

Laporan Tugas Besar 1
IF3170 Intelelegensi Artifisial
Pencarian Solusi Diagonal Magic Cube dengan Local Search



Disusun oleh:
Kelompok 67
13522066 - Nyoman Ganadipa Narayana
13522084 - Dhafin Fawwaz Ikramullah
13522107 - Rayendra Althaf Taraka Noor
13522109 - Azmi Mahmud Bazeid

Institut Teknologi Bandung

Daftar Isi

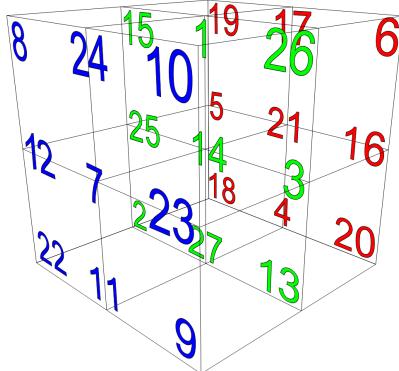
Daftar Isi.....	2
I. Deskripsi Persoalan.....	3
II. Pembahasan.....	4
A. Pemilihan Objective Function.....	4
B. Penjelasan Implementasi Algoritma Local Search.....	5
C. Hasil eksperimen dan analisis.....	6
III. Kesimpulan dan Saran.....	6
IV. Pembagian Tugas.....	7
V. Referensi.....	7

I. Deskripsi Persoalan

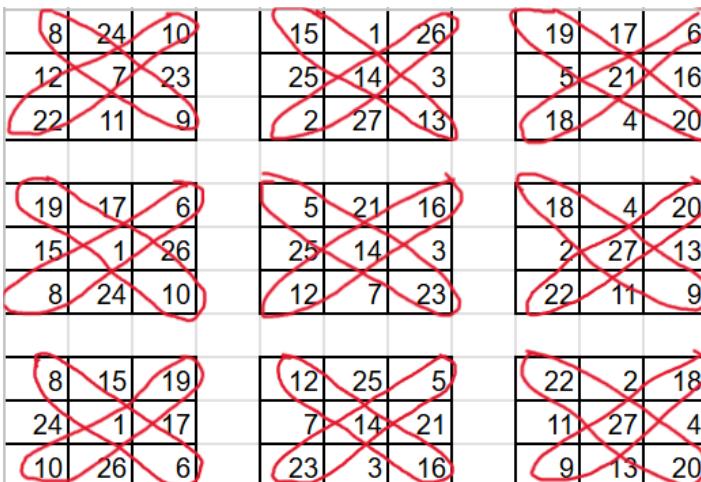
Diagonal magic cube merupakan kubus yang tersusun dari angka 1 hingga n^3 tanpa pengulangan dengan n adalah panjang sisi pada kubus tersebut. Angka-angka pada tersusun sedemikian rupa sehingga properti-properti berikut terpenuhi:

- 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

- Berikut ilustrasi dari potongan bidang yang ada pada suatu kubus berukuran 3:



- Terdapat 9 potongan bidang, yaitu:



- Diagonal yang dimaksud adalah yang dilingkari warna merah saja
- Ilustrasi dan penjelasan lebih detail bisa anda lihat di link berikut: [Features of the magic cube - Magisch vierkant](#)

Pada tugas ini, peserta kuliah akan menyelesaikan permasalahan Diagonal Magic Cube berukuran 5x5x5. Initial state dari suatu kubus adalah susunan angka 1 hingga 53 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 (tetapi hanya menukar posisi 2 angka saja juga diperbolehkan).

II. Pembahasan

A. Pemilihan Objective Function

Definisikan *sequence* sebagai suatu urutan bilangan pada *magic cube* yang mempunyai peran penting pada properti-properti *magic cube*: urutan angka yang terbentuk dari suatu baris, kolom, tiang, diagonal ruang, atau diagonal bidang. Terdapat tiga kandidat cara yang kami pikirkan untuk menjadi *objective function* persoalan ini:

1. Banyaknya *sequence* yang jumlahnya bukan magic number.
2. Seberapa jauh setiap *sequence* kepada magic number menggunakan standar deviasi.
3. Seberapa jauh setiap *sequence* kepada magic number menggunakan jumlah dari seluruh selisih.

Untuk kandidat pertama, kami menilai bahwa dengan menggunakan strategi tersebut sebagai objective function, maka terlalu banyak *shoulder* yang akan terjadi, karena tidak memperhatikan seberapa jauh atau seberapa besar *sequence* yang bukan magic number tersebut. Jadi kami berpikir untuk memperhatikan seberapa jauh *sequence* tersebut kepada magic number (kandidat 2), tetapi kami menilai bahwa menggunakan standar deviasi memerlukan langkah komputasi yang berat/banyak, sehingga dipilihlah **kandidat ketiga**, sehingga dapat memperhatikan seberapa jauh setiap *sequence*, tetapi juga dengan langkah komputasi yang ringan. Sehingga, heuristik (*h*) dan *objective function* (*f*) yang didefinisikan adalah sebagai berikut.

$$h(state) = \sqrt{\sum_{sequences} ((\sum seq - \text{magicnumber})^2)}$$

dan

$$f(state) = -h(state)$$

Perhatikan bahwa *objective function* yang dipilih selalu negatif dan nilai *f* yang lebih besar menunjukkan *state* yang lebih baik.

B. Penjelasan Implementasi Algoritma Local Search

1. Solver

```
export type Evaluator = (cube: CubeState) => number;
export type CubeStateChangeCallback = (cube: CubeState) => void;

export abstract class Solver {
    public static logConsoleEnabled = true;
    protected evaluator: Evaluator;
    protected onStateChange?: CubeStateChangeCallback;
    public constructor(onStateChange?: CubeStateChangeCallback,
evaluator: Evaluator = Solver.evaluateDeviationSqrt) {
        this.onStateChange = onStateChange;
        this.evaluator = evaluator;
    }

    static evaluateDeviationSqrt(cube: CubeState): number {
        let magicNumber = cube.calculateMagicNumber()
        let d = 0

        cube.iterateAndDo((arr: number[]) => {
            let total = arr.reduce((a, b) => a + b, 0)
            d += Math.abs(Math.sqrt(Math.abs(total - magicNumber)))
        })

        return -d
    }

    static evaluateMagicAmount(cube: CubeState): number {
        const magicNumber = cube.calculateMagicNumber()
        let magicCount = 0

        cube.iterateAndDo((arr: number[]) => {
            let total = arr.reduce((a, b) => a + b, 0)
            if(total === magicNumber) magicCount++
        })

        return magicCount
    }

    static onLog: (msg: string[]) => void = () => {};
}

protected log(startTime: number, current: CubeState, evaluator: Evaluator, iteration?: number, nMax?: number) {
    const msg = ["Time:", (performance.now() - startTime).toFixed(2) + " ms", "| Magic:", Solver.evaluateMagicAmount(current) + "/" + current.maxAmountOfMagic, "| Score:", evaluator(current).toFixed(2)];
}
```

```

        if(iteration !== undefined && nMax !== undefined)
            msg.push("| Iteration:", iteration + "/" + nMax);

        Solver.onLog(msg);

        if(!Solver.logConsoleEnabled) return;
        console.log(...msg);
    }

    abstract process(): CubeState;
}

```

Kelas	Fungsi/Metode	Penjelasan
Solver	konstruktor	Mengisi onStateChange
	evaluateDeviationSqrt()	Mengevaluasi suatu state kubus menggunakan minus dari modifikasi dari standar deviasi, yaitu digunakan standar deviasi dikali akar N.
	evaluateMagicAmount()	Mengevaluasi suatu state kubus menggunakan seberapa banyak <i>sequence</i> yang merupakan <i>magic number</i>
	log()	Prosedur untuk log suatu state kubus beserta informasi penting lainnya pada console.
	process()	Mendapatkan state dari suatu kubus setelah dilakukan prosedur tertentu. Akan diimplementasikan berdasarkan algoritma tertentu.

