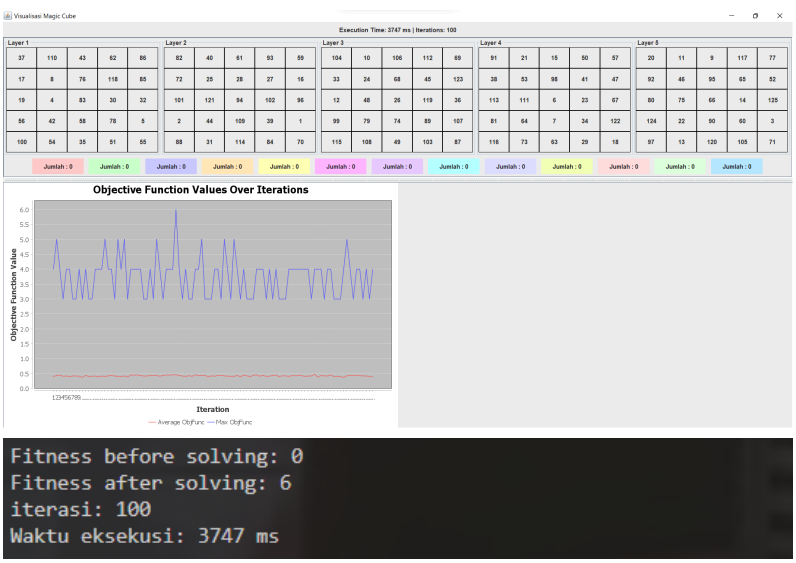
	Keterangan:
	`Fitness before solving: 0`
	`Fitness after solving: 40`
	`iterasi: 711`
	`Waktu eksekusi: 23825 ms`

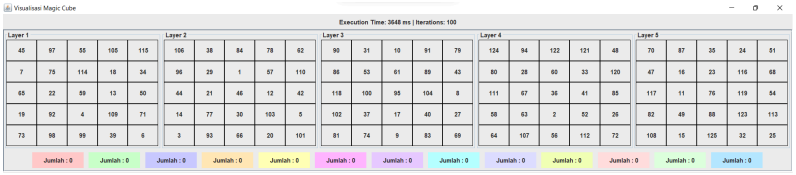
b. Algoritma Genetic

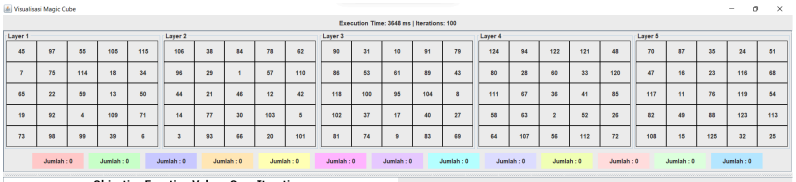
Variabel Kontrol	Keterangan	Percobaan	Hasil

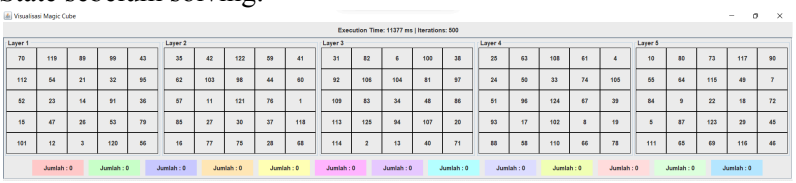
Populasi = 1000	Iterasi = 100	a	State sebelum solving:
			 
			State setelah solving:
			 
			 
			
			`Fitness before solving: 0`
			`Fitness after solving: 5`
			`iterasi: 100`
			`Waktu eksekusi: 3848 ms`

		b	State sebelum solving:  State setelah solving:

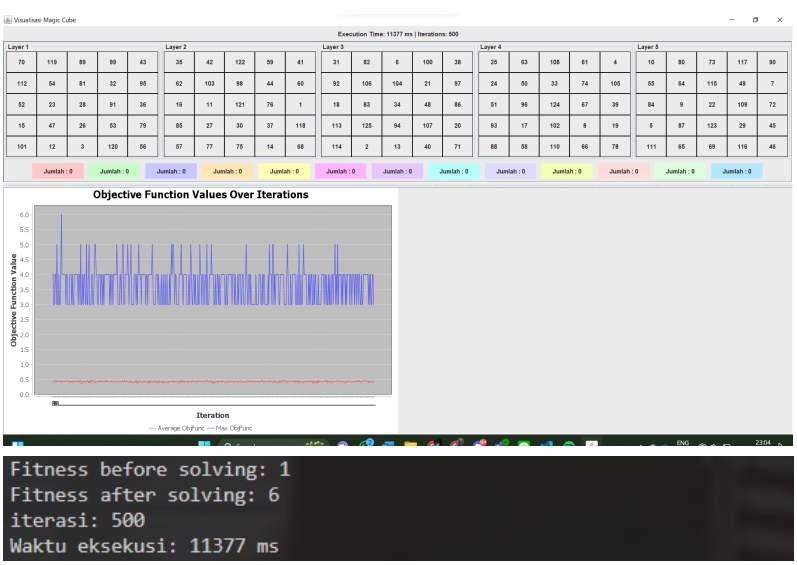
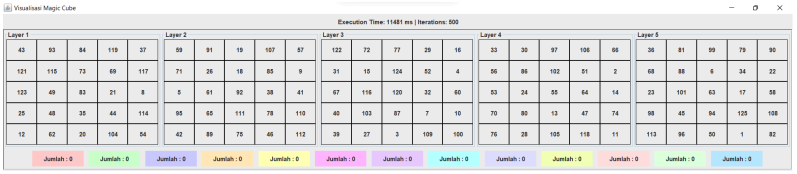
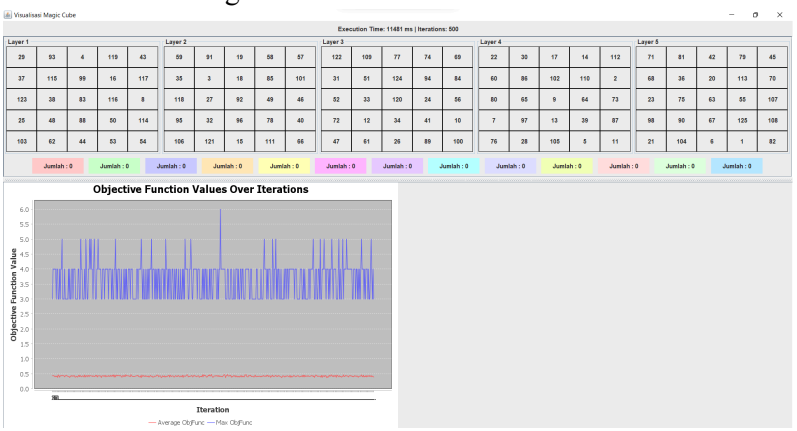
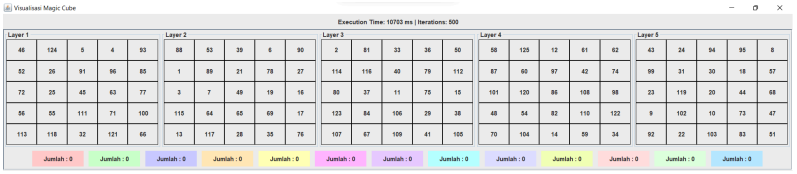
<table>
<tr>
<td rowspan="3"></td>
<td rowspan="2"></td>
<td></td>
<td> 

