

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



Disusun Oleh:
Dewantoro Triatmojo 13522011
Renaldy Arief Susanto 13522022
Moh Fairuz Alauddin Yahya 13522057
Rayhan Fadlan Azka 13522095

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

2024

Daftar Isi

Daftar Isi	3
Bab 1: Deskripsi Persoalan	4
Bab 2: Pembahasan	6
Bab 3: Kesimpulan dan Saran	7
Referensi	8
Lampiran	9

Bab 1: Deskripsi Persoalan

Diagonal magic cube merupakan kubus yang tersusun dari angka 1 hingga m^3 tanpa pengulangan dengan m 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.
- Kubus tersebut akan memiliki *magic number* bernilai:

$$M_3(m) = m(m^3 + 1)/2$$

- Jumlah angka-angka untuk setiap baris sama dengan *magic number*.

Pada kubus tersebut, terdapat m^2 baris.

- Jumlah angka-angka untuk setiap kolom sama dengan *magic number*.

Pada kubus tersebut, terdapat m^2 kolom.

- Jumlah angka-angka untuk setiap tiang sama dengan *magic number*.

Pada kubus tersebut, terdapat m^2 tiang.

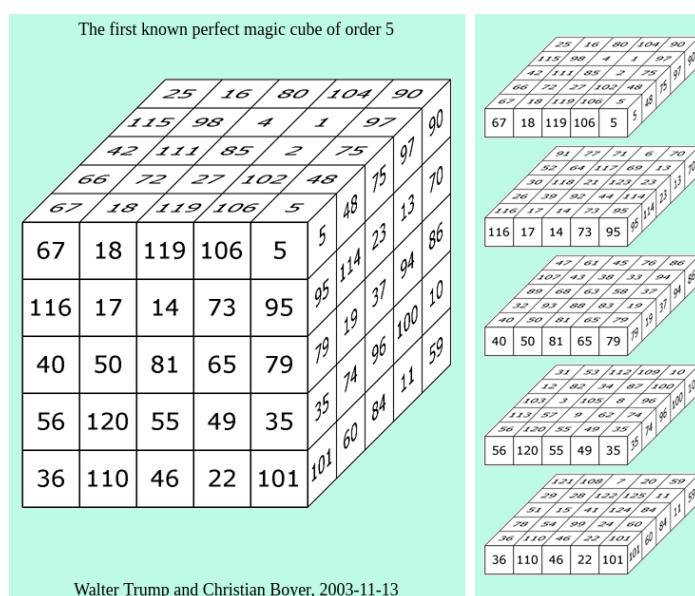
- Jumlah angka-angka untuk seluruh diagonal ruang pada kubus sama dengan *magic number*.

Pada kubus tersebut, terdapat 4 diagonal ruang.

- Jumlah angka-angka untuk seluruh diagonal pada suatu potongan bidang dari kubus sama dengan *magic number*.

Pada kubus tersebut, terdapat $6m$ diagonal bidang.

Maka totalnya, terdapat $N = 3m^2 + 6m + 4$ garis X_i yang jumlahnya harus sama dengan *magic number*.



Ilustrasi *Diagonal Magic Cube* dengan $m = 5$
(Source: Water Trump and Christian Boyer, 2003-11-13)

Pada tugas ini, terdapat beberapa batasan pada penyelesaian persoalan *diagonal magic cube* menggunakan algoritma *Local Search*.

- *Initial state* dari suatu kubus adalah susunan angka 1 hingga 125 secara acak.
- 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).

Bab 2: Pembahasan

2.1 Pemilihan Objective Function

Objective function memberikan nilai seberapa dekat suatu kubus dengan rusuk berukuran m dengan sifat *diagonal magic cube*. Untuk kasus persoalan $m = 5$, maka berdasarkan rumus pada bab sebelumnya berlaku nilai berikut:

$$M_3(5) = 5(5^3 + 1)/2 = 315$$

$$N = 3 * 5^2 + 6 * 5 + 4 = 109$$

Objective function salah satunya dapat dirumuskan sebagai berikut

$$F_{obj} = \sum_{i=1}^N f(X_i) = \sum_{i=1}^{109} f(X_i)$$

$$f(X_i) = 0 \text{ jika } X_i = M_3(5) = 315$$

$$f(X_i) = -1 \text{ jika } X_i \neq M_3(5) = 315$$

Dimana X_i merupakan jumlah angka pada suatu garis yang harus memenuhi *magic number*.

Perhatikan bahwa $-109 \leq F_{obj} \leq 0$. Fungsi objektif tersebut dipilih dengan beberapa alasan:

- Dipilih pembobotan 0 jika $X_i = M_3(5) = 315$ dan -1 jika $X_i \neq M_3(5) = 315$ agar *global maximum* (solusi) memiliki nilai fungsi objektif $F_{obj} = 0$ dan selain itu memiliki nilai $F_{obj} < 0$. Semakin negatif fungsi objektif dari suatu *state*, artinya *state* kubus semakin jauh dari *state diagonal magic cube*.
- Lebih memilih menggunakan nilai diatas dari pada menghitung
$$-\sum_{i=1}^N |X_i - M_3(5)| = -\sum_{i=1}^{109} |X_i - 315|$$
karena sebenarnya yang dilihat untuk sebuah garis pada kubus dinyatakan sebagai memenuhi syarat *magic cube* adalah apakah garis $X_i = M_3(5) = 315$ atau tidak, bukan seberapa jauh deviasinya dari nilai *magic number* $M_3(5) = 315$.
- Lebih memilih menggunakan nilai diatas dari pada menghitung simpangan rata-rata atau deviasi karena alasan poin kedua dan kedua nilai tersebut bisa bernilai desimal dan nilai desimal lebih sulit untuk dibandingkan.

2.2 Kelas MagicCube

Kelas ini merupakan representasi dari *Magic Cube* itu sendiri, memiliki 2 atribut utama yaitu *cube* merupakan *array* 3D yang merepresentasikan *current state* kubus itu sendiri dan

setiap angkanya dan *objval* merupakan *current objective function* dan tambahan *helper* atribut seperti *m* untuk ukuran kubus (dalam implementasi ini $5 \times 5 \times 5$) dan *dirty* sebagai penanda apakah *objval* perlu dihitung ulang.

Core methods yang terdapat dalam kelas MagicCube:

- `initializeCube()`: *Method* ini bertanggung jawab untuk inisialisasi awal kubus dengan mengisi array 3D menggunakan angka 1 hingga m^3 secara acak.
 - `calculateObjectiveFunction()`: *Method* ini menghitung nilai fungsi objektif berdasarkan deviasi total dari magic number. Magic number dihitung dengan rumus $(m * (m^3 + 1))/2$, dan fungsi ini memeriksa jumlah setiap baris, kolom, pilar, dan diagonal untuk menentukan seberapa jauh dari kondisi *ideal magic cube*.
 - `getAllLines()`: *Method* penting yang mengumpulkan semua kemungkinan garis dalam kubus, termasuk baris, kolom, pilar, diagonal ruang, dan diagonal bidang, yang diperlukan untuk pengecekan magic cube.
 - `generateAllSuccessors()`: *Method* yang menghasilkan semua kemungkinan state berikutnya dengan menukar posisi setiap pasangan elemen dalam kubus.
 - `getBestSuccessor()`: *Method* yang mencari successor terbaik berdasarkan nilai fungsi objektif tertinggi dari semua kemungkinan successor.
 - `generateRandomSuccessor()`: *Method* yang menghasilkan successor acak dengan menukar dua posisi acak dalam kubus.

```
// Position type for cube coordinates
type Position = [number, number, number];

// MagicCube class for representing the cube and its basic operations
export class MagicCube {
    private cube: number[][][][];
    private m: number;
    private objval: number;
    private dirty: boolean;

    constructor(initialValue: number[][][][] | null = null) {
        this.m = 5;

        if (initialValue) {
            // Copy the cube
            this.cube = JSON.parse(JSON.stringify(initialValue));
        } else {
            this.cube = [];
        }
    }

    get m() {
        return this.m;
    }

    set m(value: number) {
        if (value < 1 || value > 5) {
            throw new Error('m must be between 1 and 5');
        }
        this.m = value;
    }

    get cube() {
        return this.cube;
    }

    set cube(value: number[][][][]) {
        this.cube = value;
    }

    get objval() {
        return this.objval;
    }

    set objval(value: number) {
        this.objval = value;
    }

    get dirty() {
        return this.dirty;
    }

    set dirty(value: boolean) {
        this.dirty = value;
    }

    // Basic operations
    rotateFace(face: string, clockwise: boolean) {
        // Implementation
    }

    flipFace(face: string) {
        // Implementation
    }

    rotateCube(clockwise: boolean) {
        // Implementation
    }

    flipCube() {
        // Implementation
    }

    // Utility methods
    clone(): MagicCube {
        const clone = new MagicCube();
        clone.cube = JSON.parse(JSON.stringify(this.cube));
        clone.m = this.m;
        clone.objval = this.objval;
        clone.dirty = this.dirty;
        return clone;
    }

    toJSON(): string {
        return JSON.stringify(this.cube);
    }

    toString(): string {
        return `MagicCube(m=${this.m}, cube=${this.cube}, objval=${this.objval}, dirty=${this.dirty})`;
    }
}
```

```
    this.initializeCube();
}

this.dirty = true;
this.objval = this.calculateObjectiveFunction();
// console.log("Created a cube with first arr:")
// console.log(this.cube[0][0]);
}

public getElement(pos: Position) {
    const [i, j, k] = pos;
    return this.cube[i][j][k];
}

public getCube(): number[][][] {
    return this.cube;
}

public setElement(pos: Position, val: number) {
    const [i, j, k] = pos;
    this.cube[i][j][k] = val;
    this.dirty = true;
}

public initializeCube(): void {
    // Create a list of numbers from 1 to m^3
    const numbers = Array.from(
        { length: Math.pow(this.m, 3) },
        (_, i) => i + 1
    );

    // Shuffle the numbers
    for (let i = numbers.length - 1; i > 0; i--) {
        const j = Math.floor(Math.random() * (i + 1));
        [numbers[i], numbers[j]] = [numbers[j], numbers[i]];
    }

    // Initialize 3D array
    this.cube = Array(this.m)
```

```

    .fill(null)
    .map(() =>
      Array(this.m)
        .fill(null)
        .map(() => Array(this.m).fill(null)))
  );
}

// Fill the cube with shuffled numbers
let index = 0;
for (let i = 0; i < this.m; i++) {
  for (let j = 0; j < this.m; j++) {
    for (let k = 0; k < this.m; k++) {
      this.cube[i][j][k] = numbers[index++];
    }
  }
}
}

public swap(pos1: Position, pos2: Position): void {
  this.dirty = true;
  const [i1, j1, k1] = pos1;
  const [i2, j2, k2] = pos2;
  const temp = this.cube[i1][j1][k1];
  this.cube[i1][j1][k1] = this.cube[i2][j2][k2];
  this.cube[i2][j2][k2] = temp;
}

public getAllLines(): number[][] {
  const lines: number[][] = [];

  // Rows
  for (let i = 0; i < this.m; i++) {
    for (let j = 0; j < this.m; j++) {
      const row = Array(this.m)
        .fill(0)
        .map((_, k) => this.cube[i][j][k]);

      lines.push(row);
    }
  }
}

```

```

}

// Columns
for (let i = 0; i < this.m; i++) {
  for (let k = 0; k < this.m; k++) {
    const column = Array(this.m)
      .fill(0)
      .map((_, j) => this.cube[i][j][k]);
    lines.push(column);
  }
}

// Pillars
for (let j = 0; j < this.m; j++) {
  for (let k = 0; k < this.m; k++) {
    const pillar = Array(this.m)
      .fill(0)
      .map((_, i) => this.cube[i][j][k]);
    lines.push(pillar);
  }
}

// Space diagonals
const spaceDiagonal1 = Array(this.m)
  .fill(0)
  .map((_, i) => this.cube[i][i][i]);
const spaceDiagonal2 = Array(this.m)
  .fill(0)
  .map((_, i) => this.cube[i][i][this.m - 1 - i]);
const spaceDiagonal3 = Array(this.m)
  .fill(0)
  .map((_, i) => this.cube[i][this.m - 1 - i][i]);
const spaceDiagonal4 = Array(this.m)
  .fill(0)
  .map((_, i) => this.cube[this.m - 1 - i][i][i]);
lines.push(spaceDiagonal1, spaceDiagonal2, spaceDiagonal3,
spaceDiagonal4);

// Plane diagonals