1.1. Steepest Ascent Hill-climbing

```

class SteepestAscentSolver extends Solver {
    public initialCube: CubeState;

    public constructor(cube: CubeState, onStateChange?: CubeStateChangeCallback, evaluator: Evaluator = Solver.evaluateDeviationSqrt) {
        super(onStateChange, evaluator);
        this.initialCube = cube;
    }
}

```

```

        // Init cached stuff
        this.initialCube.calculateMagicNumber();
        this.initialCube.maxAmountOfMagic;
        CubeState.getCubeSwapPairs(cube.content.length);
    }

    public process(): CubeState {
        let startTime = performance.now()
        const { evaluator, onStateChange } = this;

        let current = this.initialCube.getCopy()
        while (!current.isMagicCube()) {
            const neighbor =
                SteepestAscentSolver.getHighestSuccessorByPairs(current, evaluator)

            if (evaluator(neighbor) <= evaluator(current)) {
                this.log(startTime, current, evaluator, 0, 0)
                return current
            }
            current = neighbor.getCopy();
            onStateChange?.(neighbor)

            this.log(startTime, current, evaluator)
        }

        return current
    }

    public static getHighestSuccessorByPairs(cube: CubeState,
evaluator: Evaluator): CubeState {
        const pairs =
            CubeState.getCubeSwapPairs(cube.content.length);
        let highestValue = evaluator(cube);
        let highestPair = pairs[0];
        for(let pair of pairs) {
            cube.swap(pair[0][0], pair[0][1], pair[0][2],
pair[1][0], pair[1][1], pair[1][2]);
            let value = evaluator(cube);
            if(value > highestValue) {
                highestValue = value;
                highestPair = pair;
            }
        }

        // Undo swap
        cube.swap(pair[0][0], pair[0][1], pair[0][2],
pair[1][0], pair[1][1], pair[1][2]);
    }
    let highestCube = cube.getCopy();
    highestCube.swap(highestPair[0][0], highestPair[0][1],
highestPair[0][2], highestPair[1][0], highestPair[1][1],

```

```

        highestPair[1][2]);
        return highestCube;
    }
}

```

Kelas	Fungsi/Metode	Penjelasan
SteepestAscentSolver	konstruktor	Mengisi cube yang akan disolve, strategi evaluation,
	process()	Memberikan hasil akhir state suatu kubus apabila diproses dengan algoritma steepest-ascent hill climbing.
	getHighestSuccessorByPairs()	Mendapatkan suksesor terbaik apabila suksesor yang di span adalah hasil dari pertukaran 2 titik pada kubus.

1.2 Steepest Ascent with Sideways Move

```

class SidewaysMoveSolver extends Solver {
    public initialCube: CubeState;
    maxSideways: number;

    public constructor(
        cube: CubeState,
        maxSideways: number,
        onStateChange?: CubeStateChangeCallback,
        evaluator: Evaluator = Solver.evaluateDeviationSqrt
    ) {
        super(onStateChange, evaluator);
        this.initialCube = cube;

        // Init cached stuff
        this.maxSideways = maxSideways;
        this.initialCube.calculateMagicNumber();
        this.initialCube.maxAmountOfMagic;
        CubeState.getCubeSwapPairs(cube.content.length);
    }

    public process(): CubeState {
        let startTime = performance.now();
        const { evaluator, onStateChange } = this;
        let sidewaysCount = 0;
    }
}

```

```

let current = this.initialCube.getCopy();
    while (!current.isMagicCube()) && sidewaysCount <
this.maxSideways) {
                                const neighbor =
SidewaysMoveSolver.getHighestSuccessorByPairs(
    current,
    evaluator
);

const neighborVal = evaluator(neighbor);
const currentVal = evaluator(current);
if (neighborVal < currentVal) {
    this.log(startTime, current, evaluator, 0, 0);
    return current;
}
if (neighborVal === currentVal) {
    sidewaysCount++;
}

current = neighbor.getCopy();
onStateChange?.(neighbor);
this.log(startTime, current, evaluator);
}

return current;
}

public static getHighestSuccessorByPairs(
    cube: CubeState,
    evaluator: Evaluator
): CubeState {
    const pairs = CubeState.getCubeSwapPairs(cube.content.length);
    let highestValue = evaluator(cube);
    let highestPair = pairs[0];
    for (let pair of pairs) {
        cube.swap(
            pair[0][0],
            pair[0][1],
            pair[0][2],
            pair[1][0],
            pair[1][1],
            pair[1][2]
        );
        let value = evaluator(cube);
        if (value > highestValue) {
            highestValue = value;
            highestPair = pair;
        }
    }

    // Undo swap
}

```

```

        cube.swap(
            pair[0][0],
            pair[0][1],
            pair[0][2],
            pair[1][0],
            pair[1][1],
            pair[1][2]
        );
    }
    let highestCube = cube.getCopy();
    highestCube.swap(
        highestPair[0][0],
        highestPair[0][1],
        highestPair[0][2],
        highestPair[1][0],
        highestPair[1][1],
        highestPair[1][2]
    );
    return highestCube;
}
}

```

Kelas	Fungsi/Metode	Penjelasan
SidewaysMoveSolver	konstruktor	Mengisi cube yang akan disolve, strategi evaluation, dan memasang jumlah maximum sideways yang mungkin.
	process()	Memberikan hasil akhir state suatu kubus apabila diproses dengan algoritma steepest-ascent hill climbing.
	getHighestSuccessorByPairs()	Mendapatkan suksesor terbaik apabila suksesor yang <i>di-span</i> adalah hasil dari pertukaran 2 titik pada kubus.

1.3 Random Restart Hill Climbing

```

class RandomRestartHillClimbingSolver extends Solver {
    public initialCube: CubeState;
    private maxRestarts: number;

    /**
     *

```

```

    * Constructs a RandomRestartHillClimbingSolver.
    * @param cube The initial CubeState.
        * @param onStateChange Optional callback invoked on state
changes.
            * @param evaluator The heuristic function to evaluate
CubeStates.
                * @param maxRestarts Maximum number of random restarts (default:
1000).
                    * @param maxIterationsPerRestart Maximum iterations per restart
(default: 1000).
                */
    public constructor(
        cube: CubeState,
        maxRestarts: number = 1000,
        onStateChange?: CubeStateChangeCallback,
        evaluator: Evaluator = Solver.evaluateDeviationSqrt,
    ) {
        super(onStateChange, evaluator);
        this.initialCube = cube;
        this.maxRestarts = maxRestarts;

        // Initialize cached properties
        this.initialCube.calculateMagicNumber();
        this.initialCube.maxAmountOfMagic;
        CubeState.getCubeSwapPairs(cube.content.length);
    }

    /**
     * Executes the Random Restart Hill Climbing algorithm.
     * @returns The best CubeState found.
     */
    public process(): CubeState {
        let startTime = performance.now();
        const { evaluator, onStateChange } = this;

        let best = this.initialCube.getCopy();
        let bestScore = evaluator(best);

        for (let restart = 0; restart < this.maxRestarts; restart++) {
            // Create a new random cube
            let randomCube = CubeState.createRandomCube(
                this.initialCube.content.length
            );

            // Perform Steepest Ascent from the random cube
            const solver = new SteepestAscentSolver(
                randomCube,
                onStateChange,
                evaluator
            );
            let current = solver.process();

```

```

        let currentScore = evaluator(current);

        // Update the best solution found so far
        if (currentScore > bestScore) {
            best = current;
            bestScore = currentScore;
            console.log(`Restart ${restart + 1}: New best score
${bestScore}`);
        }

        // Check if a magic cube is found
        if (best.isMagicCube()) {
            console.log(`Magic cube found at restart ${restart + 1}`);
            break;
        }

        console.log(`Restart ${restart + 1}: Best score so far
${bestScore}`);
    }

    this.log(startTime, best, evaluator, this.maxRestarts,
this.maxRestarts);
    return best;
}
}

```

Kelas	Fungsi/Metode	Penjelasan
Random Restart Hill Climbing	konstruktor	Mengisi cube yang akan diselesaikan, onStateChange, strategi objective function, dan jumlah maksimal restart.
	process()	Memberikan state kubus setelah melakukan algoritma random restart hill climbing dengan jumlah maksimal restart yang ditentukan.

1.4 Stochastic Hill Climbing

```

export class StochasticSolver extends Solver {

    public initialCube: CubeState;
    stochasticNMax = 100000;
}

```

```

        public constructor(cube: CubeState, stochasticNMax: number,
onStateChange?: CubeStateChangeCallback, evaluator: Evaluator =
Solver.evaluateDeviationSqrt) {
    super(onStateChange, evaluator);
    this.initialCube = cube;

    // Init cached stuff
    this.stochasticNMax = stochasticNMax;
    this.initialCube.calculateMagicNumber();
    this.initialCube.maxAmountOfMagic;
    CubeState.getCubeSwapPairs(cube.content.length);
}

public process(): CubeState {
    let startTime = performance.now()
    const { evaluator, onStateChange } = this;

    let current = this.initialCube.getCopy()
    let nMax = this.stochasticNMax
    let iteration = 0
    while (!current.isMagicCube() && iteration < nMax) {
        iteration++;
        const neighbor = current.getRandomSuccessor()

        if (evaluator(neighbor) > evaluator(current)) {
            current = neighbor
            onStateChange?.(neighbor)
            this.log(startTime, current, evaluator, iteration,
nMax)
        }
    }

    this.log(startTime, current, evaluator, iteration, nMax)
    return current
}
}

```

Kelas	Fungsi/Metode	Penjelasan
Stochastic Hill Climbing	konstruktor	Mengisi cube yang akan diselesaikan, onStateChange, strategi objective function, dan jumlah iterasi.

	process()	Memberikan state kubus setelah melakukan algoritma random restart hill climbing dengan jumlah maksimal restart yang ditentukan.
--	-----------	---