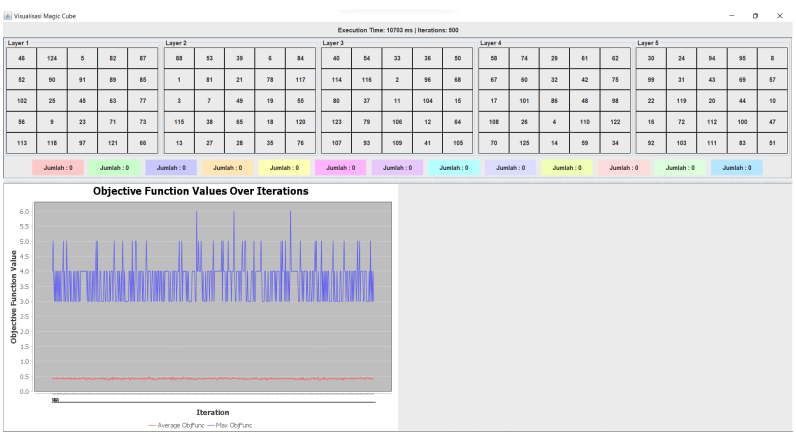
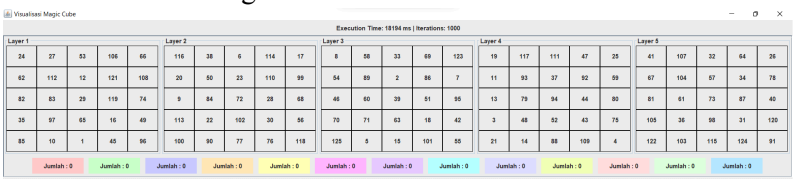
</td>
</tr>
<tr>
<td>c</td>
<td>State sebelum solving:
 

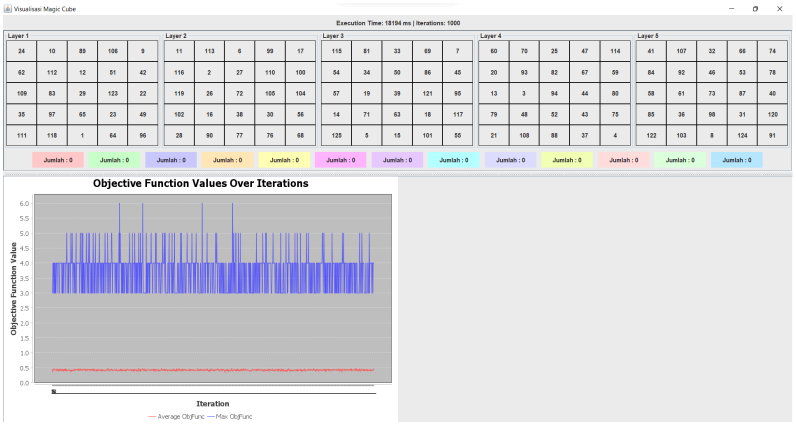
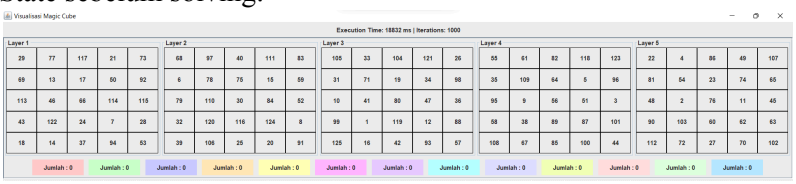
State setelah solving:
 


</td>
</tr>
<tr>
<td>Iterasi = 500</td>
<td>a</td>
<td>State sebelum solving:
 