```

```

// Frontal planes (yz planes)
for (let i = 0; i < this.m; i++) {
  const diagonal1 = Array(this.m)
    .fill(0)
    .map((_, j) => this.cube[i][j][j]);
  const diagonal2 = Array(this.m)
    .fill(0)
    .map((_, j) => this.cube[i][j][this.m - 1 - j]);
  lines.push(diagonal1, diagonal2);
}

// Sagittal planes (xz planes)
for (let j = 0; j < this.m; j++) {
  const diagonal3 = Array(this.m)
    .fill(0)
    .map((_, i) => this.cube[i][j][i]);
  const diagonal4 = Array(this.m)
    .fill(0)
    .map((_, i) => this.cube[i][j][this.m - 1 - i]);
  lines.push(diagonal3, diagonal4);
}

// Horizontal planes (xy planes)
for (let k = 0; k < this.m; k++) {
  const diagonal5 = Array(this.m)
    .fill(0)
    .map((_, i) => this.cube[i][i][k]);
  const diagonal6 = Array(this.m)
    .fill(0)
    .map((_, i) => this.cube[i][this.m - 1 - i][k]);
  lines.push(diagonal5, diagonal6);
}

return lines;
}

public calculateObjectiveFunction(): number {

if (!this.dirty) return this.objval;

```

```
this.dirty = false;

const magicNumber = (this.m * (Math.pow(this.m, 3) + 1)) / 2;
let totalDeviation = 0;

const allLines = this.getAllLines();
for (const line of allLines) {
    const lineSum = line.reduce((a, b) => a + b, 0);
    totalDeviation += lineSum === magicNumber ? 0 : -1;
}

this.objval = totalDeviation;
return totalDeviation;
}

public getAllPositions(): Position[] {
    const positions: Position[] = [];
    for (let i = 0; i < this.m; i++) {
        for (let j = 0; j < this.m; j++) {
            for (let k = 0; k < this.m; k++) {
                positions.push([i, j, k]);
            }
        }
    }
    return positions;
}

public clone(): MagicCube {
    return new MagicCube(this.cube);
}

public generateAllSuccessors(): MagicCube[] {
    const successors: MagicCube[] = [];
    const positions = this.getAllPositions();

    for (let i = 0; i < positions.length; i++) {
        for (let j = i + 1; j < positions.length; j++) {
            const newCube = this.clone();
            newCube.swap(positions[i], positions[j]);
            successors.push(newCube);
        }
    }
    return successors;
}
```

```
        successors.push(newCube);
    }
}

return successors;
}

public getBestSuccessor(): MagicCube {
    const successors = this.generateAllSuccessors();
    let bestSuccessor = successors[0];
    let bestValue = bestSuccessor.calculateObjectiveFunction();

    for (const successor of successors) {
        const value = successor.calculateObjectiveFunction();
        if (value > bestValue) {
            bestSuccessor = successor;
            bestValue = value;
        }
    }

    return bestSuccessor;
}

public generateRandomSuccessor(): MagicCube {
    const positions = this.getAllPositions();
    const pos1 = positions[Math.floor(Math.random() * positions.length)];
    let pos2;
    do {
        pos2 = positions[Math.floor(Math.random() * positions.length)];
    } while (this.isPositionEqual(pos1, pos2));

    const newCube = this.clone();
    newCube.swap(pos1, pos2);
    return newCube;
}

public isPositionEqual(pos1: Position, pos2: Position): boolean {
    return pos1[0] === pos2[0] && pos1[1] === pos2[1] && pos1[2] ===
pos2[2];
}
```

```
    }
}
```

2.3 Kelas LocalSearch

Kelas LocalSearch merupakan kelas abstrak yang merupakan *base class* untuk semua algoritma local search dalam mencari solusi Magic Cube. Kelas ini menyediakan abstraksi fungsionalitas umum yang diperlukan untuk analisis proses pencarian solusi. Kelas ini memiliki beberapa atribut utama: *states* yang menyimpan riwayat *state Magic Cube* selama pencarian, *objectiveFunctionPlot* untuk merekam nilai fungsi objektif pada setiap iterasi, *iterationCount* untuk menghitung jumlah iterasi, serta atribut waktu untuk mengukur durasi eksekusi.

Core methods dalam kelas LocalSearch meliputi:

- Method akses dasar seperti `getInitialState()`, `getFinalState()`, `getFinalObjectiveFunction()`, dan `getStates()` yang menyediakan informasi tentang state-state selama proses pencarian.
- Method plotting dan agregasi data:
 - `getObjectiveFunctionPlot()`: Mendapatkan plot fungsi objektif
 - `aggregatePlotData()`: Method protected yang mengagregasi data plot berdasarkan monotonitasnya
 - `getAggregatedObjectiveFunctionPlot()`: Menghasilkan plot yang telah diagregasi
- Method helper protected untuk manajemen state dan pengukuran:
 - `addStateEntry()`: Menambahkan state baru ke dalam riwayat
 - `addObjectiveFunctionPlotEntry()`: Mencatat nilai fungsi objektif
 - `startTimer()`, `endTimer()`: Mengukur waktu eksekusi
 - `addIterationCount()`: Menambah hitungan iterasi

```
import { MagicCube } from "./MagicCube";
import { Plot, PlotData } from "./Plot";

// LocalSearch class for handling search operations
export abstract class LocalSearch {
  protected states: MagicCube[];
  protected objectiveFunctionPlot: Plot<number, number>;
  protected iterationCount: number;
  private startTime: number;
  private endTime: number;
  protected duration: number;
```

```
constructor(initialState: MagicCube) {
    this.states = [initialState.clone()];
    this.objectiveFunctionPlot = {
        labelX: "Iteration",
        labelY: "Objective Function",
        data: [],
    };
    this.iterationCount = 0;
    this.startTime = 0;
    this.endTime = 0;
    this.duration = 0;
}

public getInitialState(): MagicCube {
    return this.states[0];
}

public getFinalState(): MagicCube {
    return this.states[this.states.length - 1];
}

public getFinalObjectiveFunction(): number {
    return this.getFinalState().calculateObjectiveFunction();
}

public getStates(): number[][][][][] {
    const matrices = this.states.map((state) => state.getCube());
    return matrices;
}

public getDuration() {
    return this.duration;
}

public getIterationCount() {
    return this.iterationCount;
}

public getObjectiveFunctionPlot() {
```

```

        return this.objectiveFunctionPlot;
    }

    /**
     * Get the aggregated objective function plot data
     * Grouped by monotonicity of the objective function over the
     * iterations
     * @param plotData Plot<number, number>[]
     * @param n number (minimum number of iterations to take for the middle
     * values)
     * @returns Plot<number, number>[]
     */
    protected aggregatePlotData(
        plotData: PlotData<number, number>[],
        n: number
    ): PlotData<number, number>[] {
        type Range = [number, number];

        if (plotData.length < 2) {
            return plotData.length ? [{ x: 0, y: 0 }] : [];
        }

        // Group by monotonicity
        const ranges: Range[] = [];
        let start = 0;
        let currentTrend: "increasing" | "decreasing" | "constant" =
            plotData[1] > plotData[0]
                ? "increasing"
                : plotData[1] < plotData[0]
                ? "decreasing"
                : "constant";

        for (let i = 1; i < plotData.length; i++) {
            let newTrend: typeof currentTrend;

            if (i < plotData.length - 1) {
                if (plotData[i + 1] > plotData[i]) {
                    newTrend = "increasing";
                } else if (plotData[i + 1] < plotData[i]) {
                    newTrend = "decreasing";
                } else {
                    newTrend = "constant";
                }
            }
        }
    }
}

```

```

        newTrend = "decreasing";
    } else {
        newTrend = "constant";
    }

    if (newTrend !== currentTrend) {
        ranges.push([start, i]);
        start = i;
        currentTrend = newTrend;
    }
} else {
    // Handle the last range
    ranges.push([start, i]);
}
}

// For each range, take the first, &middle values and last value (if
final range)
const aggregatedPlotData: PlotData<number, number>[] = [];
for (let rIdx = 0; rIdx < ranges.length; rIdx++) {
    const [startIdx, endIdx] = ranges[rIdx];

    aggregatedPlotData.push({
        x: plotData[startIdx].x,
        y: plotData[startIdx].y,
    });

    if (rIdx == ranges.length - 1) {
        aggregatedPlotData.push({
            x: plotData[endIdx].x,
            y: plotData[endIdx].y,
        });
    }
}

// Middle values
// Take a minimum of n iterations for the middle values, or all if
less than 10
if (endIdx - startIdx - 1 < n) {
    // If less than n iterations, take all
}

```

```

        for (let i = startIdx + 1; i < endIdx; i++) {
            aggregatedPlotData.push({
                x: plotData[i].x,
                y: plotData[i].y,
            });
        }
    } else {
        const delta = Math.floor((endIdx - startIdx) / n);
        for (let i = startIdx + 1; i < endIdx; i += delta) {
            aggregatedPlotData.push({
                x: plotData[i].x,
                y: plotData[i].y,
            });
        }
    }
}

return aggregatedPlotData;
}

public getAggregatedObjectiveFunctionPlot(n: number): Plot<number, number> {
    const aggregatedData = this.aggregatePlotData(
        this.objectiveFunctionPlot.data,
        n
    );

    return {
        labelX: this.objectiveFunctionPlot.labelX,
        labelY: this.objectiveFunctionPlot.labelY,
        data: aggregatedData,
    };
}

protected addStateEntry(state: MagicCube): void {
    this.states.push(state.clone());
}

protected addObjectiveFunctionPlotEntry(x: number, y: number): void {

```

```

        this.objectiveFunctionPlot.data.push({ x, y });
    }

protected startTimer(): void {
    this.startTime = performance.now();
}

protected endTimer(): void {
    this.endTime = performance.now();
    this.duration = (this.endTime - this.startTime) / 1000;
}

protected addIterationCount() {
    this.iterationCount++;
}
}

```

2.4 Algoritma Steepest Ascent Hill-climbing

2.4.1 Alur Algoritma Steepest Ascent Hill-climbing

Alur algoritma Steepest Ascent Hill-climbing dalam penyelesaian diagonal magic cube:

1. Inisialisasi initial state dimana kubus terdiri dari suatu susunan angka 1 hingga 125 secara acak sebagai current state. Hitung juga value dari initial state tersebut dengan fungsi objektif.
2. Bangkitkan semua nilai successor dari current state dengan menukar posisi 2 angka pada kubus dan hitung juga value dari successornya dengan fungsi objektif. Pilih successor dengan value tertinggi sebagai neighbor.
3. Terdapat dua kasus:
 - Jika neighbor value \leq current value, maka kembalikan current state.
 - Selain dari itu, ubah current state menjadi neighbor dan ulangi lagi pencarian pada step nomor 2.