1.5. Simulated Annealing

```

export class SimulatedAnnealingSolver extends Solver {
  public initialCube: CubeState;
  private initialTemperature: number;
  private finalTemperature: number;
  private temperature?: number;
  private coolingRate: number;
  private current?: CubeState;
  private currentScore?: number;

  /**
   * Constructs a SimulatedAnnealingSolver.
   * @param cube The initial CubeState.
   * @param onStateChange Optional callback invoked on state changes.
   * @param evaluator The heuristic function to evaluate CubeStates.
   * @param initialTemperature Starting temperature (default: 1e5).
   * @param finalTemperature Temperature at which to stop the algorithm (default: 1).
   * @param coolingRate Rate at which the temperature decreases (default: 0.99).
   */
  public constructor(
    cube: CubeState,
    onStateChange?: CubeStateChangeCallback,
    evaluator: Evaluator = Solver.evaluateDeviationSqrt
  ) {
    super(onStateChange, evaluator);
    this.initialCube = cube;
    this.initialTemperature = 2000;
    this.finalTemperature = 0;
    this.coolingRate = 0.999;

    // Initialize cached properties
    this.initialCube.calculateMagicNumber();
    this.initialCube.maxAmountOfMagic;
    CubeState.getCubeSwapPairs(cube.content.length);
  }

  /**
   * Executes the Simulated Annealing algorithm.
   * @returns The final CubeState after solving.
   */
}

```

```
public process(): CubeState {
    let startTime = performance.now();
    const { evaluator } = this;

    this.current = this.initialCube.getCopy();
    this.currentScore = evaluator(this.current);

    this.temperature = this.initialTemperature;

    let iteration = 0;

    while (
        this.temperature > this.finalTemperature &&
        !this.current.isMagicCube()
    ) {
        iteration++;

        // Generate a random neighbor
        let neighbor = this.current.getRandomSuccessor();
        let neighborScore = evaluator(neighbor);
        let delta = neighborScore - this.currentScore;

        if (delta > 0) {
            this.accept(neighbor, neighborScore);
        } else {
            // Accept the worse neighbor with a probability
            let probability = Math.exp(delta / this.temperature);

            if (Math.random() < probability) {
                this.accept(neighbor, neighborScore);
            }
        }

        this.cooldown();
        this.log(startTime, this.current, evaluator, iteration);
    }

    this.log(startTime, this.current, evaluator, iteration);

    return this.current;
}

private accept(neighbor: CubeState, neighborScore: number) {
    const { onStateChange } = this;

    this.current = neighbor.getCopy();
    this.currentScore = neighborScore;
    onStateChange?.(this.current);
}

private cooldown() {
```

```

        if (!this.temperature) return;
        this.temperature = this.temperature * this.coolingRate;
    }
}

```

Kelas	Fungsi/Metode	Penjelasan
Simulated Annealing	konstruktor	Mengisi cube yang akan diselesaikan, onStateChange, strategi objective function, dan onStateChange.
	process()	Memberikan state kubus setelah melakukan algoritma simulated annealing hingga temperatur di bawah yang ditentukan.
	accept()	Menerima suksesor tersebut menjadi state yang baru.
	cooldown()	Mendinginkan temperatur pada algoritma simulated annealing

1.6 Genetic Algorithm

```

export class GeneticSolver extends Solver {
    degree: number;
    populationSize: number;
    maxIteration: number;

    public constructor(degree: number, maxIteration: number,
populationSize: number, onStateChange?: CubeStateChangeCallback,
evaluator: Evaluator = Solver.evaluateDeviationSqrt) {
        super(onStateChange, evaluator);
        this.degree = degree;
        this.maxIteration = maxIteration;
        this.populationSize = populationSize;

        // Init cached stuff
    }

    public process(): CubeState {
        // Control variables
        const populationSize: number = this.populationSize;
    }
}

```

```

        const maxIteration: number = this.maxIteration;
        const mutationRate: number = 0.05;

        const degree: number = this.degree;
        let population: CubeState[] = Array.from({ length: populationSize }, () => CubeState.createRandomCube(degree));

        let iteration: number = 0;
        while (true) {
            const [parent1, parent2] =
GeneticSolver.chooseRandomPair(population, this.evaluator);
            const [parent3, parent4] =
GeneticSolver.chooseRandomPair(population, this.evaluator);

            let [child1, child2] = GeneticSolver.reproduce(parent1,
parent2);
            let [child3, child4] = GeneticSolver.reproduce(parent3,
parent4);

            const uniqueParents = new Set([parent1, parent2, parent3,
parent4]);
            population = population.filter(individual =>
!uniqueParents.has(individual));

            if (Math.random() < mutationRate)
GeneticSolver.mutate(child1);
            if (Math.random() < mutationRate)
GeneticSolver.mutate(child2);
            if (Math.random() < mutationRate)
GeneticSolver.mutate(child3);
            if (Math.random() < mutationRate)
GeneticSolver.mutate(child4);

            population.push(child1, child2, child3, child4);

            const bestCube: CubeState = population.reduce((best,
current) =>
    this.evaluator(current) > this.evaluator(best) ?
current : best
);

            if (bestCube.isMagicCube()) {
                return bestCube;
            }

            iteration += 1;
            if (iteration >= maxIteration) {
                console.log("Total:",
Solver.evaluateMagicAmount(bestCube));
                return bestCube;
            }
        }
    }
}

```

```

    }
}

public static chooseRandomPair(population: CubeState[], evaluator: Evaluator = Solver.evaluateDeviationSqrt): [CubeState, CubeState] {
    const totalScore = population.reduce((sum, cube) => sum + evaluator(cube), 0);

    const selectRandom = (): CubeState => {
        const randomValue = Math.random() * totalScore;
        let cumulativeScore = 0;
        for (const cube of population) {
            cumulativeScore += evaluator(cube);

            if (cumulativeScore <= randomValue) {
                return cube;
            }
        }
        return population[population.length - 1];
    };

    const first = selectRandom();
    const second = selectRandom();

    return [first, second];
}

public static reproduce(parent1: CubeState, parent2: CubeState): [CubeState, CubeState] {
    const n = parent1.content.length;
    let child1 = parent1.getCopy();
    let child2 = parent2.getCopy();

    const fixedNumbers1 = GeneticSolver.getFixedNumbers(parent1);
    const notFixedList1 = GeneticSolver.getNonFixedNumbers(parent1, parent2);

    const fixedNumbers2 = GeneticSolver.getFixedNumbers(parent2);
    const notFixedList2 = GeneticSolver.getNonFixedNumbers(parent2, parent1);

    let count1 = 0;
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < n; j++) {
            for (let k = 0; k < n; k++) {
                if
(fixedNumbers1.has(child1.content[i][j][k])) continue;
                child1.content[i][j][k] = notFixedList1[count1++];
            }
        }
    }
}

```

```

        }
    }

let count2 = 0;
for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++) {
        for (let k = 0; k < n; k++) {
            if
(fixedNumbers2.has(child2.content[i][j][k])) continue;
            child2.content[i][j][k] = notFixedList2[count2++];
        }
    }
}

return [child1, child2];
}

public static mutate(cube: CubeState) {
cube.swapRandom();
}

public static getFixedNumbers(cube: CubeState): Set<number> {
const n = cube.content.length;
const possibleLocations = GeneticSolver.getSpaceDiagonalLocations(n)
.concat(GeneticSolver.getFaceDiagonalLocations(n))
.concat(GeneticSolver.getParallelSideLocations(n));

const fixedNumbers = new Set<number>();
for (const location of possibleLocations) {
    const locationSum = location.reduce((sum, [i, j, k]) =>
sum + cube.content[i][j][k], 0);
    if (locationSum === CubeState.calculateMagicNumber(n)) {
        for (const [i, j, k] of location) {
            fixedNumbers.add(cube.content[i][j][k]);
        }
    }
}

return fixedNumbers;
}
public static getNonFixedNumbers(cube: CubeState, otherCube: CubeState): number[] {
const fixedNumbers = GeneticSolver.getFixedNumbers(cube);
const nonFixedList: number[] = [];

for (const row of otherCube.content) {
    for (const subRow of row) {
        for (const value of subRow) {

```

```

        if (!fixedNumbers.has(value)) {
            nonFixedList.push(value);
        }
    }
}

return nonFixedList;
}
public static getSpaceDiagonalLocations(n: number): Array<Array<[number, number, number]>> {
    const diagonals: Array<Array<[number, number, number]>> = [];

    diagonals.push(Array.from({ length: n }, (_, i) => [i, i, i] as [number, number, number]));
    diagonals.push(Array.from({ length: n }, (_, i) => [i, i, n - 1 - i] as [number, number, number]));
    diagonals.push(Array.from({ length: n }, (_, i) => [i, n - 1 - i, i] as [number, number, number]));
    diagonals.push(Array.from({ length: n }, (_, i) => [i, n - 1 - i, n - 1 - i] as [number, number, number]));

    return diagonals;
}

public static getFaceDiagonalLocations(n: number): Array<Array<[number, number, number]>> {
    const diagonals: Array<Array<[number, number, number]>> = [];

    for (let x = 0; x < n; x++) {
        diagonals.push(Array.from({ length: n }, (_, i) => [x, i, i] as [number, number, number]));
        diagonals.push(Array.from({ length: n }, (_, i) => [x, i, n - 1 - i] as [number, number, number]));
    }

    for (let y = 0; y < n; y++) {
        diagonals.push(Array.from({ length: n }, (_, i) => [i, y, i] as [number, number, number]));
        diagonals.push(Array.from({ length: n }, (_, i) => [i, y, n - 1 - i] as [number, number, number]));
    }

    for (let z = 0; z < n; z++) {
        diagonals.push(Array.from({ length: n }, (_, i) => [i, i, z] as [number, number, number]));
        diagonals.push(Array.from({ length: n }, (_, i) => [i, n - 1 - i, z] as [number, number, number]));
    }

    return diagonals;
}