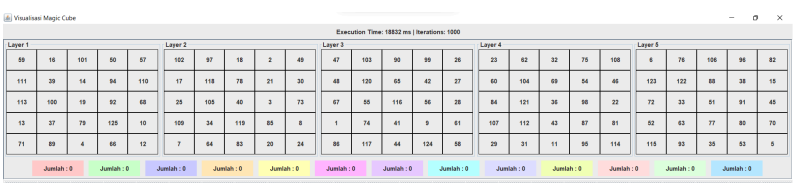
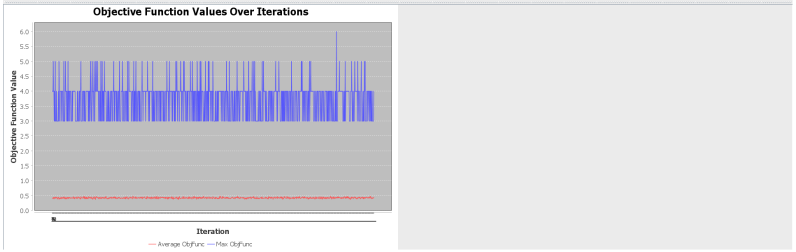
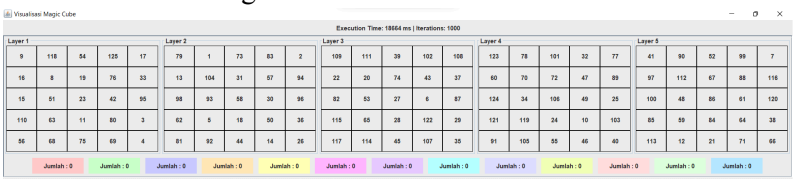
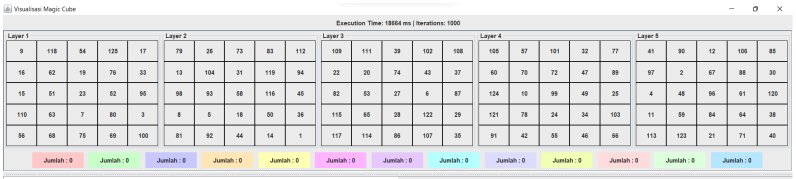
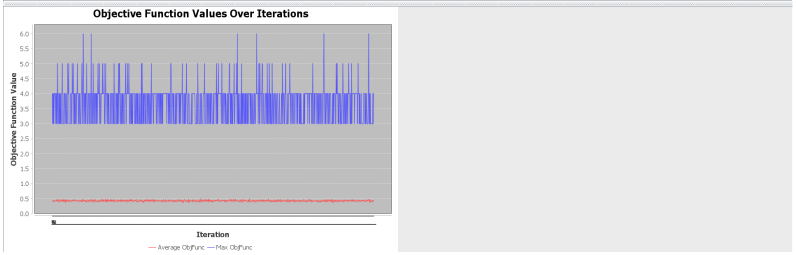
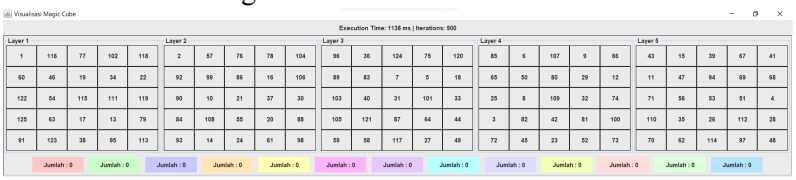
State setelah solving:</td>
</tr>
</table>

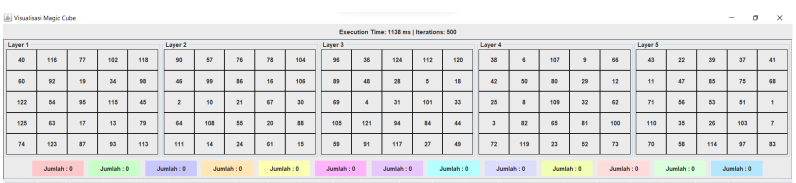
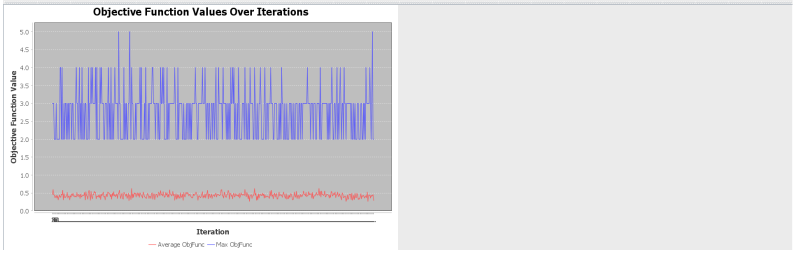
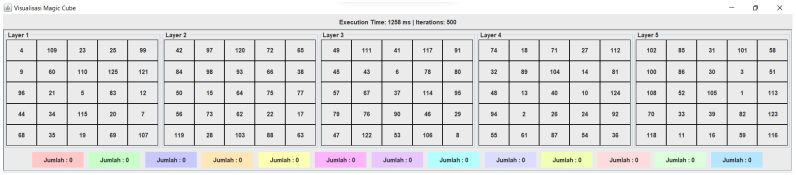
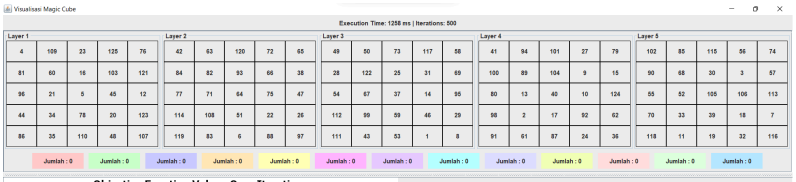
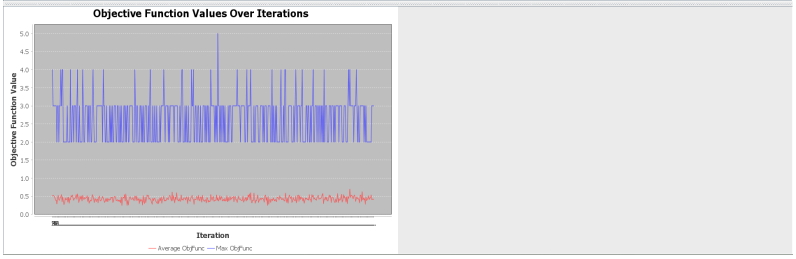
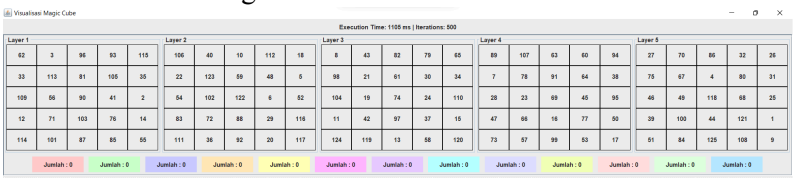
			 Fitness before solving: 1 Fitness after solving: 6 iterasi: 500 Waktu eksekusi: 11377 ms
		b	State sebelum solving:  State setelah solving:  Fitness before solving: 1 Fitness after solving: 6 iterasi: 500 Waktu eksekusi: 11481 ms
		c	State sebelum solving:  State setelah solving:

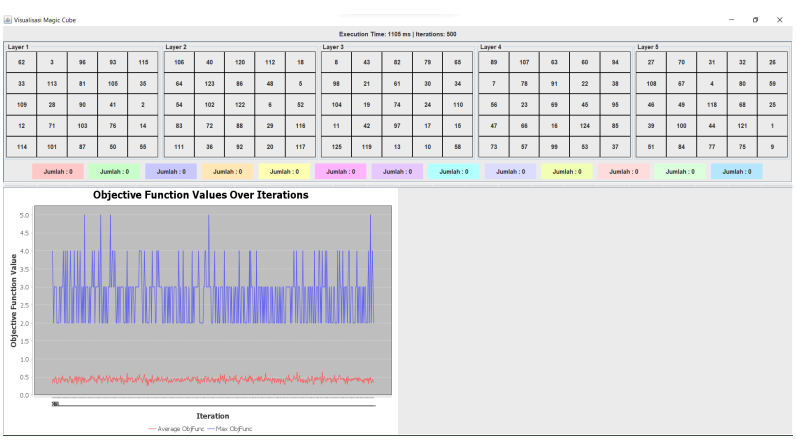
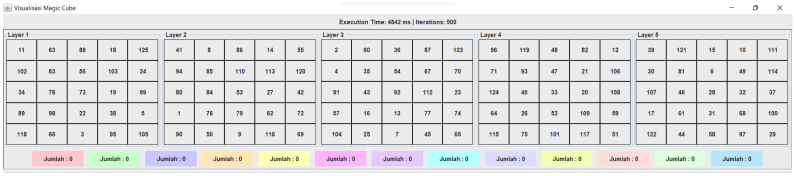
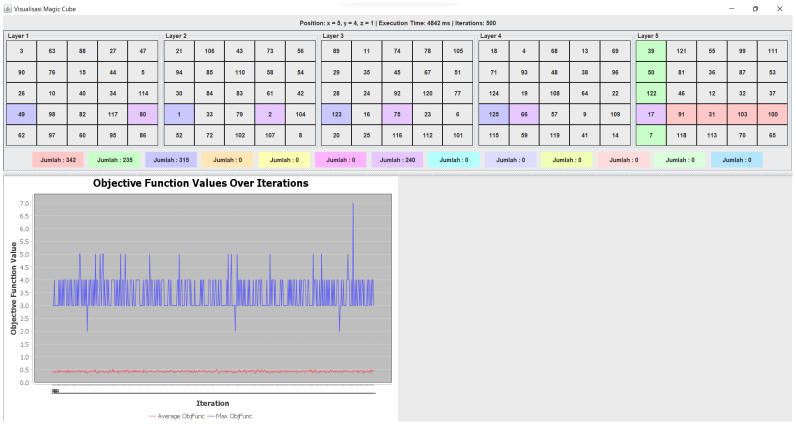
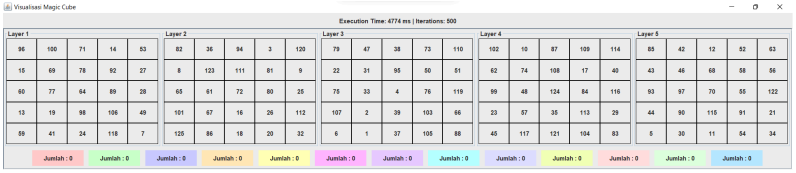
<table>
<tr>
<td rowspan="3"></td>
<td></td>
<td></td>
<td> 
</td>
</tr>
<tr>
<td>Iterasi = 1000</td>
<td>a</td>
<td>State sebelum solving:
 

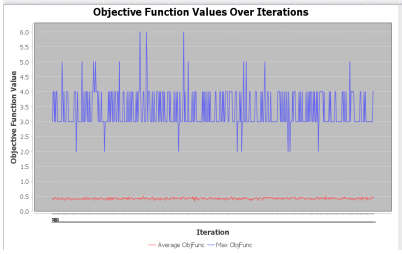
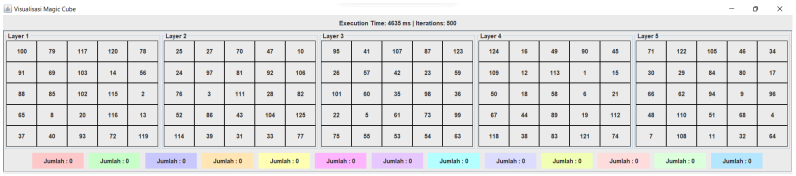
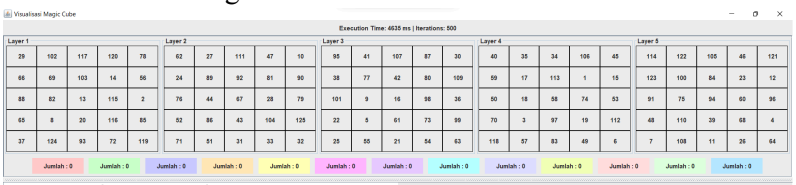
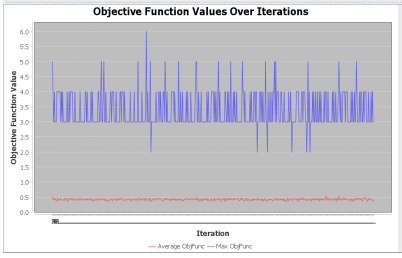
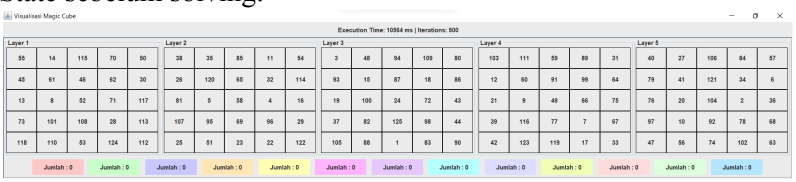
State setelah solving:
 