2.4.2 Implementasi Algoritma Steepest Ascent Hill-climbing

```

export class SteepestAscent extends LocalSearch{

    private cube : MagicCube;
}

```

```
constructor(cube:MagicCube){
    super(cube);
    this.cube = cube;
}

public solve(): void {

    this.startTimer();

    let currentState = this.cube;

    while(currentState.calculateObjectiveFunction() != 0){
        const nextState = currentState.getBestSuccessor();

        const currentObj = currentState.calculateObjectiveFunction();
        const nextObj = nextState.calculateObjectiveFunction();

        const deltaE = nextObj - currentObj;

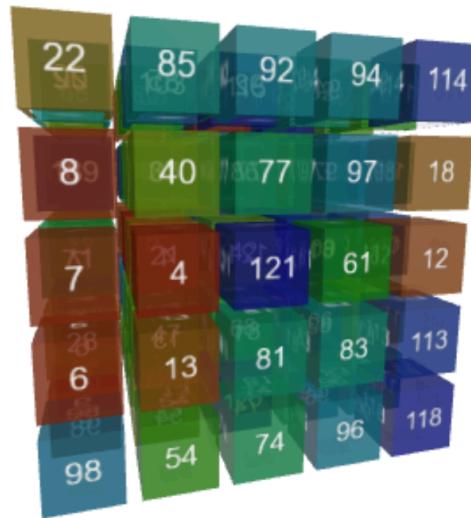
        this.addStateEntry(currentState);
        this.addObjectiveFunctionPlotEntry(this.iterationCount, currentObj);

        if(deltaE > 0){
            this.addIterationCount();
            currentState = nextState;
        }
        else{
            break;
        }
    }
    this.endTimer();
}
```

2.4.4 Hasil Eksperimen

2.4.4.1 Eksperimen 1

State awal kubus :



②

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

State akhir kubus :



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

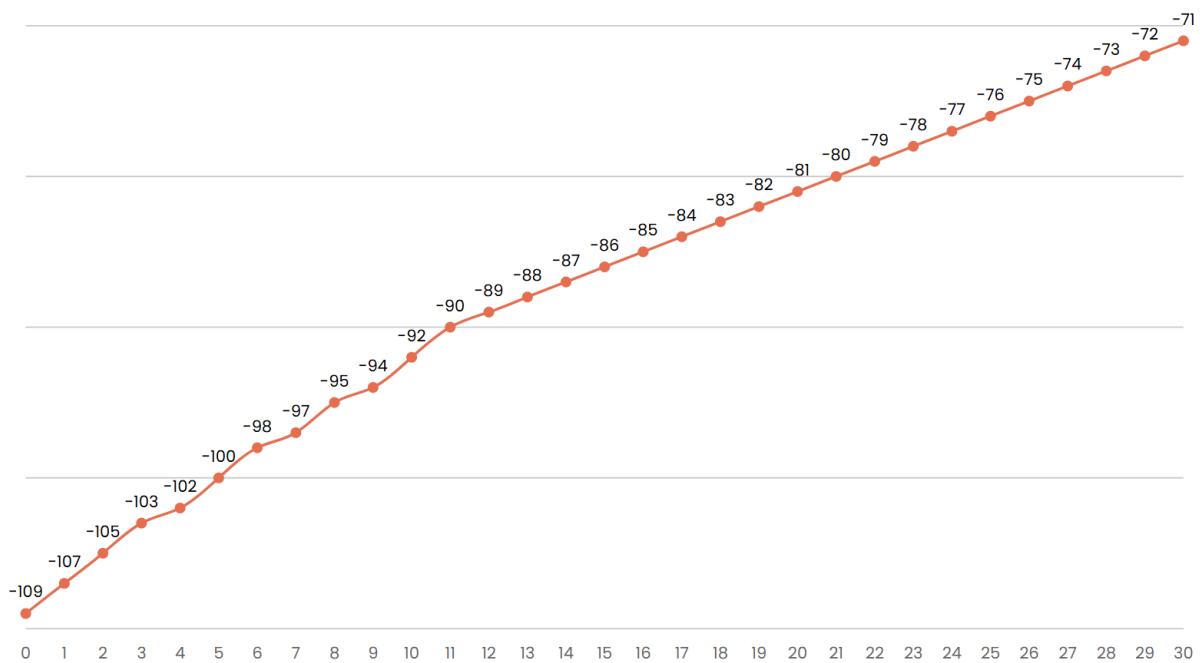
Nilai objective function akhir yang dicapai, durasi pencarian, dan jumlah iterasi :

Duration: 15.39s Final Objective Function Value: -71 Iteration Count: 30

Plot nilai objective function dengan jumlah iterasi :

Objective Function vs Iteration

Algorithm progression over time



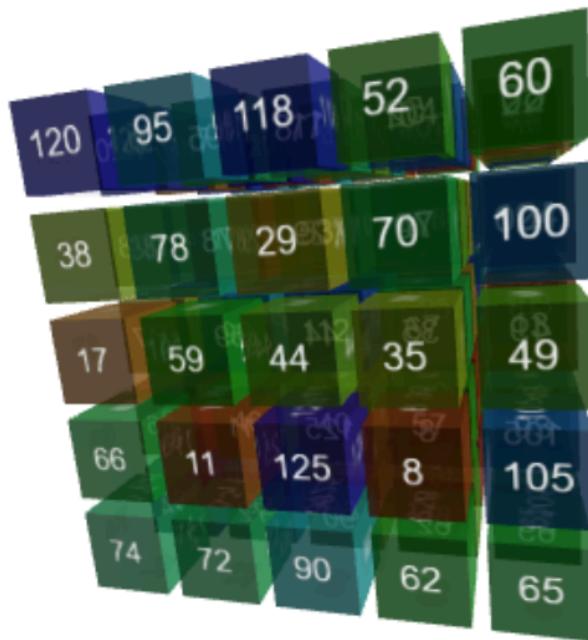
2.4.4.2 Eksperimen 2

State awal :

State akhir :



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



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

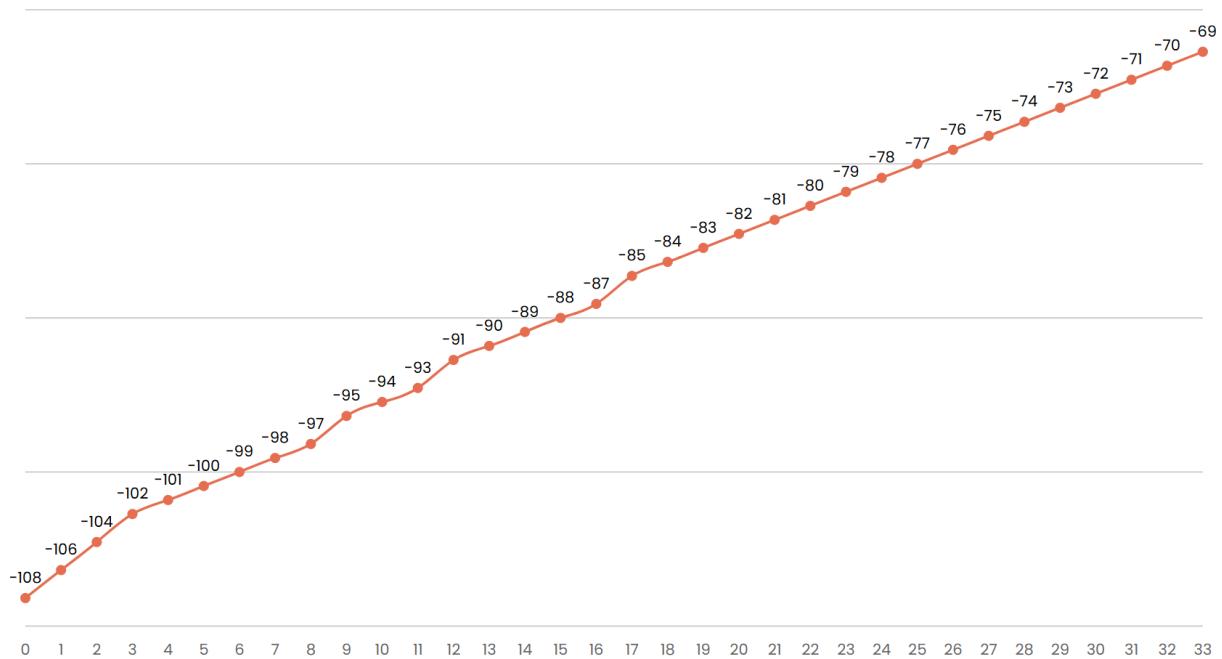
Nilai objective function akhir yang dicapai, durasi pencarian, dan jumlah iterasi :

Duration: 13.52s Final Objective Function Value: -69 Iteration Count: 33

Plot nilai objective function dengan jumlah iterasi :

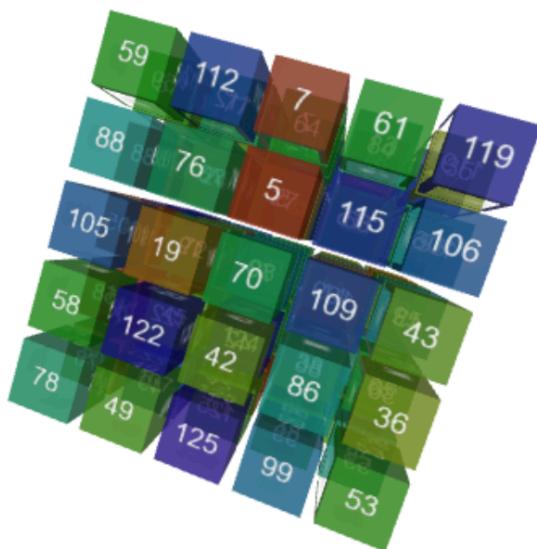
Objective Function vs Iteration

Algorithm progression over time



2.4.4.4 Eksperimen 3

State awal :



12	23	101	12	79	80	69	114	104	67	13	37	60	3	117	57	77	64	84	35	59	112	7	61	119
44	103	143	34	26	25	6	68	87	96	28	71	121	120	62	8	24	27	63	93	88	76	5	115	106
110	108	52	40	29	123	83	91	22	15	118	17	73	92	50	11	72	98	14	95	105	19	70	109	43
54	33	74	116	32	31	81	47	111	21	89	39	75	113	82	1	45	124	38	65	58	122	42	86	36
55	30	66	94	41	90	18	16	97	102	56	100	10	9	46	20	107	51	48	85	78	49	125	99	53

State akhir :



69	41	101	120	79	49	42	83	104	67	13	37	60	3	86	66	77	64	55	35	59	112	7	94	119
72	103	143	34	73	25	90	17	87	96	122	71	121	2	9	8	24	27	63	31	88	76	54	115	106
110	108	52	40	29	93	114	50	22	15	118	68	26	92	11	46	62	98	14	95	32	19	70	109	1
85	33	74	102	111	123	81	125	45	21	6	39	75	113	82	43	45	124	38	65	58	28	42	57	36
117	30	84	61	23	44	18	40	97	116	56	100	10	105	91	20	107	51	48	89	78	80	47	99	53

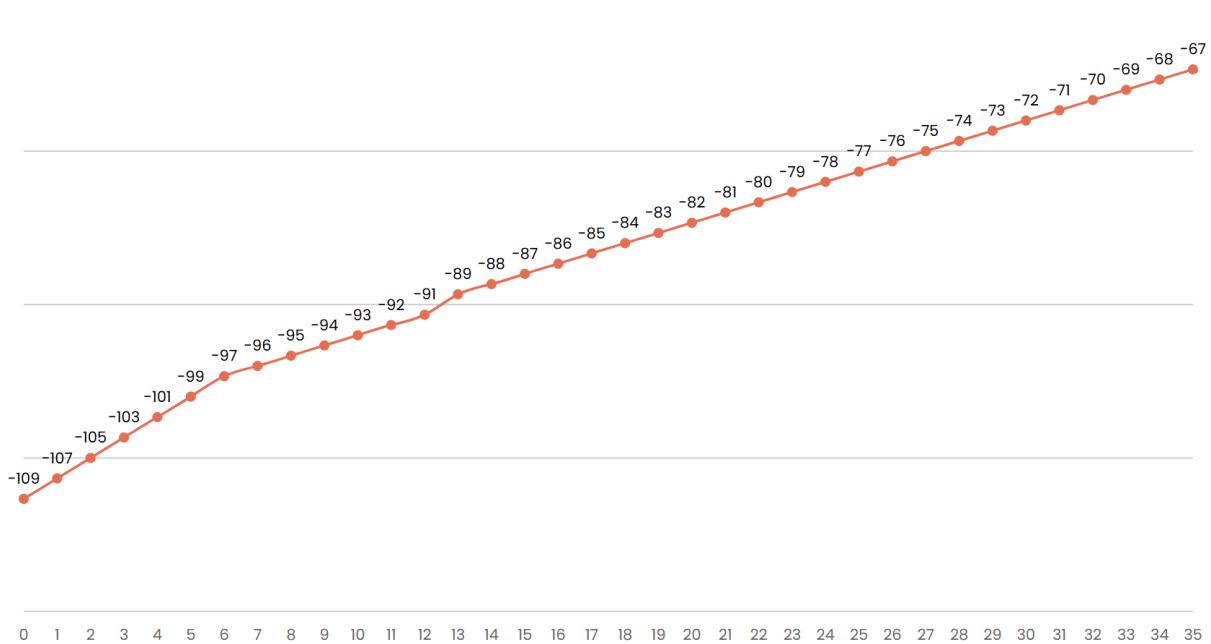
Nilai objective function akhir yang dicapai, durasi pencarian, dan jumlah iterasi :

Duration: 14.38s Final Objective Function Value: -67 Iteration Count: 35

Plot nilai objective function dengan jumlah iterasi :

Objective Function vs Iteration

Algorithm progression over time



2.4.5 Hasil Analisis

Algoritma Steepest Ascent Hill Climbing merupakan algoritma dengan jumlah iterasi paling sedikit dibanding algoritma lainnya, waktu yang dibutuhkan juga tidak terlalu lama (sekitar 15 detik) namun bukan yang tercepat juga. Hasil yang dihasilkan juga tidak mendekati global optimum dengan nilai objective function yang dihasilkan kurang lebih -70 dengan nilai global optimum 0. Hasil objective function dan durasi dari berbagai eksperimen yang dihasilkan oleh algoritma ini juga nilainya konsisten karena tidak terdapat kerandoman pada algoritmanya (selalu mencari best successor, dengan kerandoman hanya pada initial state).