```

```

    }

    public static getParallelSideLocations(n: number):
Array<Array<[number, number, number]>> {
    const sides: Array<Array<[number, number, number]>> = [];

    for (let y = 0; y < n; y++) {
        for (let z = 0; z < n; z++) {
            sides.push(Array.from({ length: n }, (_, i) => [i,
y, z] as [number, number, number]));
        }
    }

    for (let x = 0; x < n; x++) {
        for (let z = 0; z < n; z++) {
            sides.push(Array.from({ length: n }, (_, i) => [x,
i, z] as [number, number, number]));
        }
    }

    for (let x = 0; x < n; x++) {
        for (let y = 0; y < n; y++) {
            sides.push(Array.from({ length: n }, (_, i) => [x,
y, i] as [number, number, number]));
        }
    }

    return sides;
}
}

```

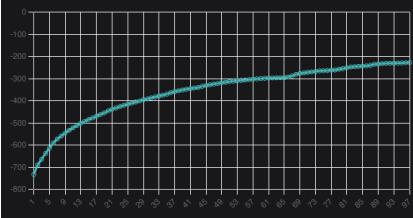
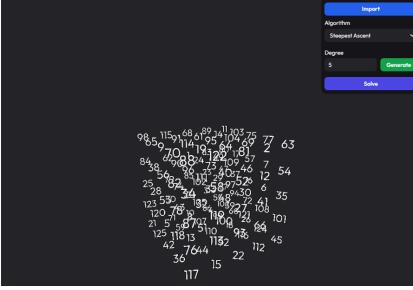
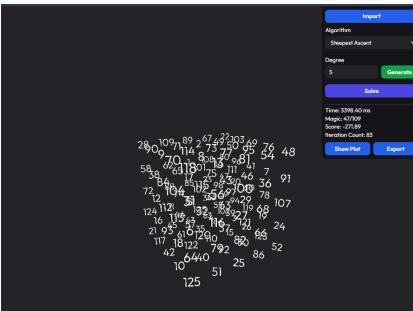
Kelas	Fungsi/Metode	Penjelasan
GeneticSolver	konstruktor	Mengisi variabel kontrol (iterasi maksimal dan ukuran populas) dan strategi evaluation.
	process()	Memberikan state kubus setelah melakukan algoritma genetic algorithm hingga temperatur di bawah yang ditentukan.
	chooseRandomPair()	Memilih 2 elemen dari populasi dengan probabilitas yang proportional dengan hasil evaluasi
	reproduce()	Menerima 2 kubus dan melakukan reproduksi atau

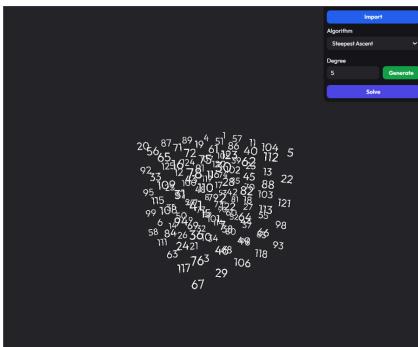
		crossover sehingga menghasilkan 2 kubus yang baru.
	mutate()	Melakukan mutasi pada sebuah kubus.
	getFixedNumbers()	Menerima sebuah kubus dan menentukan sel-sel mana yang sudah dalam posisi di mana elemen tersebut pada suatu sequence yang terpenuhi
	getNonFixedNumbers()	Menerima dua buah kubus. Mengeluarkan komplemen dari hasil dari getFixedNumbers pada kubus pertama. Hasil yang dikeluarkan terurut relatif pada kubus kedua.

C. Hasil eksperimen dan analisis

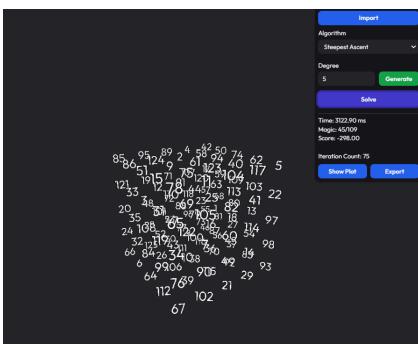
1. Eksperimen Steepest Hill Climbing

Eksperimen ke-	Visualisasi	Penjelasan Eksperimen
1	<p>Initial state:</p> <p>Final state:</p>	<p>Time: 4105.10 ms Magic: 49/109 Score: -227.80 Iteration Count: 98</p>

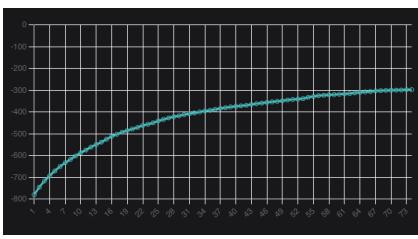
	<p>Plot Objective function terhadap iterasi:</p> 	
2	<p>Initial state:</p>  <p>Final state:</p> 	<p>Time: 3398.40 ms Magic: 47/109 Score: -271.89 Iteration Count: 83</p>
3	<p>Initial state:</p>	<p>Time: 3122.90 ms Magic: 45/109 Score: -298.00 Iteration Count: 75</p>



Final State:

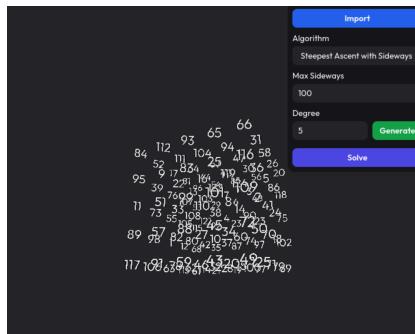


Plot Objective function terhadap iterasi:

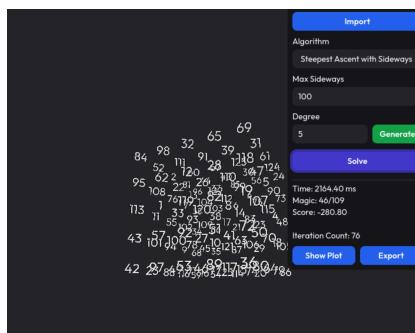


2. Eksperimen Steepest Ascent with Sideways Move

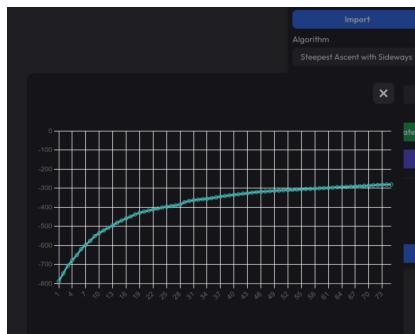
Eksperimen ke-	Visualisasi	Penjelasan Eksperimen
1	Initial state: [Blank]	Time: 2164.40 ms Magic: 46/109 Score: -280.80 Iteration Count: 76



Final State:

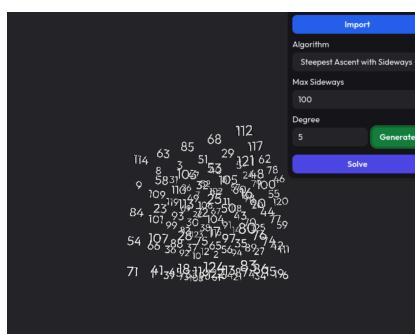


Plot *objective function* terhadap iterasi:



2

Initial state:



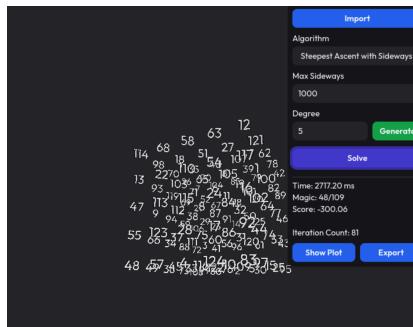
Time: 2717.20 ms

Magic: 48/109

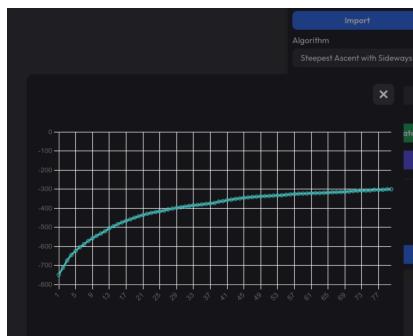
Score: -300.06

Iteration Count: 81

Final state:

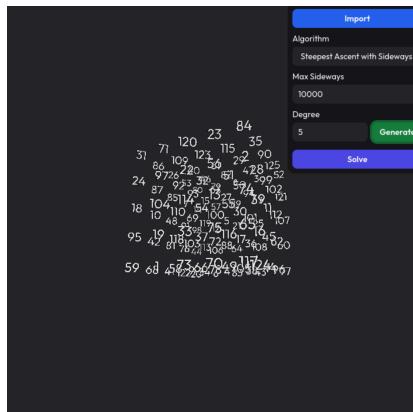


Plot *objective function* terhadap iterasi:



3

Initial state:



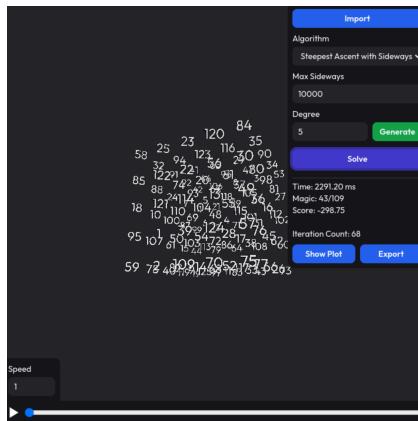
Final state:

Time: 2291.20 ms

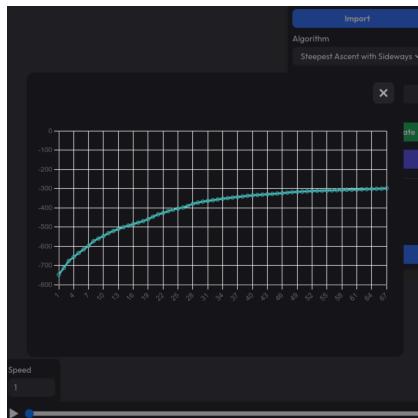
Magic: 43/109

Score: -298.75

Iteration Count: 68

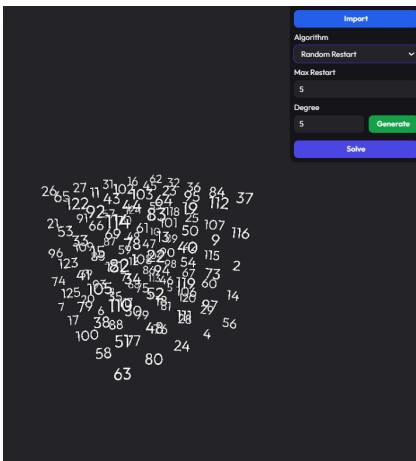


Plot *objective function* terhadap iterasi:

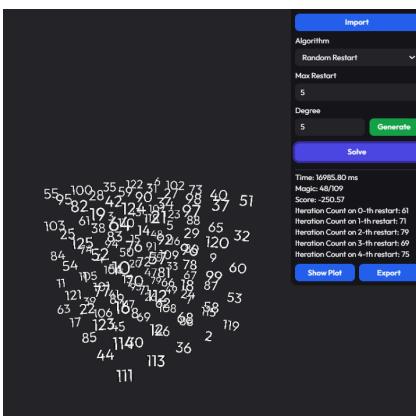


3. Eksperimen Random Restart Hill Climbing

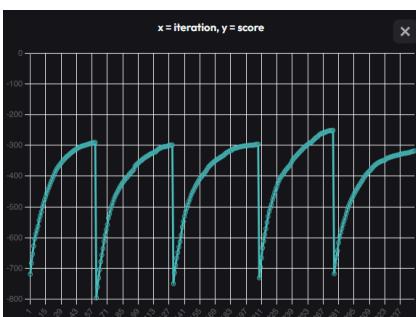
Eksperimen ke-	Visualisasi	Penjelasan Eksperimen
1	Initial state: 	Max restart: 5 Time: 16985.80 ms Magic: 48/109 Score: -250.57 Iteration Count on 0-th restart: 61 Iteration Count on 1-th restart: 71 Iteration Count on 2-th restart: 79 Iteration Count on 3-th restart: 69 Iteration Count on 4-th restart: 75



Final State:



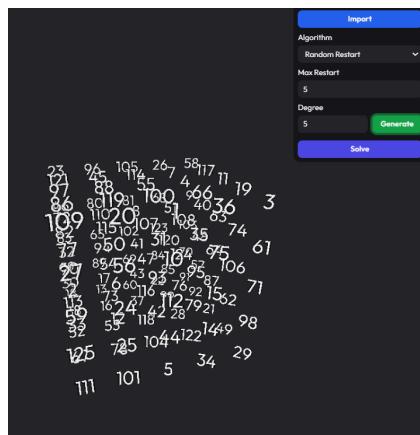
Plot Objective function terhadap iterasi:



2

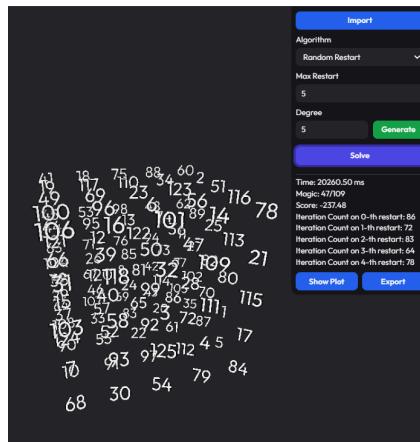
Initial state:

Max restart: 5
Time: 20260.50 ms
Magic: 47/109
Score: -237.48
Iteration Count on 0-th restart: 86
Iteration Count on 1-th restart: 72

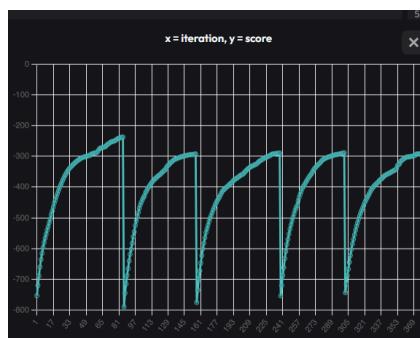


Iteration Count on 2-th restart: 83
Iteration Count on 3-th restart: 64
Iteration Count on 4-th restart: 78

Final state:



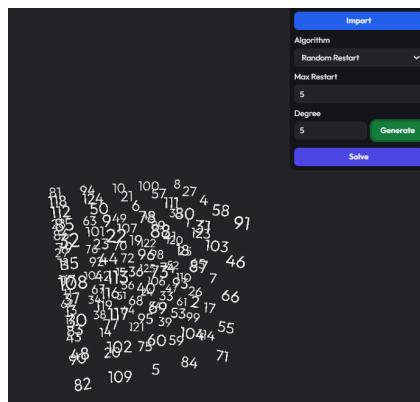
Plot Objective function terhadap iterasi:



3

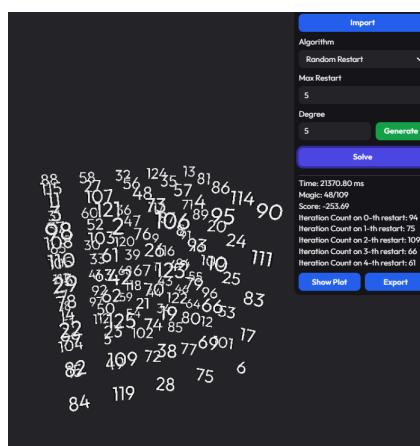
Initial state:

Max restart: 5
Time: 21370.80 ms
Magic: 48/109
Score: -253.69
Iteration Count on 0-th restart: 94



Iteration Count on 1-th restart: 75
 Iteration Count on 2-th restart: 109
 Iteration Count on 3-th restart: 66
 Iteration Count on 4-th restart: 61

Final state:

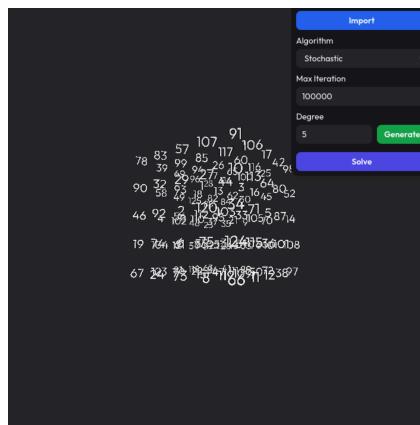


Plot Objective function terhadap iterasi:



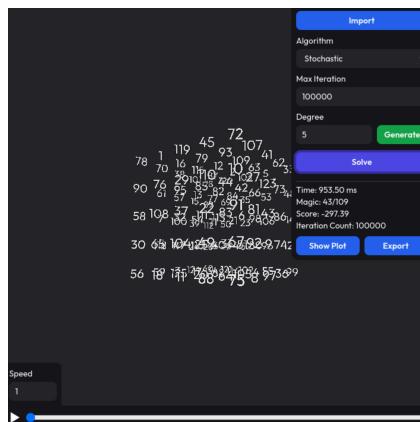
4. Eksperimen Stochastic Hill Climbing

Eksperimen ke-	Visualisasi	Penjelasan Eksperimen
1	Initial state: Time: 953.50 ms Magic: 43/109 Score: -297.39	



Iteration Count: 100000

Final state:



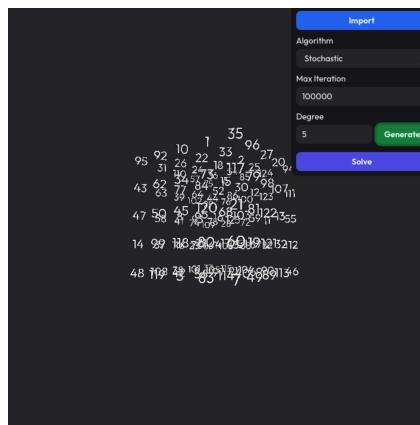
Plot *objective function* terhadap iterasi