</td>
</tr>
<tr>
<td>b</td>
<td>State sebelum solving:
 

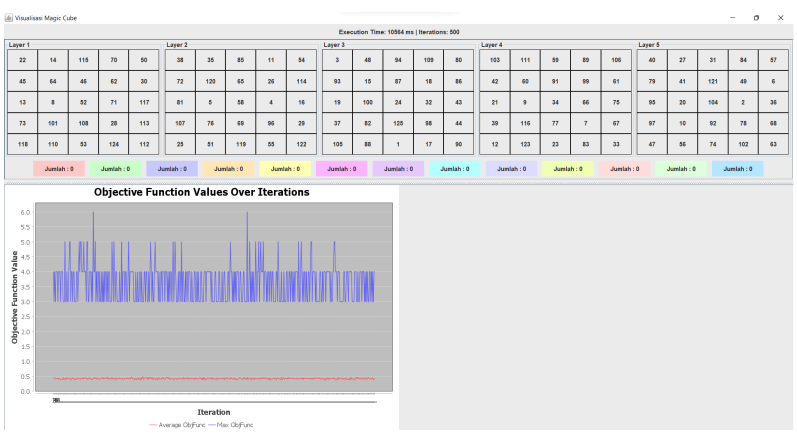
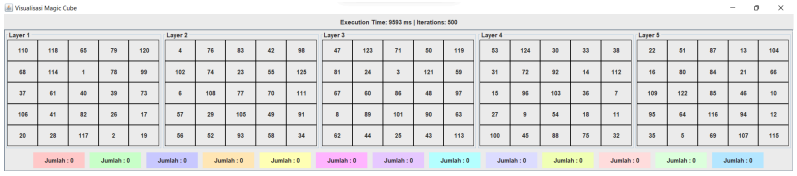
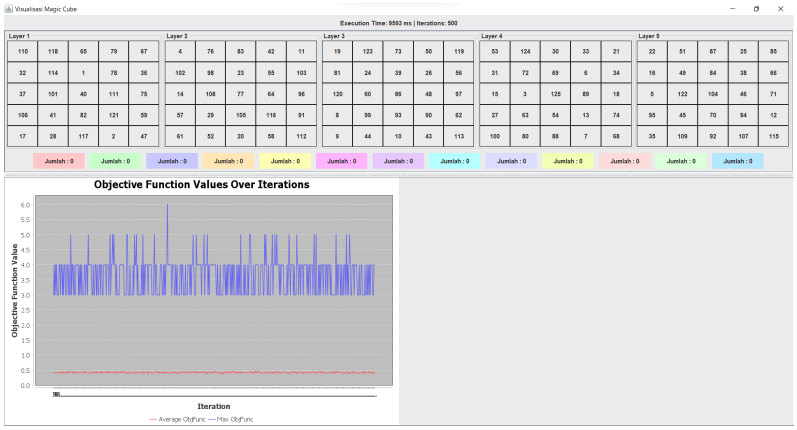
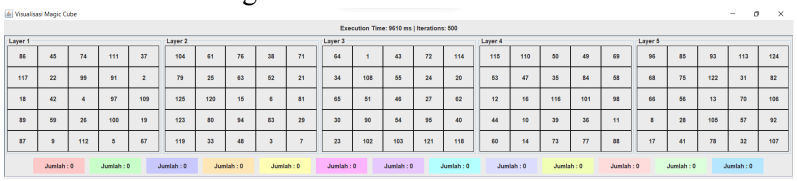
State setelah solving:</td>
</tr>
</table>

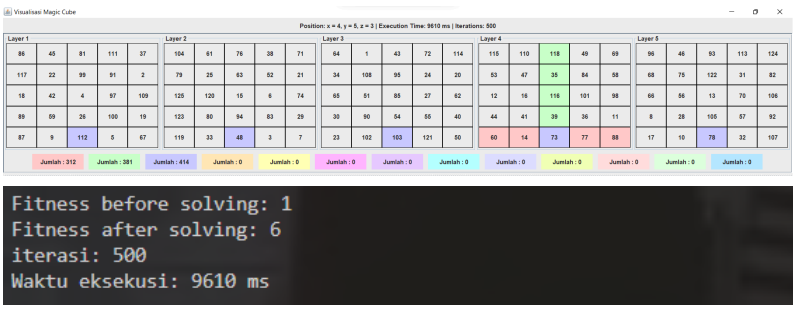
			  
		c	State sebelum solving:  State setelah solving:   
Iterasi = 500	Populasi = 100	a	State sebelum solving:  State setelah solving:

			  Fitness before solving: 0 Fitness after solving: 5 iterasi: 500 Waktu eksekusi: 1138 ms
		b	State sebelum solving:  State setelah solving:   Fitness before solving: 0 Fitness after solving: 5 iterasi: 500 Waktu eksekusi: 1258 ms
		c	State sebelum solving:  State setelah solving:

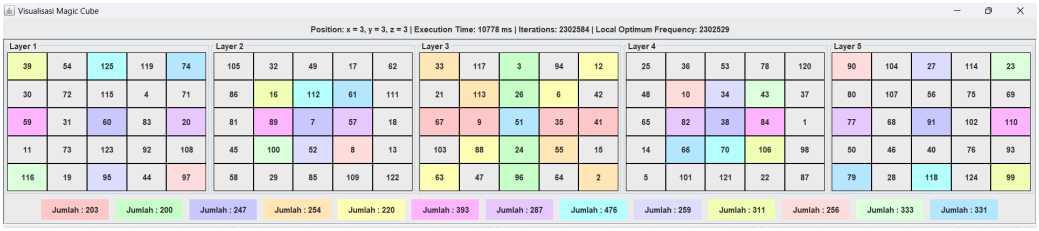
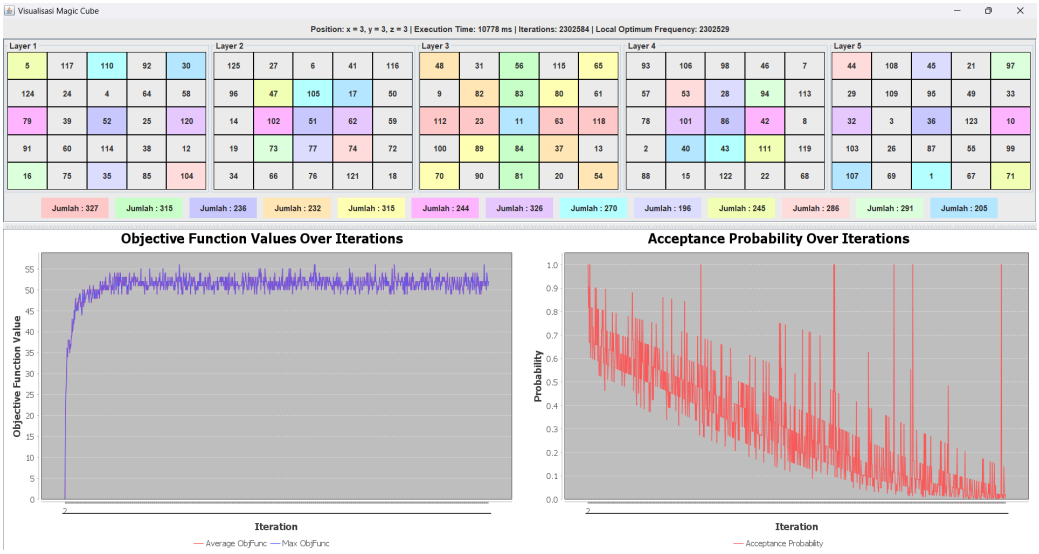
			 Fitness before solving: 0 Fitness after solving: 5 iterasi: 500 Waktu eksekusi: 1105 ms
	Populasi = 500	a	State sebelum solving:  State setelah solving:  Fitness before solving: 1 Fitness after solving: 7 iterasi: 500 Waktu eksekusi: 4842 ms
		b	State sebelum solving:  State setelah solving:

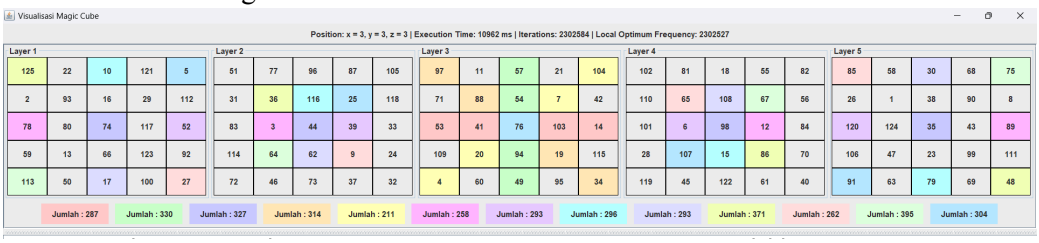
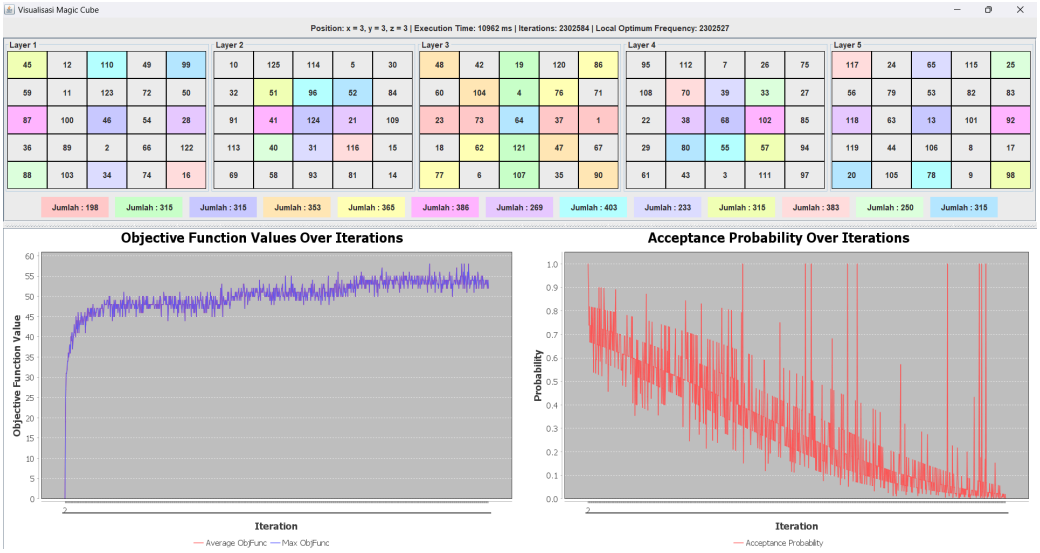
			  
		c	State sebelum solving:  State setelah solving:   
	Populasi = 1000	a	State sebelum solving:  State setelah solving:

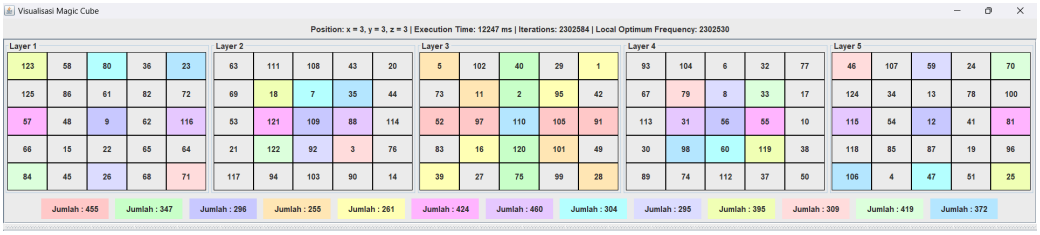
			 Fitness before solving: 0 Fitness after solving: 6 iterasi: 500 Waktu eksekusi: 10564 ms
		b	State sebelum solving:  State setelah solving:  Fitness before solving: 1 Fitness after solving: 6 iterasi: 500 Waktu eksekusi: 9593 ms
		c	State sebelum solving:  State setelah solving:

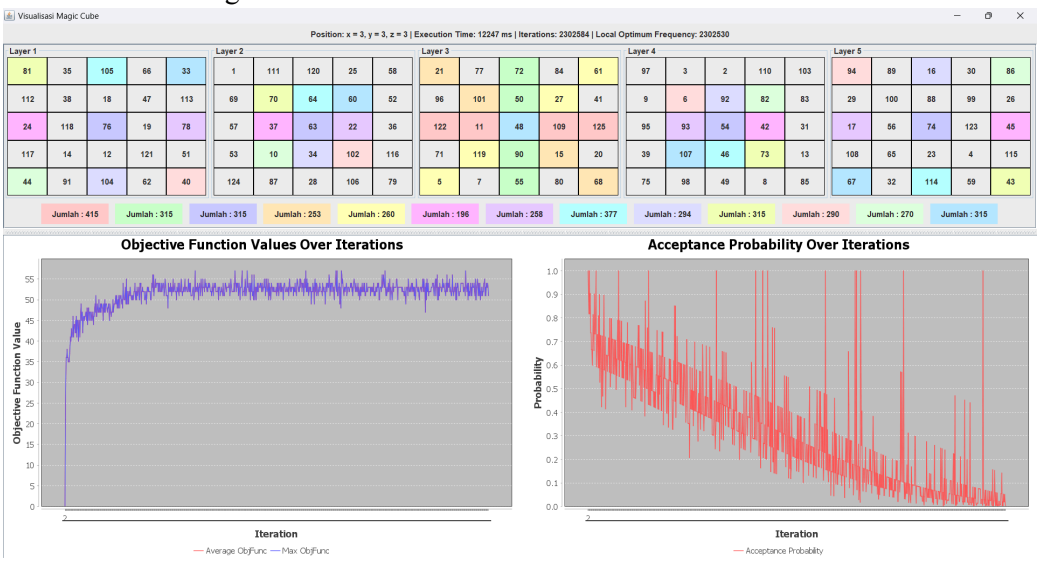
c. Algoritma Simulated Annealing

Percobaan	Hasil
1	State sebelum solving:  State setelah solving: 

	Number of iterations: 2302584 Local Optimum Stuck Frequency: 2302529 Fitness before solving: 0 Fitness after solving: 56 iterasi: 2302584 Waktu eksekusi: 10778 ms
2	State sebelum solving:  State setelah solving:  Number of iterations: 2302584 Local Optimum Stuck Frequency: 2302527 Fitness before solving: 0 Fitness after solving: 58 iterasi: 2302584 Waktu eksekusi: 10962 ms
3	State sebelum solving:



State setelah solving:




```
Number of iterations: 2302584
Local Optimum Stuck Frequency: 2302530
Fitness before solving: 0
Fitness after solving: 57
iterasi: 2302584
Waktu eksekusi: 12247 ms
```

Dari hasil eksperimen, genetic algorithm memiliki hasil solusi yang sangat jauh dari global optima. Hal ini disebabkan oleh pembangkitan populasinya yang dilakukan secara acak, dan ketika dilakukan random selection hingga crossover, hasilnya tidak dijamin mendekati global optima.

Kemudian, algoritma hill climbing with sideways move menunjukkan solusi yang cenderung lebih mendekati, namun tidak sampai setengah dari state value global optima. Hal ini disebabkan oleh algoritma Hill Climbing yang cukup tergantung pada titik awal dan banyaknya iterasi yang dilakukan. Ketika algoritma Hill Climbing dimulai dengan state awal yang cukup ‘beruntung’, maka akan mempermudah langkah selanjutnya dan meningkatkan kemungkinan untuk menemukan global optimum. Selain itu, permasalahan Magic Cube juga memiliki kemungkinan yang cukup besar untuk terjebak di

dalam local optimum, sehingga keberuntungan pada State awal akan sangat membantu untuk keluar dari plateau.

Setiap kali dijalankan, algoritma Hill Climbing with Sideways Move menghasilkan Magic Cube dengan nilai objective function sekitar 40 - 45. Dari beberapa kali percobaan, hasil Magic Cube cukup konsisten dan dieksekusi dalam durasi yang mirip. Durasi dari eksekusi algoritma Hill Climbing with Sideways Move ini rata-rata menghabiskan waktu sekitar 20 detik atau lebih hingga akhirnya berhenti karena sudah mencapai batas banyak iterasi sideways moves yang diatur yaitu 500.

Perbedaan parameter jumlah sideway moves maksimal berpengaruh pada runtime dan keakuratan program. Semakin banyak sideway moves yang diizinkan, maka program akan semakin akurat dan mendekati nilai maksimal hingga mencapai local optimal tanpa tetangga yang bernilai lesih. Akan tetapi, runtime program akan menjadi lebih lambat secara signifikan. Kami bereksperimen dan menentukan 500 sebagai jumlah sideway moves maksimal. Angka 500 konsisten menghasilkan hasil yang sama dengan angka sideway moves lainnya, sembai memiliki runtime yang relatif singkat dan tetap menjaga konsistensi.

Berbeda dengan algoritma lainnya, algoritma Simulated Annealing (SA) cenderung menghasilkan solusi yang paling mendekati global optimum, dengan state value sering mencapai lebih dari setengah nilai global optimum. Pendekatan ini mengandalkan peluang berbasis suhu untuk mengizinkan pergerakan ke solusi yang lebih buruk, terutama pada tahap awal eksplorasi, sehingga mencegah algoritma terjebak di lokal optimum. Seiring waktu, suhu menurun dan peluang menerima solusi buruk berkurang, mendorong eksploitasi solusi terbaik yang ditemukan. Dengan banyak iterasi, SA semakin mendekati solusi optimal karena eksplorasi berangsur-angsur dikurangi, menjaga keseimbangan eksplorasi dan eksploitasi. Ini membuat SA efektif dan fleksibel dalam menghadapi masalah optimasi kompleks. Dari banyak percobaan yang dilakukan diluar eksperimen di atas, SA konsisten akan hasil solusi state value nya di rentang 54 hingga 63 (sekitar 57,8% mendekati global optimum). Akan tetapi, algoritma ini juga tetap stuck di lokal optimum dan belum pernah bisa mencapai state value lebih dari 70. Ada beberapa kemungkinan penyebab akan terjadinya hal ini:

- Parameter Tidak Tepat

  Parameter suhu awal dan cooling rate mungkin tidak cukup untuk mengeksplorasi ruang solusi secara optimal. Suhu awal yang terlalu rendah atau cooling rate yang terlalu cepat dapat membatasi kemampuan algoritma untuk menerima solusi buruk yang dibutuhkan untuk keluar dari lokal optimum.