2.5 Algoritma Hill-climbing with Sideways Move

2.5.1 Alur Algoritma Hill-climbing with Sideways Move

Alur algoritma *Hill-climbing with Sideways Move* dalam penyelesaian *diagonal magic cube* mirip dengan 2.4.1, namun tidak terminate ketika melewati lintasan yang “flat” pada kurva fungsi objektif. Berikut algoritmanya:

1. Inisialisasi *initial state* dimana kubus terdiri dari suatu susunan angka 1 hingga 125 secara acak sebagai *current state*. Hitung juga value dari *initial state* tersebut dengan fungsi objektif.
2. Bangkitkan semua *nilai successor* dari *current state* dengan menukar posisi 2 angka pada kubus dan hitung juga value dari *successornya* dengan fungsi objektif. Pilih *successor* dengan *value tertinggi* sebagai *neighbor*.
3. Terdapat dua kasus:
 - Jika *neighbor value < current value*, maka kembalikan *current state*.
 - Selain dari itu, ubah *current state* menjadi *neighbor* dan ulangi lagi pencarian pada step nomor 2

2.5.2 Implementasi Algoritma Hill-climbing with Sideways Move

```
export class SidewaysMove extends LocalSearch {
  private cube : MagicCube;

  private visitedStates = new Set<string>();

  constructor(cube:MagicCube){
    super(cube);
    this.cube = cube;

  }

  private serializeState(state : MagicCube) : string {
    return JSON.stringify(state);
  }

  public solve(maxSideways : number = 100) : void {
    this.startTimer();

    let currentState = this.cube;
    let currentSideways = 0;
```

```
    this.visitedStates.add(this.serializeState(currentState));

    while(currentState.calculateObjectiveFunction() != 0 &&
currentSideways < maxSideways){

        let idx = 0;
        const magicCubes = currentState.getSortedSuccessors();
        let nextState = magicCubes[idx];
        while(this.visitedStates.has(this.serializeState(nextState))){
            idx++;
            nextState = magicCubes[idx];
        }

        const currentObj = currentState.calculateObjectiveFunction();
        const nextObj = nextState.calculateObjectiveFunction();

        const deltaE = nextObj - currentObj;

        this.addStateEntry(currentState);
        this.addObjectiveFunctionPlotEntry(this.iterationCount,
currentObj);

        const serializedNextState = this.serializeState(nextState);

        if(deltaE >= 0){
            if(deltaE == 0){
                currentSideways++;
            }
            this.addIterationCount();
            currentState = nextState;

            this.visitedStates.add(serializedNextState);
        }
        else{
            break;
        }
    }
}
```

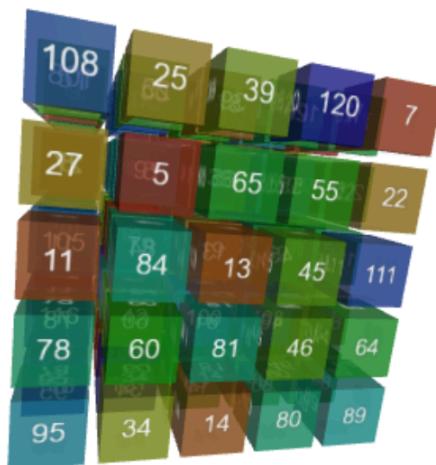
```
        }
        this.endTimer();
    }
```

2.5.3 Hasil Eksperimen

2.6.3.1 Eksperimen 1

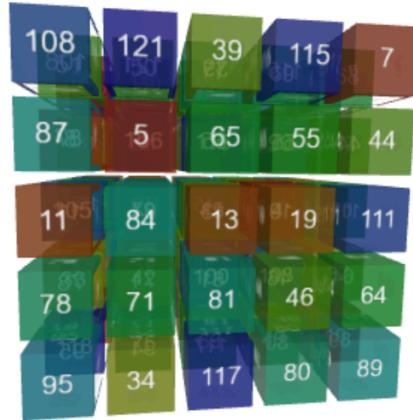
Maximum sideways : 10

State awal :



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

State akhir :



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

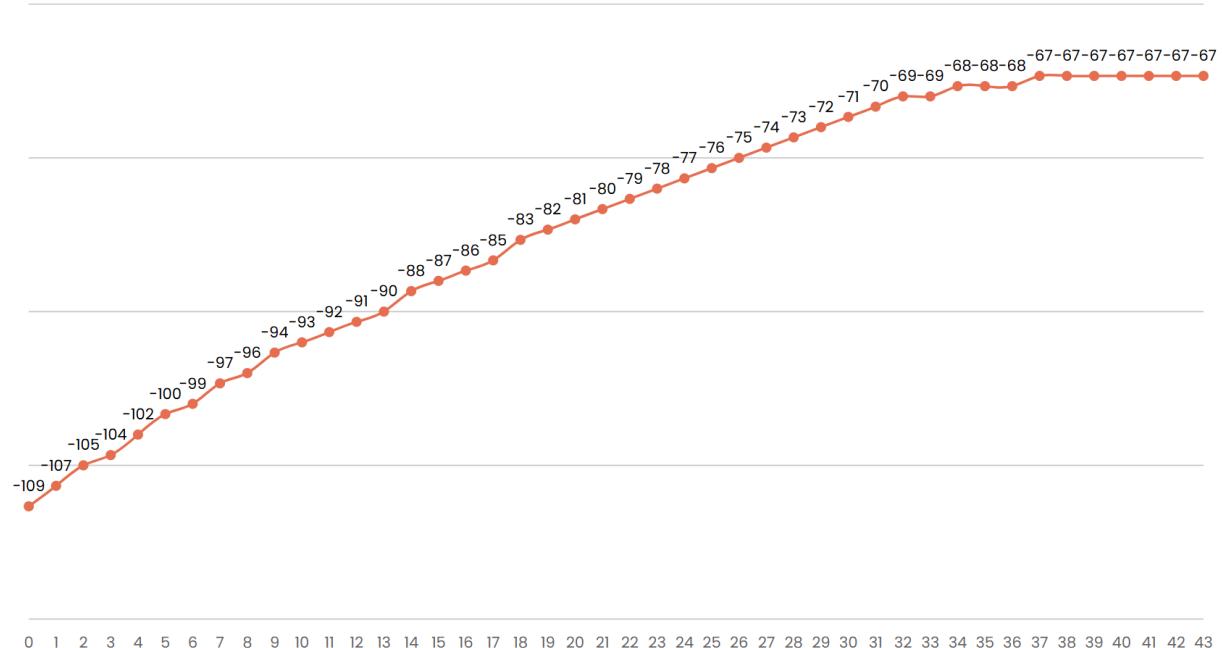
Nilai objective function akhir yang dicapai, durasi pencarian, dan jumlah iterasi :

Duration: 21.18s Final Objective Function Value: -67 Iteration Count: 44

Plot nilai objective function dengan jumlah iterasi :

Objective Function vs Iteration

Algorithm progression over time



2.6.3.2 Eksperimen 2

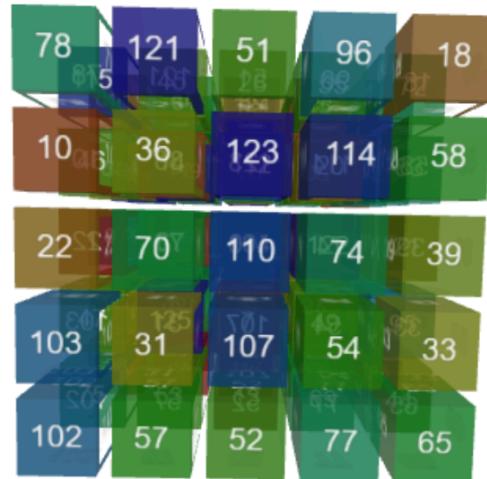
Max sideways : 40

State awal :



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

State akhir :



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

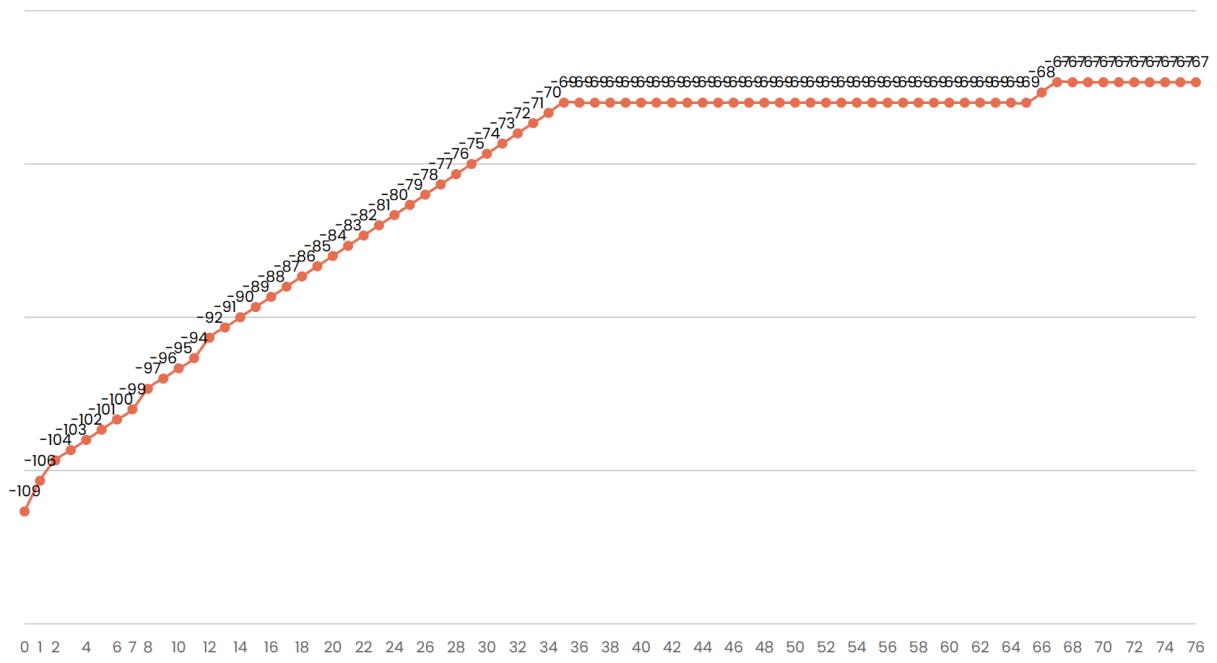
Nilai objective function akhir yang dicapai, durasi pencarian, dan jumlah iterasi :

Duration: 33.99s Final Objective Function Value: -67 Iteration Count: 77

Plot nilai objective function dengan jumlah iterasi :

Objective Function vs Iteration

Algorithm progression over time



2.6.3.3 Eksperimen 3

Maximum sideways : 100

State awal :



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

State akhir :