2

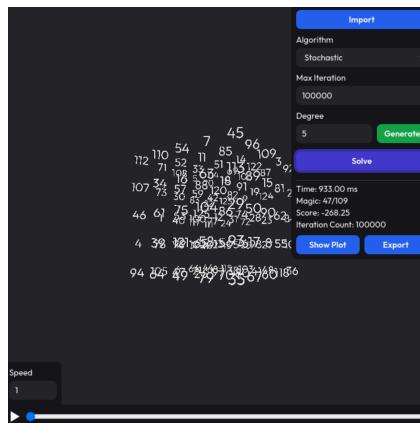
Initial state:

Time: 933.00 ms
Magic: 47/109

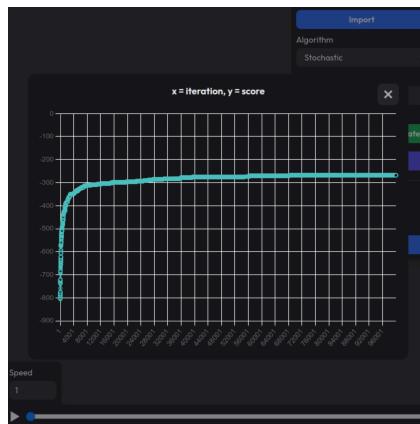


Score: -268.25
Iteration Count: 100000

Final state:



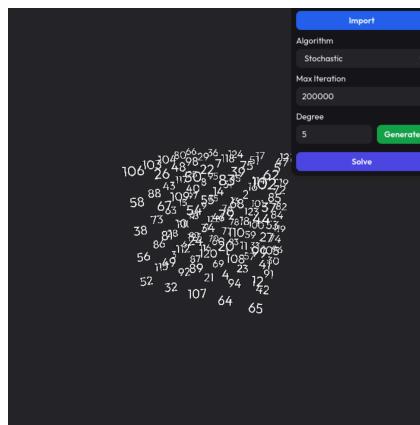
Plot *objective function* terhadap iterasi:



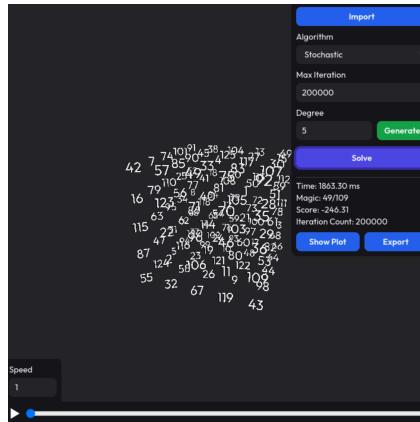
3

Initial state:

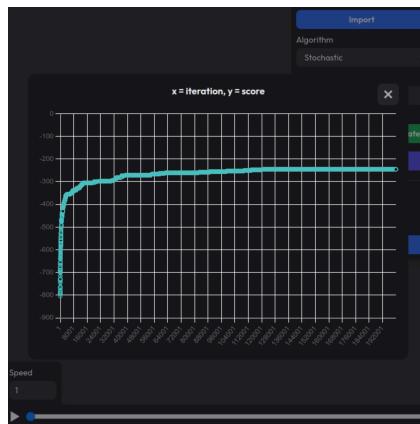
Time: 1863.30 ms
Magic: 49/109
Score: -246.31
Iteration Count: 200000



Final state:



Plot *objective function* terhadap iterasi:

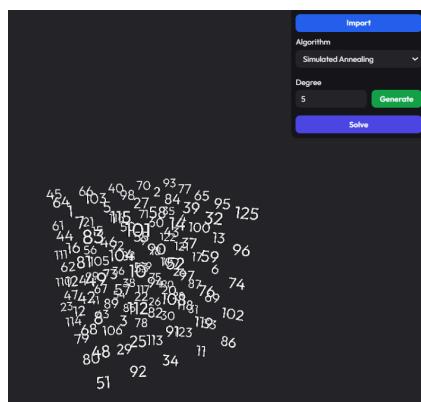


5. Eksperimen Simulated Annealing

Eksperimen ke-	Visualisasi	Penjelasan Eksperimen
----------------	-------------	-----------------------

1

Initial state:



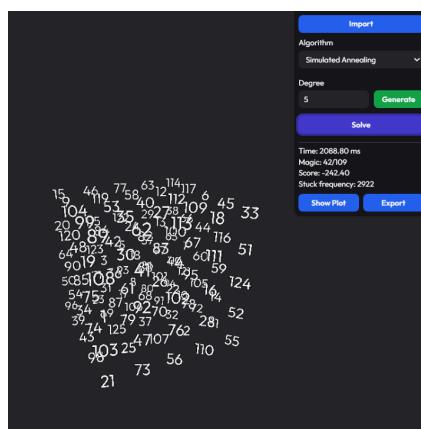
Time: 2088.80 ms

Magic: 42/109

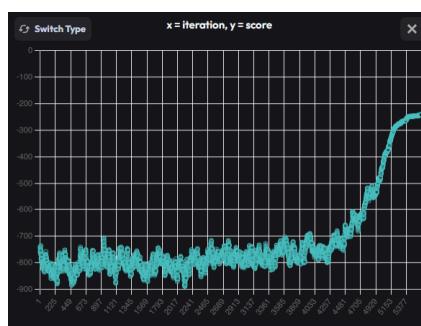
Score: -242.40

Stuck frequency: 2922

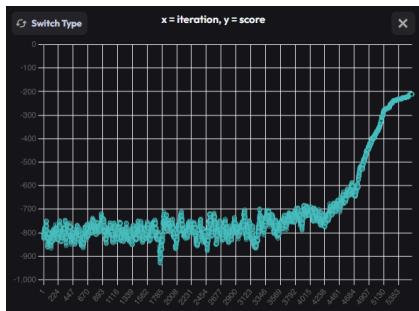
Final state:



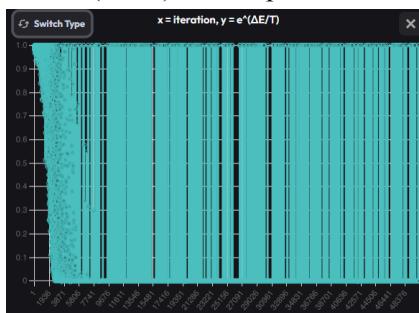
Plot Objective function terhadap iterasi:

Plot $e^{(\Delta E/T)}$ terhadap iterasi:

2	<p>Initial state:</p> <pre> 39 68 102 127 14 07 6 121 112 41 64 105 31 7 24 10 5 35 48 32 6 59 98 87 75 35 54 82 118 9 78 13 76 66 120 19 44 2 61 80 57 108 109 65 72 62 84 12 44 10 23 53 63 84 12 126 37 2 37 77 88 36 16 35 5 104 88 45 89 10 106 22 85 109 50 51 5 60 0 103 93 42 27 69 103 113 55 </pre>	Time: 2320.70 ms Magic: 49/109 Score: -211.20 Stuck frequency: 2908
	<p>Final state:</p> <pre> 58 69 41 75 123 56 12 48 59 73 107 37 35 76 48 102 2 9 125 89 118 2 79 101 100 157 83 89 19 180 125 69 99 83 56 101 79 251 45 110 70 86 101 113 251 128 110 70 86 101 113 982 265 4 320 40 113 261 873 103 4 320 40 113 472 123 83 15 77 59 109 87 96 18 31 124 92 38 84 49 104 31 124 92 38 84 28 104 31 124 92 38 84 102 63 52 78 46 </pre>	Plot Objective function terhadap iterasi:

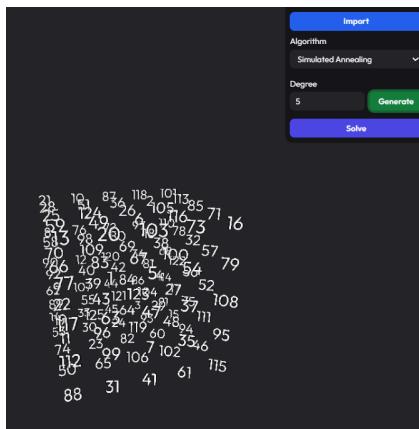


Plot $e^{-(\Delta E/T)}$ terhadap iterasi:



3

Initial state:



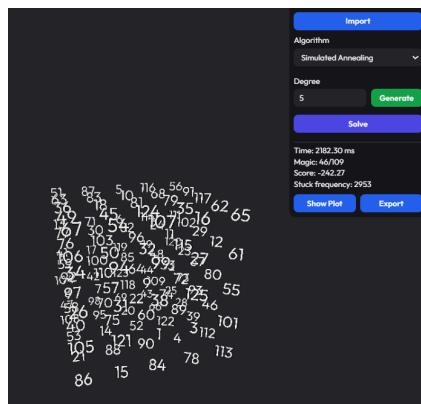
Final state:

Time: 2182.30 ms

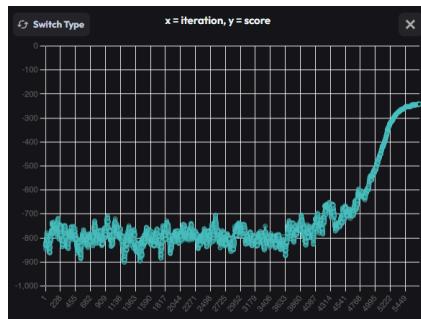
Magic: 46/109

Score: -242.27

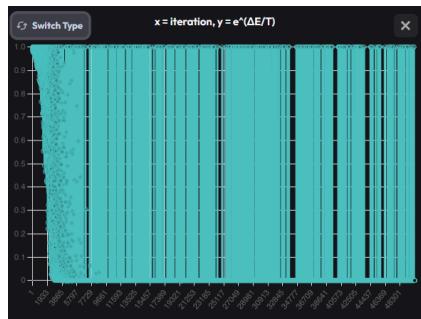
Stuck frequency: 2953



Plot Objective function terhadap iterasi:

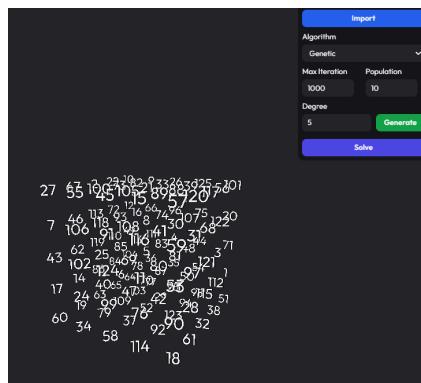


Plot $e^{-(\Delta E/T)}$ terhadap iterasi:

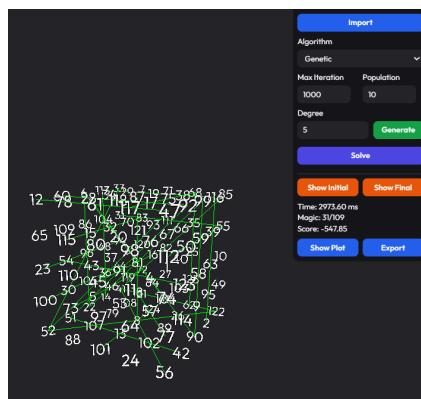


6. Eksperimen Genetic Algorithm (variasi iterasi)

Eksperimen ke-	Visualisasi	Penjelasan Eksperimen
1	Initial state:	Time: 2973.60 ms Magic: 31/109 Score: -547.85 Iteration: 1000 Population: 10



Final state:

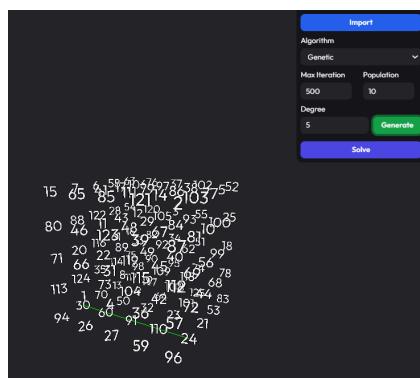


Plot Objective function terhadap iterasi:



2

Initial state:



Time: 1242.90 ms

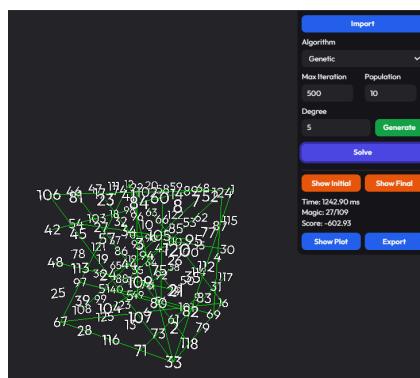
Magic: 27/109

Score: -602.93

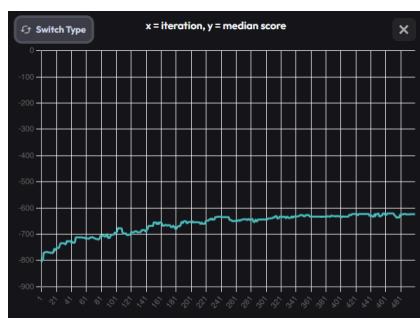
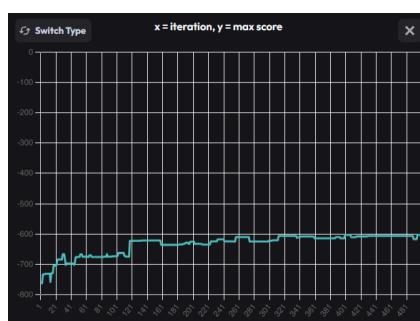
Iteration: 500

Population: 10

Final state:

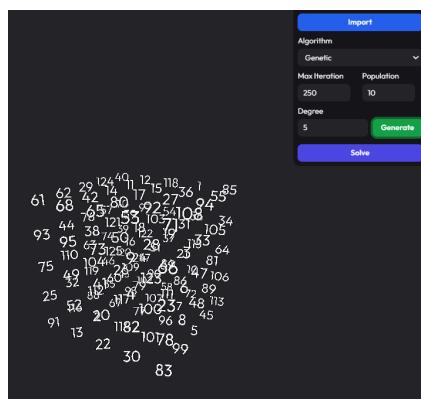


Plot Objective function terhadap iterasi:



3

Initial state:



Time: 429.70 ms

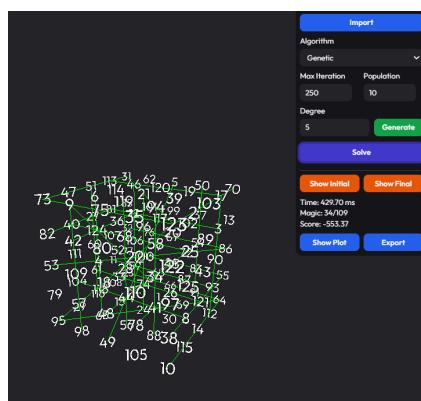
Magic: 34/109

Score: -553.37

Iteration: 250

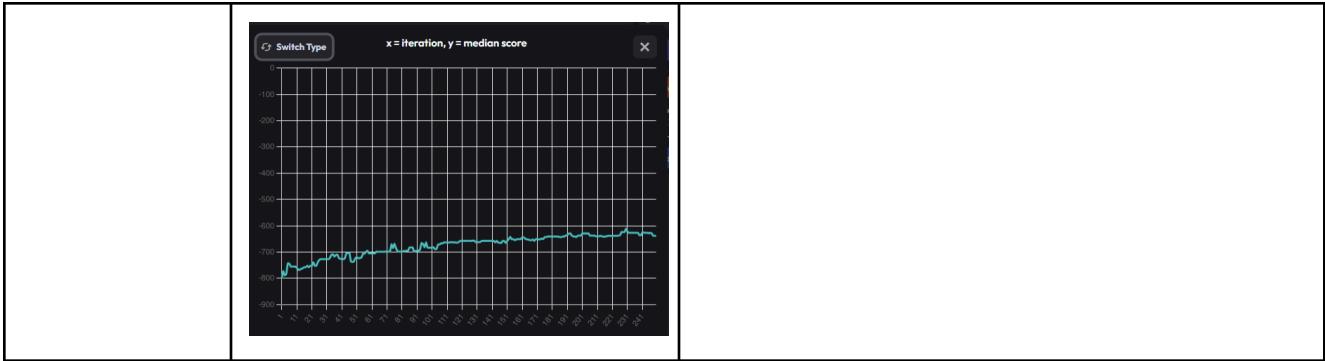
Population: 10

Final state:



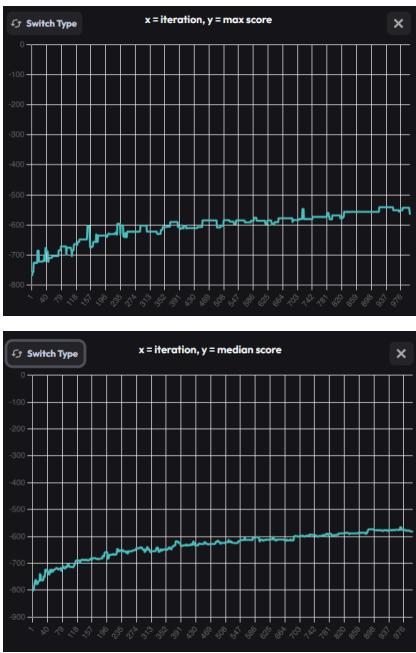
Plot Objective function terhadap iterasi:





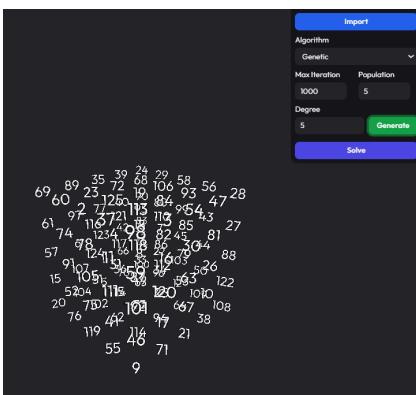
7. Eksperimen Genetic Algorithm (variasi populasi)

Eksperimen ke-	Visualisasi	Penjelasan Eksperimen
1	<p>Initial state:</p> <p>Final state:</p> <p>Plot Objective function terhadap iterasi:</p> <p>Time: 3020.30 ms Magic: 30/109 Score: -566.29 Iteration: 1000 Population: 10</p>	



2

Initial state:



Time: 3229.00 ms

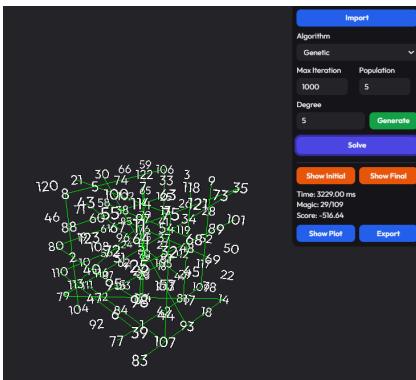
Magic: 29/109

Score: -516.64

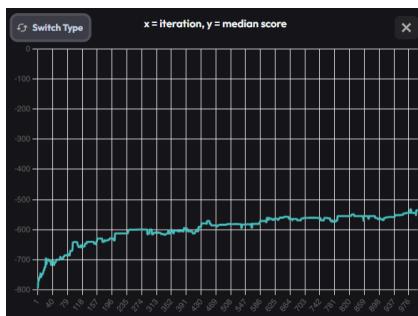
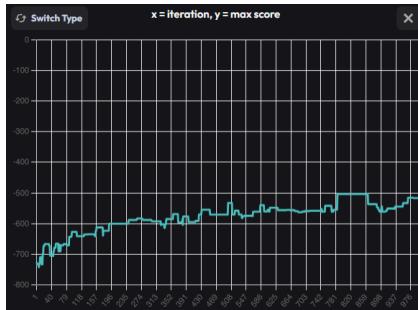
Iteration: 1000

Population: 5

Final state:

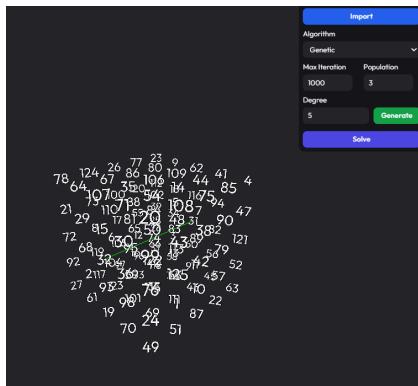


Plot Objective function terhadap iterasi:



3

Initial state:



Time: 2956.70 ms

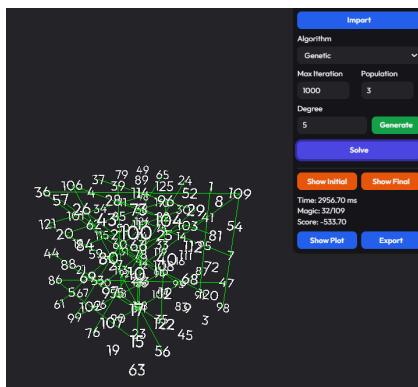
Magic: 32/109

Score: -533.70

Iteration: 1000

Population: 3

Final state:



Plot Objective function terhadap iterasi:



Dari hasil eksperimen di atas, berikut adalah rincian dari masing - masing algoritma terhadap seberapa dekat tiap algoritma mendekati global optima:

1. Random restart hill climb (dengan maksimal restart ditentukan)

Random restart hill climb mendapatkan rata-rata jumlah magic number yang terdekat dengan global maxima (47.3/109), hal ini disebabkan karena algoritma ini melakukan sejumlah iterasi dari steepest ascent dan mengambil hasil yang terbaik dari iterasi tersebut.

2. Steepest ascent hill climb

Algoritma ini menunjukkan performa yang cukup baik (47/109), meskipun masih sangat jauh dari global optima, tetapi dengan waktu yang sangat cepat dapat mencapai objective function yang baik.

3. Simulated Annealing

Algoritma ini memiliki *objective function* terbaik di antara algoritma yang lain, namun memiliki rata-rata jumlah magic number yang lebih sedikit. Algoritma ini dapat melakukan pencarian ke state yang memiliki *objective function* yang lebih kecil untuk menghindari local optima.

4. Stochastic Hill Climbing

Algoritma ini menunjukkan performa yang sama dekatnya dengan simulated annealing dalam mendekati global optima, tetapi juga dengan performa yang jauh lebih cepat dibandingkan algoritma yang lain.

5. Steepest ascent hill climb with sideways move

Algoritma ini menunjukkan performa yang sama dekatnya dengan algoritma steepest ascent hill climb, karena objective function yang digunakan hampir tidak mungkin mendapatkan suksesor dengan value yang sama, sehingga sideways move hampir tidak digunakan.

6. Genetic Algorithm

Algoritma ini memiliki jarak yang paling jauh dengan global optima, dimungkinkan karena algoritma dari konsep ini adalah merging dan mutation dan tidak cocok dengan permasalahan yang ada.

Jika dibandingkan hasil yang didapatkan antara satu dengan yang lainnya, berikut adalah perbedaan antara satu algoritma dengan yang lain

1. Random restart hill climbing unggul dalam hasil akhir, tetapi memiliki beban komputasional yang jauh lebih tinggi, dan cukup konsisten jika melihat hasil dari 3 eksperimen, hal ini disebabkan karena algoritma ini berulang kali iterasi
2. Simulated annealing memaksimalkan keseimbangan antara waktu, beban komputasional, dan hasil yang didapatkan dan mendapatkan hasil akhir yang konsisten.
3. Steepest ascent hill climbing dengan atau tanpa sideways move memiliki performa yang baik tetapi kurang konsisten.
4. Stochastic hill climbing memaksimalkan keseimbangan antara waktu, beban komputasional dan hasil yang didapatkan dan mendapatkan hasil akhir yang konsisten.
5. Genetic algorithm mendapatkan *objective function* yang terburuk diantara algoritma yang lain serta tidak konsisten menghasilkan hasil yang baik sehingga dirasa kurang cocok untuk digunakan pada masalah ini.

Berdasarkan percobaan yang dilakukan, didapat urutan durasi penyelesaian dari yang tercepat sebagai berikut, stochastic hill climbing (1249,93 ms), simulated annealing (2197,26 ms), genetic algorithm (2308,7 ms), sideways hill climbing (2390,93 ms), steepest hill climbing (3452,13 ms), random restart hill climbing dengan 5 restart (19538,93 ms). Namun, pada perhitungan waktu, perlu diperhatikan bahwa pada beberapa algoritma, jumlah iterasi ditentukan secara manual dan kecepatannya pun bergantung pada input dari pengguna.

Pada genetic algorithm, relasi atau pengaruh antara banyaknya iterasi dan jumlah populasi sangat tergantung pada fungsi objektif, mekanisme crossover, dan mekanisme mutasi sebab hal-hal tersebut menentukan seberapa cepat genetic algorithm akan konvergen pada suatu solusi serta hal-hal yang berkaitan dengan fluktuasi yang dapat menyebabkan genetic algorithm untuk melakukan eksplorasi-eksplorasi elemen-elemen yang baru. Dengan menggunakan *setup* genetic algorithm yang kami lakukan, diperoleh kecenderungan hasil yang membaik ketika jumlah iterasi maksimal ditingkatkan dengan jumlah populasi yang konstan. Diperoleh juga kecenderungan untuk konvergen pada suatu solusi yang bagus yang cukup lama ketika jumlah populasi dinaikkan; hal ini terjadi karena butuh waktu yang lebih lama agar solusi-solusi yang bagus tersebar secara merata pada populasi.

III. Kesimpulan dan Saran

Berdasarkan implementasi algoritma local search untuk pencarian solusi *diagonal magic cube*, diperoleh bahwa algoritma ini cukup cepat dan baik untuk mencari solusi yang mendekati *global optima*. Algoritma ini berhasil menemukan konfigurasi yang memenuhi sekitar setengah dari keseluruhan kriteria. Selain itu,

algoritma *local search* juga berhasil menemukan konfigurasi-konfigurasi tersebut dengan waktu yang relatif cepat dibandingkan dengan *exhaustive/complete search*.

Hasil eksperimen menunjukkan memperlihatkan berbagai aspek dari setiap algoritma local search dengan berbagai parameter yang divariasikan. Hal-hal seperti mengubah jumlah iterasi menjadi lebih banyak dan yang lain-lainnya cenderung mendapatkan solusi lebih optimal dengan waktu yang lebih lama. Setiap algoritma local search mempunyai keunikan tersendiri sehingga didapatkan bahwa semua algoritma tersebut tidak memiliki performa dan kecepatan yang sama.

Sebagai saran, berbagai hal-hal dapat dilakukan untuk melakukan optimisasi pencarian seperti menyesuaikan parameter (meningkatkan jumlah iterasi, meningkatkan max restart, dan yang lain-lainnya), menggunakan heuristic-heuristic (diantaranya yang paling sederhana yang sudah dilakukan seperti menggunakan rumus untuk setiap sequence), dan eksperimen lanjut mengenai algoritma mana yang lebih cocok.

IV. Pembagian Tugas

No.	Nim	Nama	Tugas
1	13522066	Nyoman Ganadipa Narayana	Algoritma, Integrasi, Laporan
2	13522084	Dhafin Fawwaz Ikramullah	FE, Algoritma, Integrasi, Laporan
3	13522107	Rayendra Althaf Taraka Noor	Algoritma, Integrasi, Laporan
4	13522109	Azmi Mahmud Bazeid	Algoritma, Integrasi, Laporan

V. Referensi

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

<https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>

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