- Landscape Fungsi Objektif

  Jika fungsi objektif memiliki banyak puncak lokal yang sangat menonjol dan tidak cukup perbedaan dengan solusi global, SA mungkin kesulitan untuk keluar dari jebakan lokal optimum tersebut.

- Distribusi Solusi Buruk

Mungkin solusi yang lebih buruk di sekitar lokal optimum tidak cukup buruk sehingga peluang untuk keluar dari lokal optimum rendah. Ini menyebabkan algoritma menjadi cepat konvergen sebelum mencapai solusi global.

- Kriteria Penerimaan

  Batas probabilitas solusi mungkin perlu disesuaikan agar algoritma dapat menerima solusi buruk dalam jumlah yang memadai. Batas probabilitas yang terlalu ketat dapat menyebabkan algoritma SA sulit menjelajah area di sekitar lokal optimum.

Berdasarkan hasil eksperimen, tiap algoritma memiliki durasi yang berbeda-beda. Namun, durasi setiap program ini tidak bisa dibandingkan secara langsung karena adanya faktor atau variabel yang tidak dikontrol, seperti parameter tiap algoritma, sehingga durasi waktu (dalam hal ini sebagai variabel terikat) tidak bisa dikaitkan dengan algoritma (dalam hal ini sebagai variabel bebas) untuk dibandingkan dari satu algoritma ke algoritma lain.

Algoritma Hill Climbing dengan Sideway Moves dan Simulated Annealing menghasilkan nilai objective function yang cukup konsisten, berada dalam kisaran 40-60. Sebaliknya, Genetic Algorithm hanya mencapai nilai objective function sebesar 6, yang menunjukkan performa yang jauh lebih rendah. Hal ini menunjukkan bahwa Genetic Algorithm kurang cocok untuk skenario ini karena pola pencarian solusinya tidak dapat mengandalkan mekanisme crossover untuk menemukan solusi optimal, terutama dalam menghadapi pola khusus yang dibutuhkan dalam masalah ini.

Pada Genetic Algorithm, banyaknya iterasi maksimal dan jumlah populasi berpengaruh terhadap hasil pencarian, meskipun pada penerapannya, pengaruhnya tidak begitu signifikan. Jumlah populasi berkontribusi dalam memperkaya kemungkinan-kemungkinan dalam mencari jalan pada solution space. Banyaknya iterasi dapat memberi kesempatan lebih banyak untuk algoritma dalam melakukan eksplorasi dari gen-gen yang dihasilkan.

BAB III
KESIMPULAN DAN SARAN

1. Kesimpulan

Pada tugas besar ini, kami telah berhasil mengimplementasikan algoritma local search dalam menyelesaikan persoalan *magic cube*. Dengan pendekatan ini, kami hanya mampu mendapatkan solusi yang 57,8% mendekati global optimum yang memenuhi syarat-syarat dari suatu *magic cube*. Algoritma local search yang digunakan dapat meminimalkan jumlah langkah yang diperlukan, serta menunjukkan efisiensi dan efektivitas dalam mencapai hasil yang diinginkan.

Dari ketiga algoritma yang telah kami implementasikan, kami menemukan bahwa Algoritma Simulated Annealing merupakan algoritma yang terbaik untuk menyelesaikan permasalahan ini. Hal ini karena dari hasil pengujian yang telah dilakukan, Algoritma Simulated Annealing rata-rata berhasil menyusun Magic Cube dengan nilai objective function lebih tinggi daripada algoritma Genetic dan Hill Climbing with Side Move. Sehingga dapat disimpulkan bahwa algoritma Simulated Annealing dapat mendekati kondisi Magic Cube yang paling mendekati optimal.

2. Saran

Terdapat beberapa saran yang ingin kami sampaikan dalam pengerjaan tugas besar ini, yaitu:

1. Perlu dipertimbangkan kembali terkait pemilihan Objective Function dalam algoritma. Objective Function yang lebih relevan dan dan spesifik dengan karakteristik Magic Cube dapat meningkatkan kinerja algoritma untuk menghasilkan solusi yang lebih efisien dalam waktu dan iterasi yang lebih singkat.
2. Perlu dilakukan evaluasi menyeluruh terhadap pemilihan bahasa pemrograman Java dan struktur kode program Magic Cube Solver. Analisis mendalam terhadap kinerja, kemudahan pengembangan, dan skalabilitas bahasa lain akan membantu mengidentifikasi bahasa yang lebih sesuai untuk mencapai efisiensi optimal dalam pemecahan masalah Magic Cube. Selain itu, optimasi kode yang ada juga perlu dilakukan untuk meningkatkan keterbacaan dan mempermudah pemeliharaan program.

BAB IV
PEMBAGIAN TUGAS

NIM	Nama	Tugas
13522015	Yusuf Ardian Sandi	MagicCube Class, Algoritma Simulated Annealing, Main Class, Menu Class
13522018	Ibrahim Ihsan Rasyid	Kelas GeneticAlgorithm
13522031	Zaki Yudhistira Candra	Kelas HillClimbing
13522032	Tazkia Nizami	Visualisasi Magic Cube, Graph Data

BAB V
REFERENSI

- Link Github: https://github.com/TazakiN/Tubes1-AI
- Spesifikasi Tugas: Spesifikasi Tugas Besar 1 IF3070 Dasar Inteligensi Artifisial & IF3170 Inteligensi Artifisial 2024/2025
- Magic Cube : Features of the magic cube - Magisch vierkant
- Salindia Kuliah Intelegensi Buatan : Beyond Classical Search