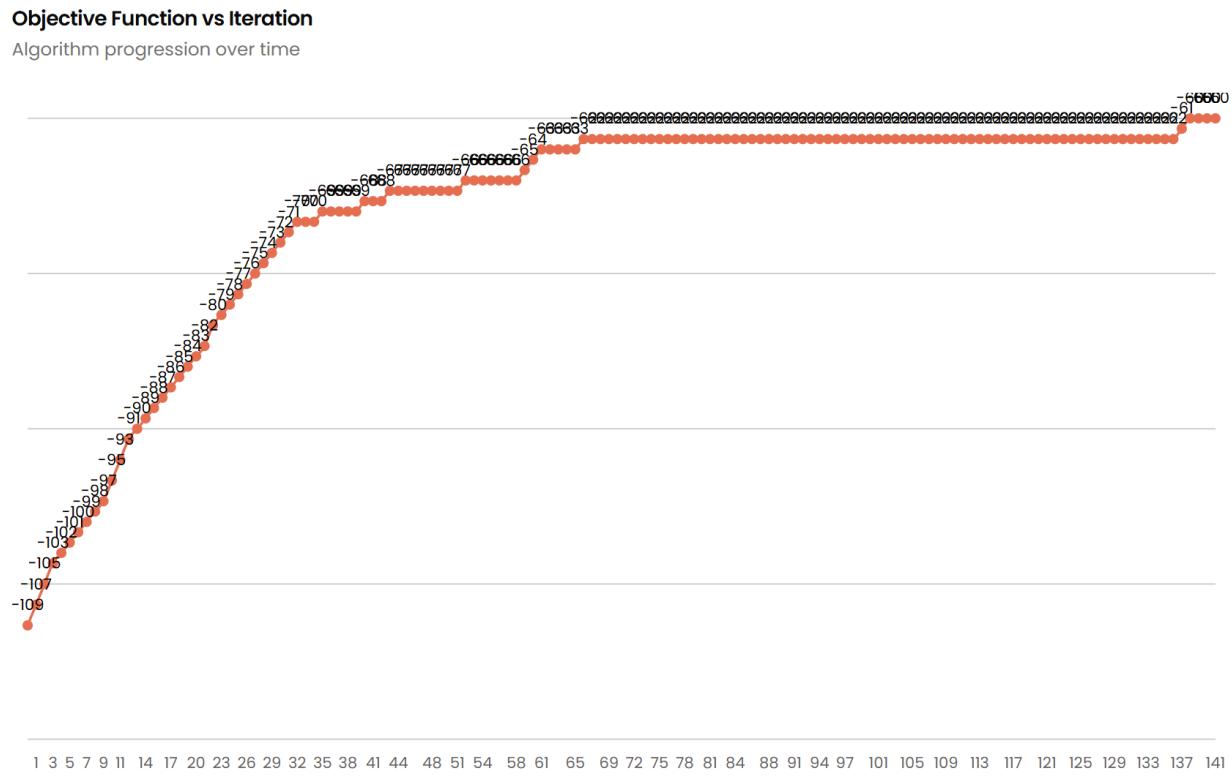


101	56	9	53	96	1	11	40	20	87	113	100	54	22	26	17	62	30	67	31	83	86	34	97	15
104	25	36	42	108	98	35	73	74	28	79	8	46	105	77	91	2	122	82	12	85	121	38	23	48
13	109	92	95	6	118	78	75	41	3	4	69	114	71	57	61	123	18	63	50	119	116	52	110	47
90	81	65	37	45	43	24	16	115	117	51	49	107	103	5	19	89	72	33	102	7	124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

Nilai objective function akhir yang dicapai, durasi pencarian, dan jumlah iterasi :

Duration: 64.82s Final Objective Function Value: -60 Iteration Count: 142

Plot :



2.5.4 Hasil Analisis

Sideways Hill climbing cara kerjanya sangat mirip dengan Steepest Ascent Hill Climbing dengan perbedaan ia memperbolehkan pindah ke state yang memiliki objective function yang sama. Hal ini terbukti membuat hasil yang didapatkan lebih baik seiring meningkatnya maksimum sideways yang diperbolehkan. Dalam algoritma ini kami menambahkan heuristics yaitu suatu state tidak dapat berpindah ke state yang sudah dikunjungi sebelumnya, hal ini dilakukan agar pencarian tidak berputar-putar hanya di dua state ketika mencapai local maxima, hal ini terbukti pada eksperimen 3 dimana pencarian mencapai lokal maksima yang berbeda berkali-kali.

Algoritma ini lebih baik dari Steepest Ascent dalam mencari nilai objective function tertinggi karena algoritma tidak langsung berhenti ketika mencapai lokal maksima, akan tetapi menambah jumlah iterasi dan durasi pencarian.

Pertambahan jumlah sideways move yang boleh dilakukan membuat algoritma dapat menemukan hasil yang lebih baik, seperti pada eksperimen 3 sebelum mencapai lokal maksimum terakhir, state terjebak di lokal maksimum yang panjang. Peningkatan jumlah sideways move yang diperbolehkan memiliki tradeoff yaitu jumlah iterasi lebih banyak dan durasi pencarian lebih lama.

2.6 Algoritma Random Restart Hill-climbing

2.6.1 Alur Algoritma Random Restart Hill-climbing

Alur algoritma *Random Restart Hill Climbing* dalam penyelesaian diagonal *magic cube* mirip dengan 2.4.1, namun pencarian akan diulang sampai mendapatkan *diagonal magic cube* ($F_{obj} = 0$):

1. Inisialisasi *initial state* dimana kubus terdiri dari suatu susunan angka 1 hingga 125 secara acak sebagai *current state*. Hitung juga *value* dari *initial state* tersebut dengan fungsi objektif.
2. Bangkitkan semua nilai *successor* dari *current state* dengan menukar posisi 2 angka pada kubus dan hitung juga *value* dari *successornya* dengan fungsi objektif. Pilih *successor* dengan *value* tertinggi sebagai *neighbor*.
3. Terdapat 3 kasus:
 - Jika *neighbor value* \leq *current value* dan *current value* = 0, maka kembalikan *current state*.
 - Jika *neighbor value* \leq *current value* dan *current value* < 0, maka ulangi kembali *hill-climbing* dari awal (step nomor 1).
 - Selain dari itu, ubah *current state* menjadi *neighbor* dan ulangi lagi pencarian pada step nomor 2.

2.6.2 Implementasi Algoritma Random Restart Hill-climbing

Berikut merupakan implementasi algoritma random restart hill climbing dalam bahasa TypeScript.

```
import { LocalSearch } from "./LocalSearch";
import { MagicCube } from "./MagicCube";
import { Plot } from "./Plot";
import { SearchDto } from "./SearchDto";

export interface RandomRestartSearchDto extends SearchDto {
    restartCount: number;
    iterationCounter: number[];
    plots: Plot<number, number>[];
}

export class RandomRestartHC extends LocalSearch {
```

```
private readonly maxRestarts: number;
private restartCount = 0;
private iterationCounter: number[] = [];
private bestObjectiveFunction = -Infinity;
private plots: Plot<number, number>[] = [];
private currentPlotData: { x: number; y: number }[] = [];

constructor(private cube: MagicCube, maxRestarts: number) {
    super(cube);
    this.maxRestarts = maxRestarts;
}

public solve(): void {
    this.startTimer();
    this.plots = []; // Reset plots at the start

    while (this.restartCount < this.maxRestarts) {
        // Initialize new plot data for this restart
        this.currentPlotData = [];

        if (this.performSingleRestart()) {
            break; // Found global optimum
        }

        // After each restart, create a plot entry
        this.plots.push({
            labelX: `Iteration Number, Number Restart ${this.restartCount + 1}`,
            labelY: "Objective Function Value",
            data: [...this.currentPlotData],
        });

        this.restartCount++;
    }

    this.endTimer();
}

private performSingleRestart(): boolean {
```

```
this.cube = new MagicCube();
let iterationNumber = 0;

while (true) {
    const currentValue = this.cube.calculateObjectiveFunction();
    const bestNeighbor = this.cube.getBestSuccessor();
    const neighborValue = bestNeighbor.calculateObjectiveFunction();

    // Record current state for plotting
    this.updateSearchState(iterationNumber, currentValue);

    if (neighborValue <= currentValue) {
        this.handlePeak(currentValue, iterationNumber);
        return currentValue === 0; // Return true if global optimum found
    }

    this.cube = bestNeighbor;
    iterationNumber++;
}
}

private updateSearchState(
    iterationNumber: number,
    currentValue: number
): void {
    this.addStateEntry(this.cube);

    // Add data point for current iteration
    this.currentPlotData.push({
        x: iterationNumber,
        y: currentValue,
    });
}

private handlePeak(currentValue: number, iterationNumber: number): void {
    if (currentValue > this.bestObjectiveFunction) {
        this.bestObjectiveFunction = currentValue;
    }
}
```

```

        this.iterationCounter.push(iterationNumber);
    }

    public toSearchDto(): RandomRestartSearchDto {
        return {
            duration: this.getDuration(),
            finalStateValue: this.bestObjectiveFunction,
            iterationCount: this.calculateTotalIterations(),
            states: this.getStates(),
            restartCount: this.restartCount,
            iterationCounter: this.iterationCounter,
            plots: this.plots, // Return all plots for all restarts
        };
    }

    public getPlots(): Plot<number, number>[] {
        return this.plots;
    }

    private calculateTotalIterations(): number {
        return this.iterationCounter.reduce((sum, count) => sum + count, 0);
    }
}

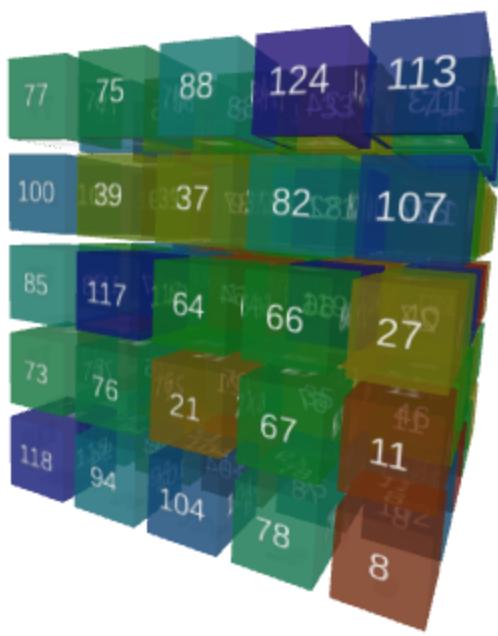
```

2.6.3 Hasil Eksperimen

2.6.3.1 Eksperimen 1

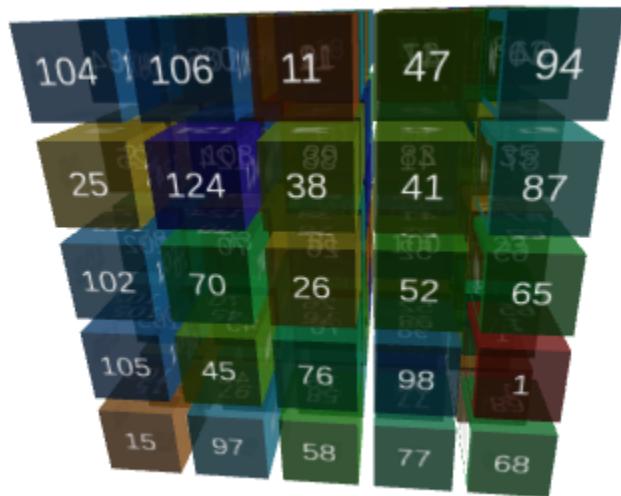
Dengan menggunakan maximum restart params bernilai 2

State awal kubus :



87	36	105	61	93	108	72	51	80	115	9	45	26	43	86	98	48	56	33	71	77	75	88	124	113
10	60	42	34	41	69	97	52	28	30	110	109	116	84	32	125	17	92	121	55	100	39	37	82	107
79	63	112	189	20	122	59	83	23	1	96	16	57	18	111	58	99	13	119	40	85	117	64	66	27
25	24	53	114	65	63	7	103	101	38	47	50	22	14	63	15	62	70	35	46	73	76	21	67	11
106	54	4	31	29	91	123	90	5	68	120	19	81	49	2	74	44	12	95	102	118	94	104	78	8

State akhir kubus :



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

Nilai objective function akhir yang dicapai, durasi pencarian, jumlah restart, dan jumlah iteration tiap restart :

Duration: 19.4s Final Objective Function Value: -71 Random Restart Count: 2

Iteration Count per Restart

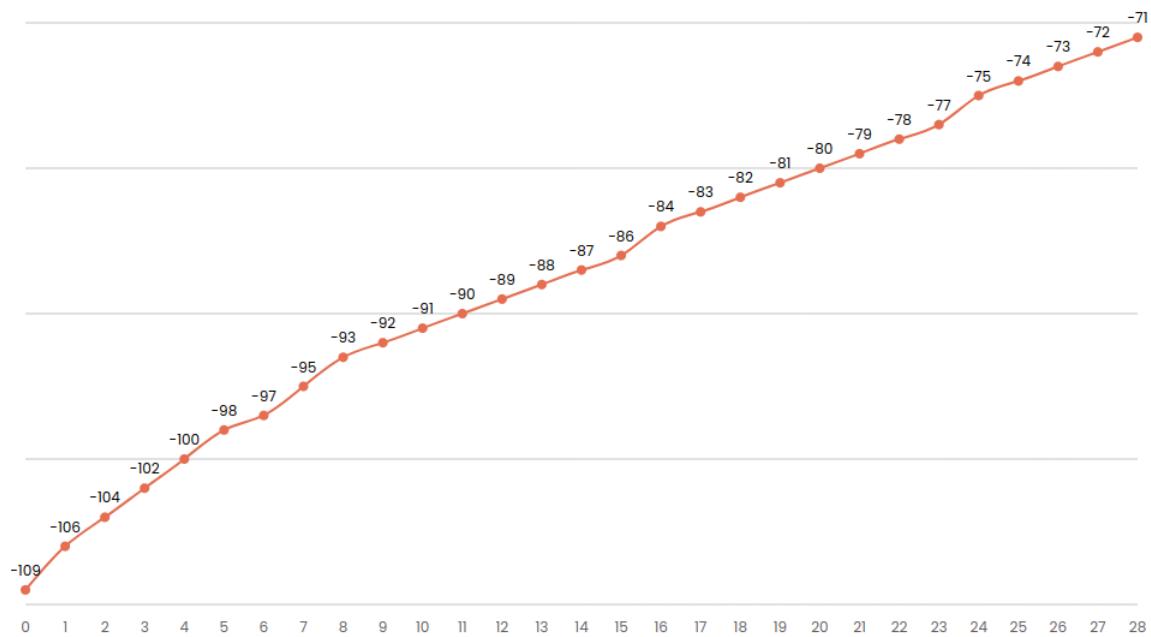
Iteration 1, Count: 28

Iteration 2, Count: 28

Plot nilai objective function dengan jumlah iterasi :

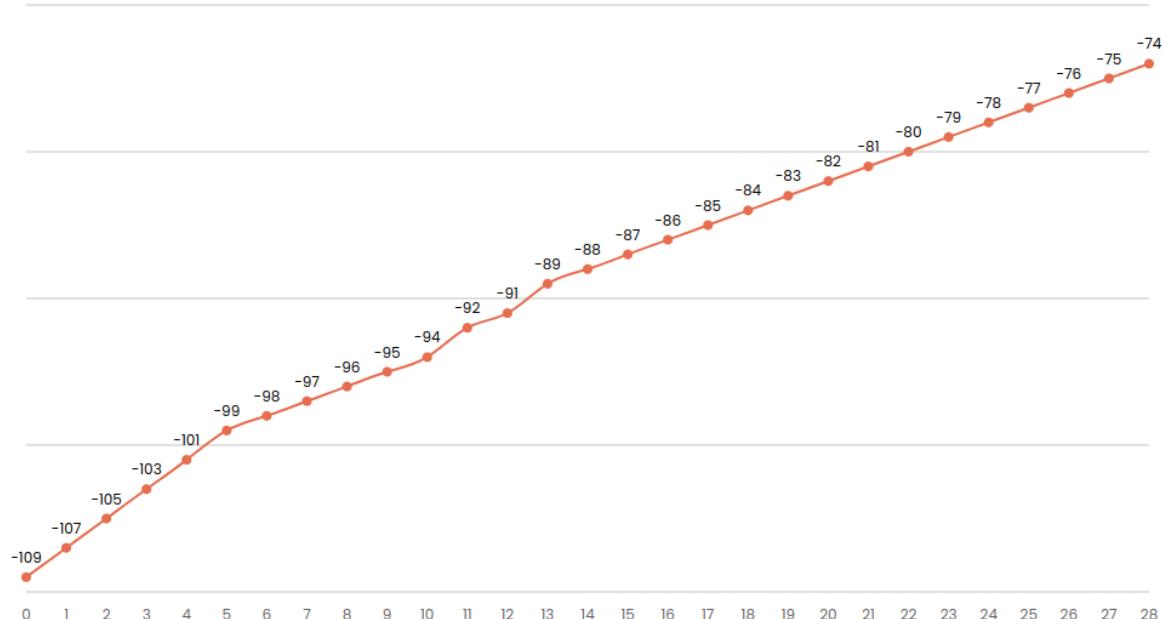
Objective Function Value vs Iteration Number, Number Restart 1

Algorithm progression over time



Objective Function Value vs Iteration Number, Number Restart 2

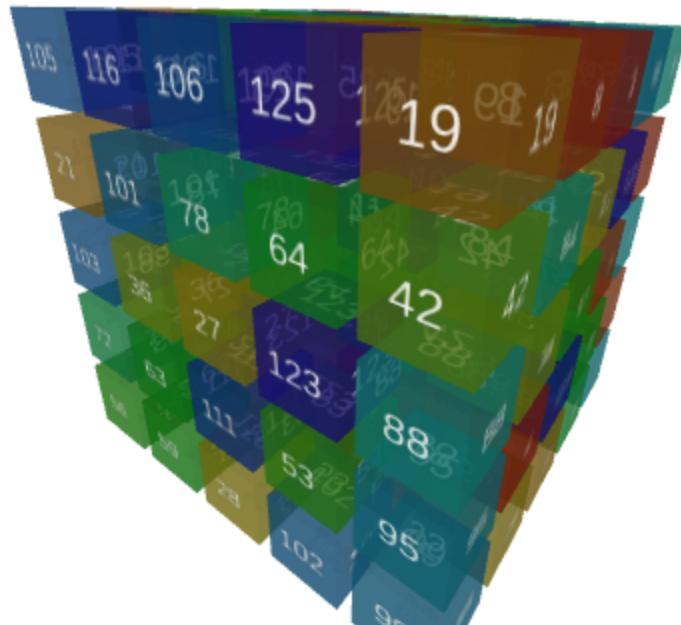
Algorithm progression over time



2.6.3.2 Eksperimen 2

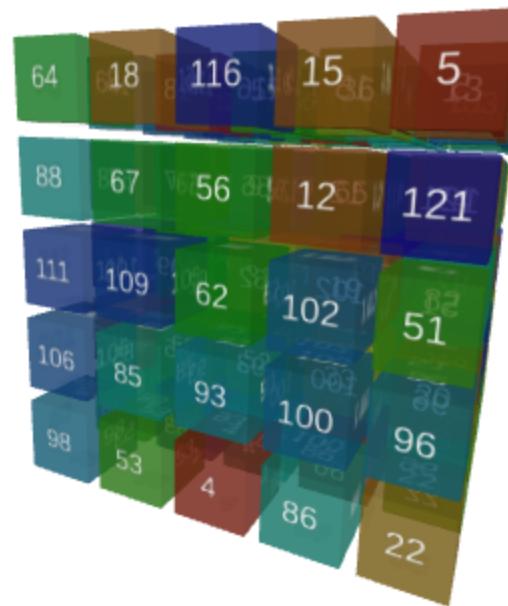
Dengan menggunakan maximum restart params bernilai 2

State awal kubus :



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

State akhir kubus :



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

Nilai objective function akhir yang dicapai, durasi pencarian, jumlah restart, dan jumlah iteration tiap restart :

Duration: 28.6s Final Objective Function Value: -68 Random Restart Count: 3

Iteration Count per Restart

Iteration 1, Count: 33

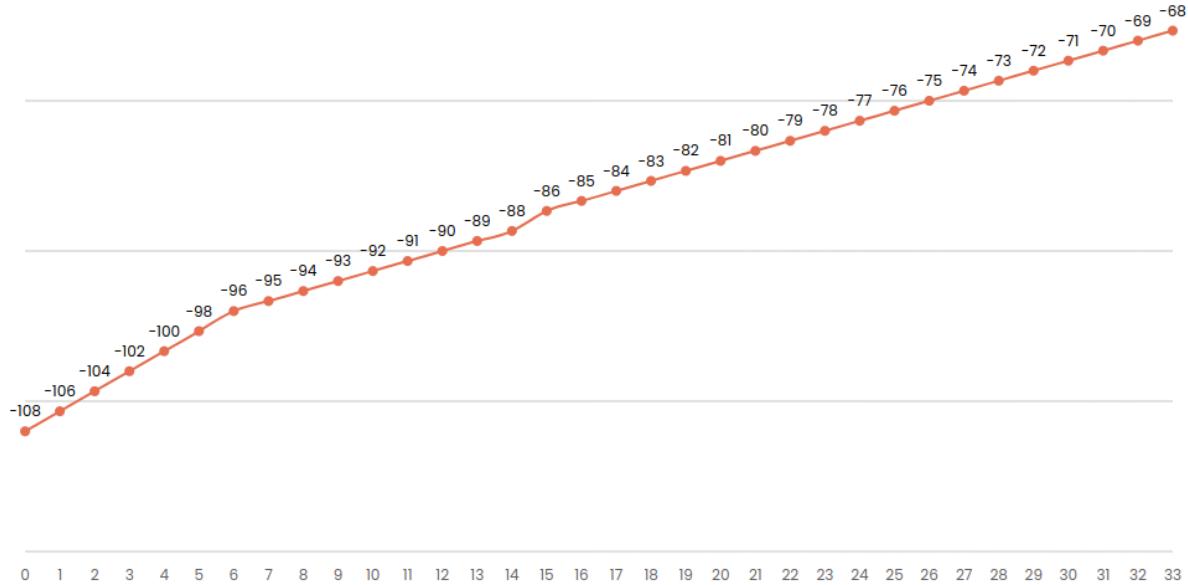
Iteration 2, Count: 32

Iteration 3, Count: 26

Plot nilai objective function dengan jumlah iterasi :

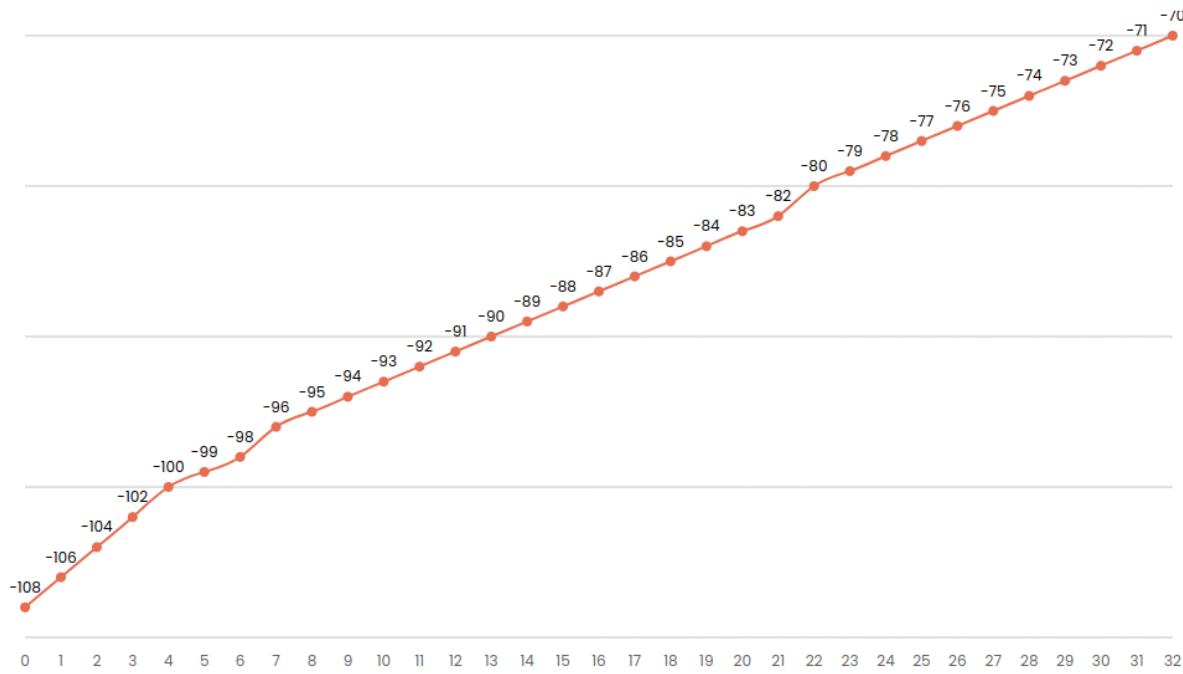
Objective Function Value vs Iteration Number, Number Restart 1

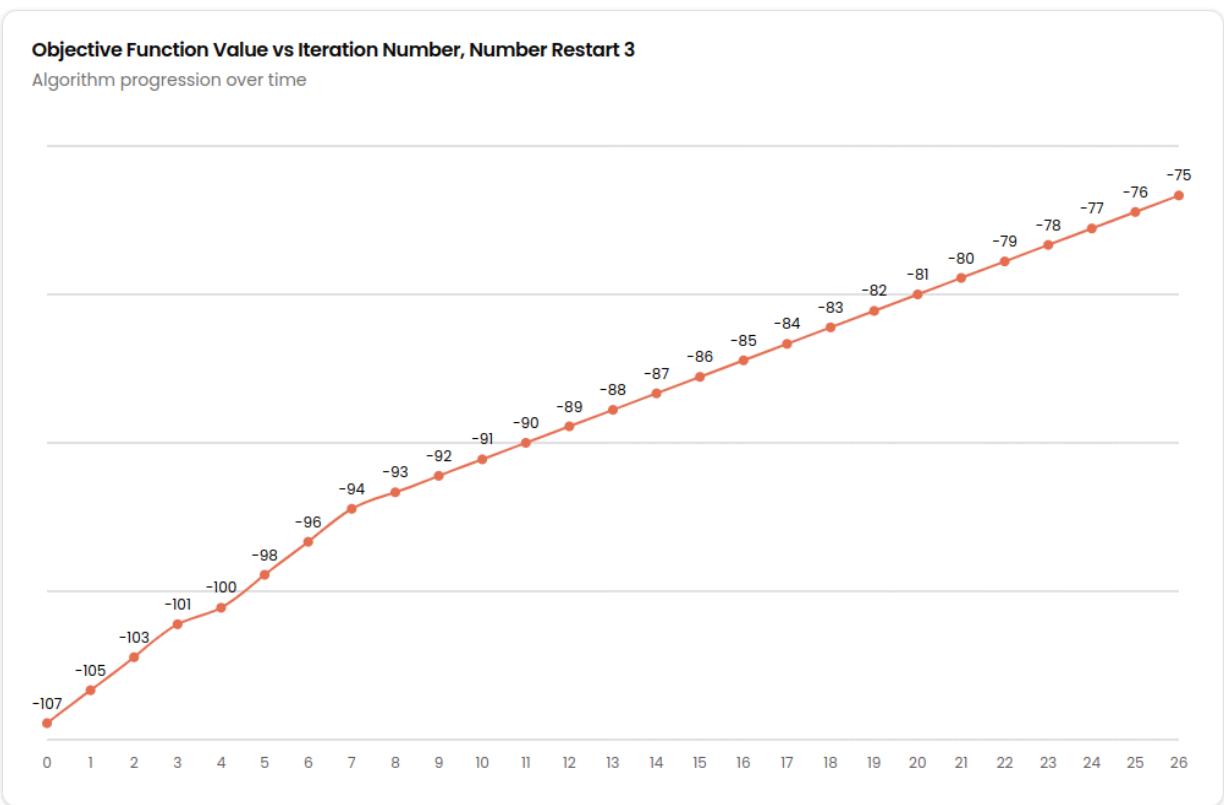
Algorithm progression over time



Objective Function Value vs Iteration Number, Number Restart 2

Algorithm progression over time

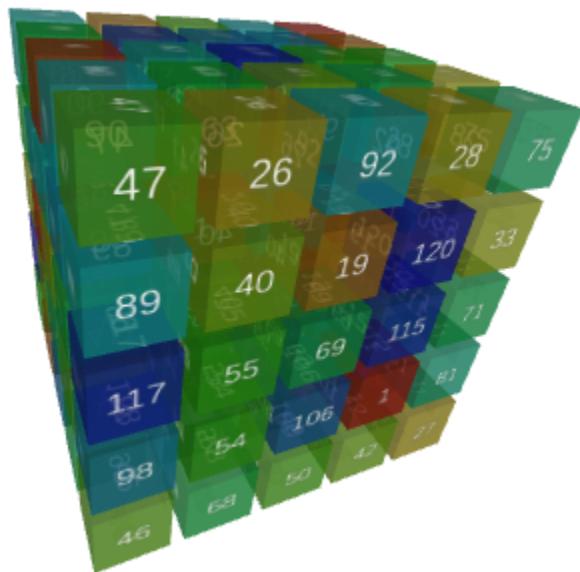




2.6.3.3 Eksperimen 3

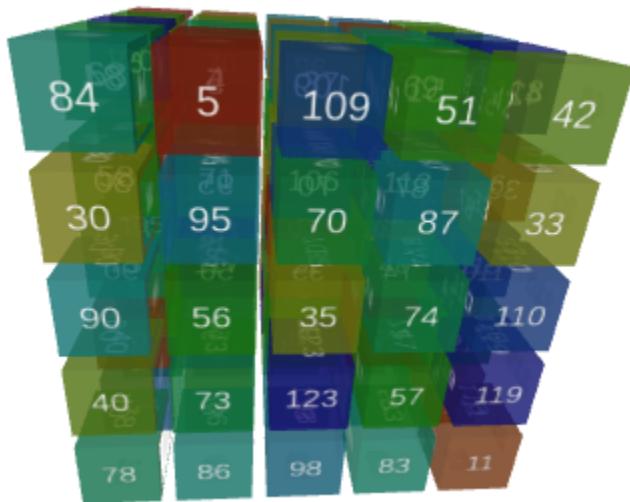
Dengan menggunakan maximum restart params bernilai 5

State awal kubus :



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

State akhir kubus :



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

Nilai objective function akhir yang dicapai, durasi pencarian, jumlah restart, dan jumlah iteration tiap restart :

Duration: 56.95s Final Objective Function Value: -58 Random Restart Count: 5

Iteration Count per Restart

Iteration 1, Count: 25

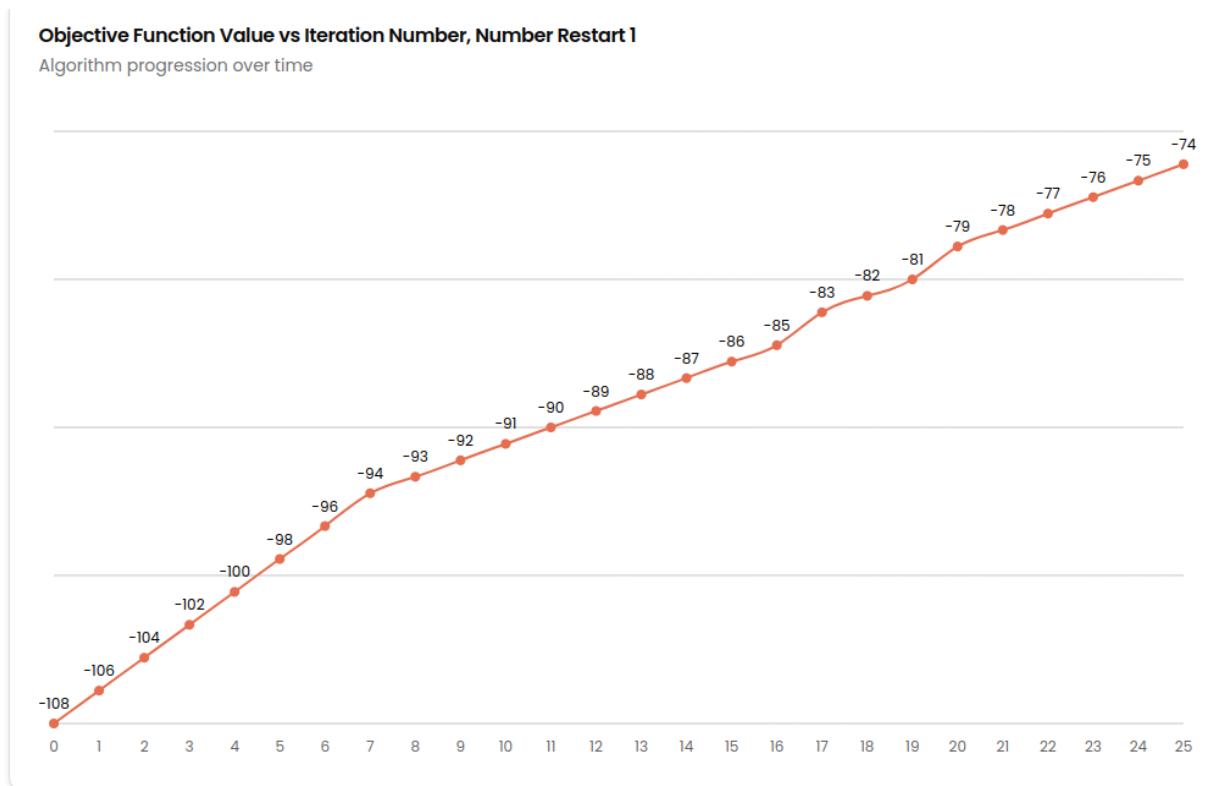
Iteration 2, Count: 34

Iteration 3, Count: 32

Iteration 4, Count: 43

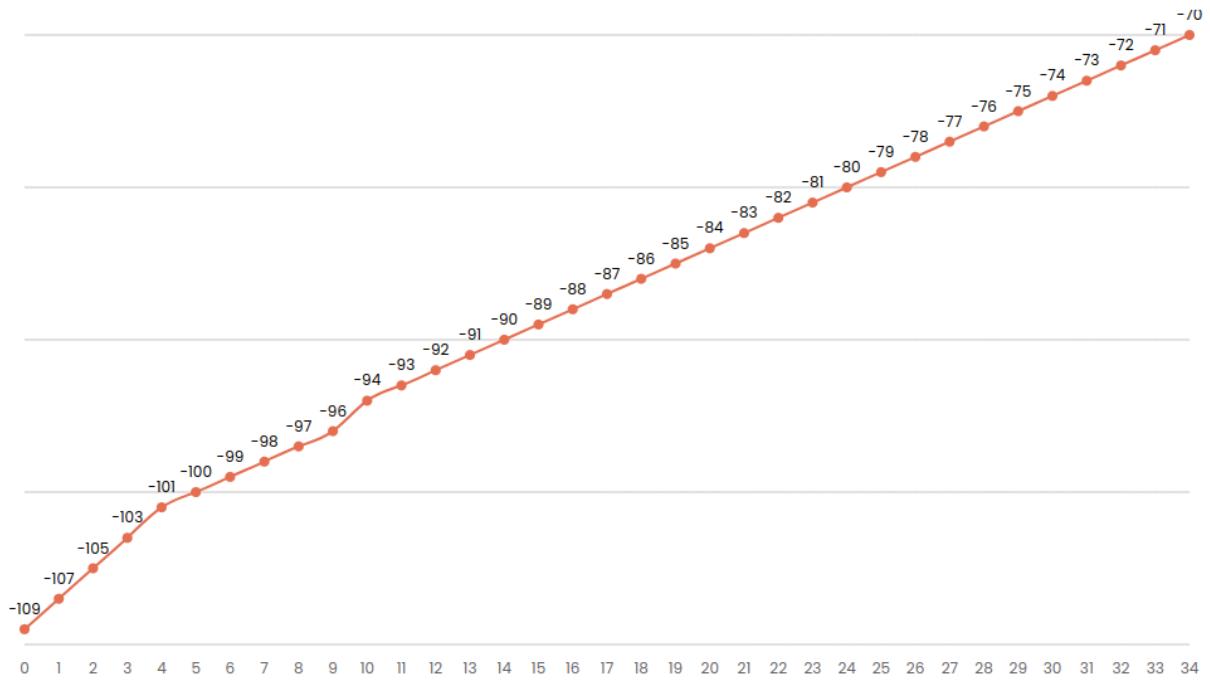
Iteration 5, Count: 32

Plot nilai objective function dengan jumlah iterasi :



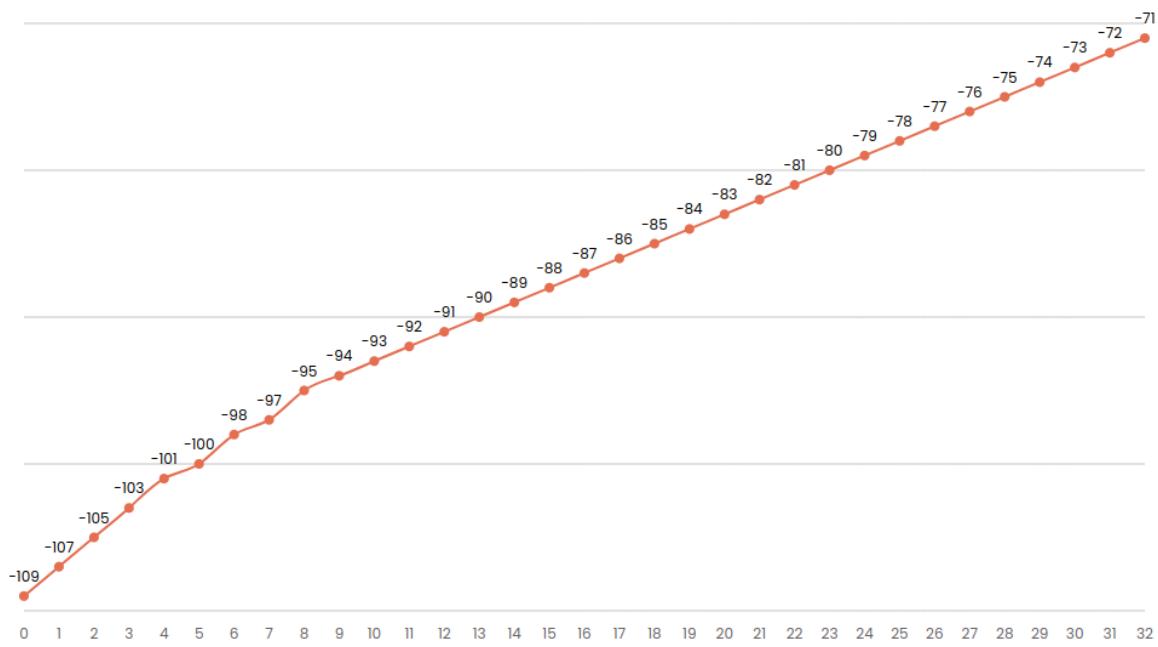
Objective Function Value vs Iteration Number, Number Restart 2

Algorithm progression over time



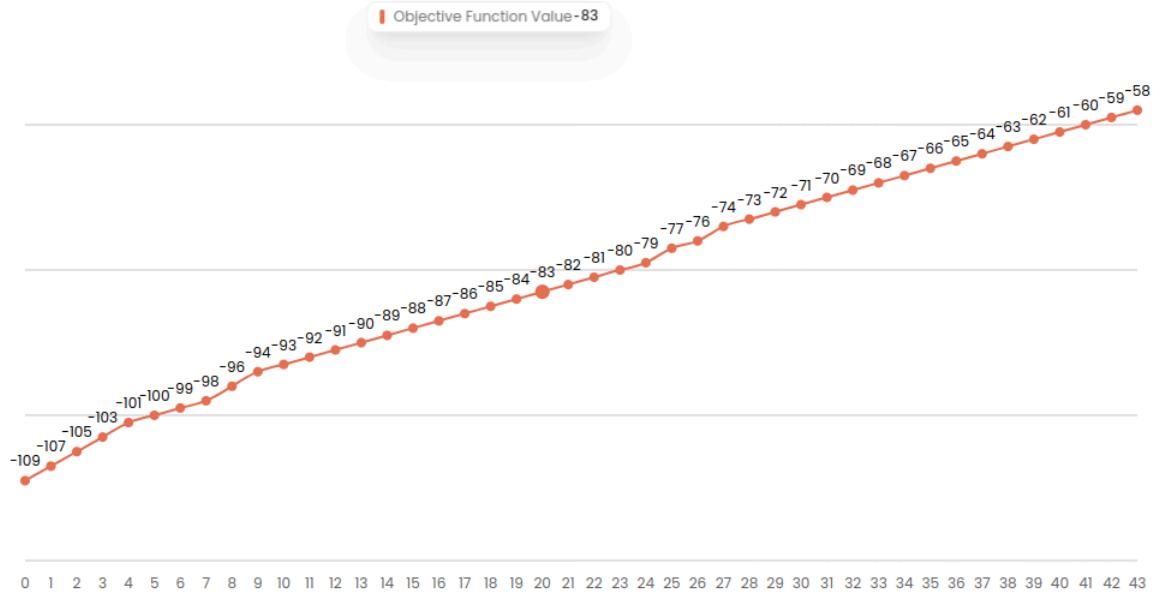
Objective Function Value vs Iteration Number, Number Restart 3

Algorithm progression over time



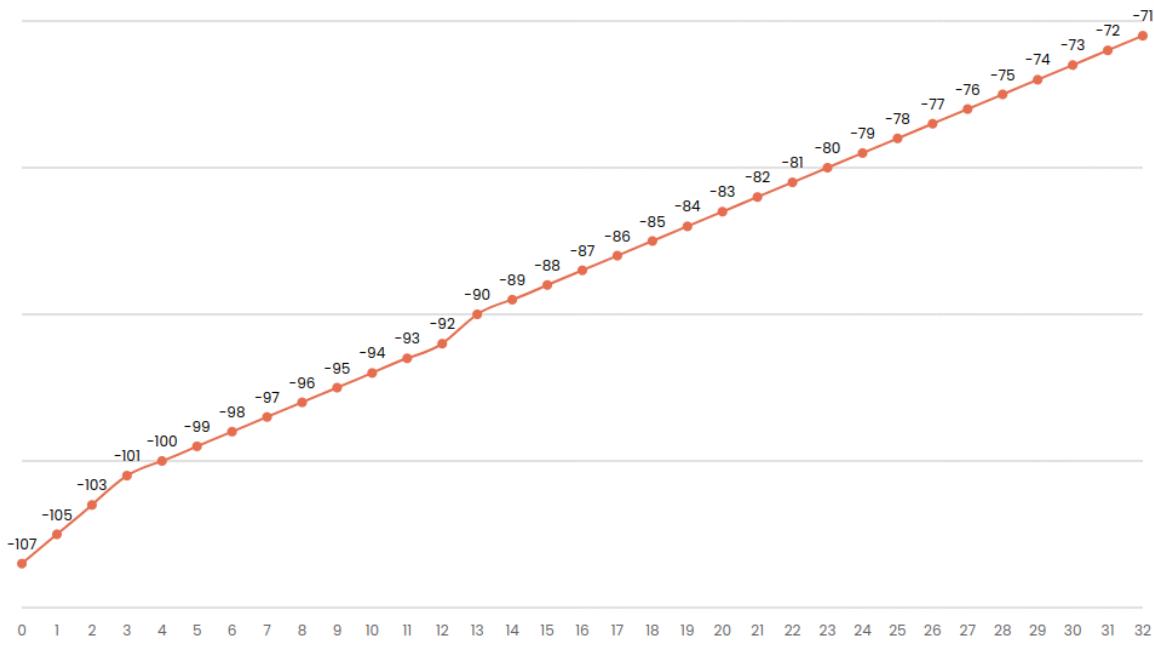
Objective Function Value vs Iteration Number, Number Restart 4

Algorithm progression over time



Objective Function Value vs Iteration Number, Number Restart 5

Algorithm progression over time



2.6.4 Hasil Analisis

Pada algoritma random restart, terdapat parameter yang dapat diatur oleh user yaitu jumlah restart maksimum yang berfungsi membatasi frekuensi pengulangan algoritma hill climbing dari titik awal yang berbeda

Pada eksperimen 1 [bagian 2.6.3.1](#), dengan maksimum restart bernilai 2, didapatkan hasil durasi pencarian selama 19.4 detik, algoritma dapat mencapai nilai objective function akhir sebesar -71 dengan jumlah iterasi pada saat itu adalah 28.

Lalu pada eksperimen 2 [bagian 2.6.3.2](#), dengan maksimum restart bernilai 3, didapatkan durasi pencarian yang lebih lama yaitu 28.6 detik, nilai objective function akhir yang dicapai lebih baik, yaitu -68 dengan jumlah iterasi pada saat itu adalah 33.

Pola serupa terlihat pada [bagian 2.6.3.3](#), di mana dengan durasi pencarian terlama yaitu 56.95 detik dan 5 kali restart, algoritma dapat mencapai nilai objective function akhir terbaik dari eksperimen yang lain, yaitu -58 dengan jumlah iterasi pada saat itu adalah 43.

Hasil analisis ini menunjukkan bahwa algoritma random restart memiliki trade-off antara durasi pencarian dan kualitas solusi yang dihasilkan. Semakin banyak jumlah restart yang dilakukan, semakin lama durasi pencarian yang dibutuhkan, namun memiliki kemungkinan semakin baik pula nilai objective function yang dapat dicapai tergantung jumlah iterasi juga yang dihasilkan pada setiap restartnya, semakin besar jumlah iterasinya maka semakin besar kemungkinan juga nilai *objective function* nya. Hal ini menunjukkan bahwa parameter jumlah maksimum restart menjadi salah satu faktor penting yang dapat disesuaikan oleh pengguna untuk mencapai kualitas solusi yang diharapkan.

2.7 Algoritma Stochastic Hill-climbing

2.7.1 Alur Algoritma Stochastic Hill-climbing

Alur algoritma *Stochastic Hill-climbing* mirip dengan 2.4.1, namun terdapat beberapa perbedaan yang cukup signifikan:

1. Inisialisasi sebuah nilai untuk batas iterasi *nMax*.
2. Inisialisasi *initial state* dimana kubus terdiri dari suatu susunan angka 1 hingga 125 secara acak sebagai *current state*. Hitung juga value dari *initial state* tersebut dengan fungsi objektif.
3. Bangkitkan sebuah nilai *successor* acak dari *current state* dengan menukar posisi 2 angka pada kubus dan hitung juga *value* dari *successornya* dengan fungsi objektif. Jadikan *successor* tersebut sebagai *neighbor*. Jika *neighbor value* > *current value*, maka ubah *current* menjadi *neighbor*. Ulangi langkah ini sebanyak *nMax* kali.
4. Kembalikan nilai *current state*.

2.7.2 Implementasi Algoritma Stochastic Hill-climbing

```
export class Stochastic extends LocalSearch {
    private cube: MagicCube;

    constructor(cube: MagicCube) {
        super(cube);
        this.cube = cube;
    }

    public solve(nmax: number = 1000): void {
        this.startTimer();
        let currentState = this.cube;

        for (let i = 0; i < nmax; i++) {
            const nextState = currentState.generateRandomSuccessor();
            const currentObj = currentState.calculateObjectiveFunction();
            const nextObj = nextState.calculateObjectiveFunction();

            const deltaE = nextObj - currentObj;

            this.addStateEntry(currentState);
            this.addObjectiveFunctionPlotEntry(this.iterationCount, currentObj);

            this.addIterationCount();
            if (deltaE > 0) {
                currentState = nextState;
            }
        }

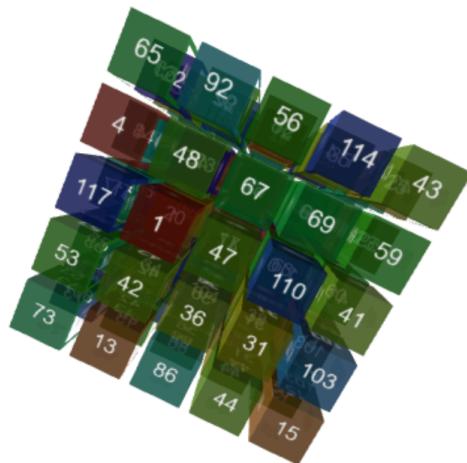
        this.endTimer();
    }
}
```

2.7.3 Hasil Eksperimen

2.7.3.1 Eksperimen 1

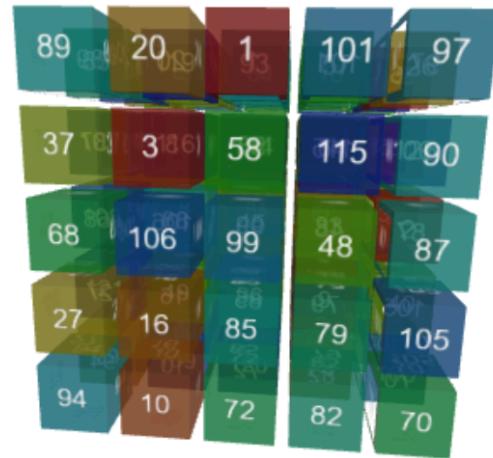
Nmax : 5000

State awal :



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

State akhir :



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

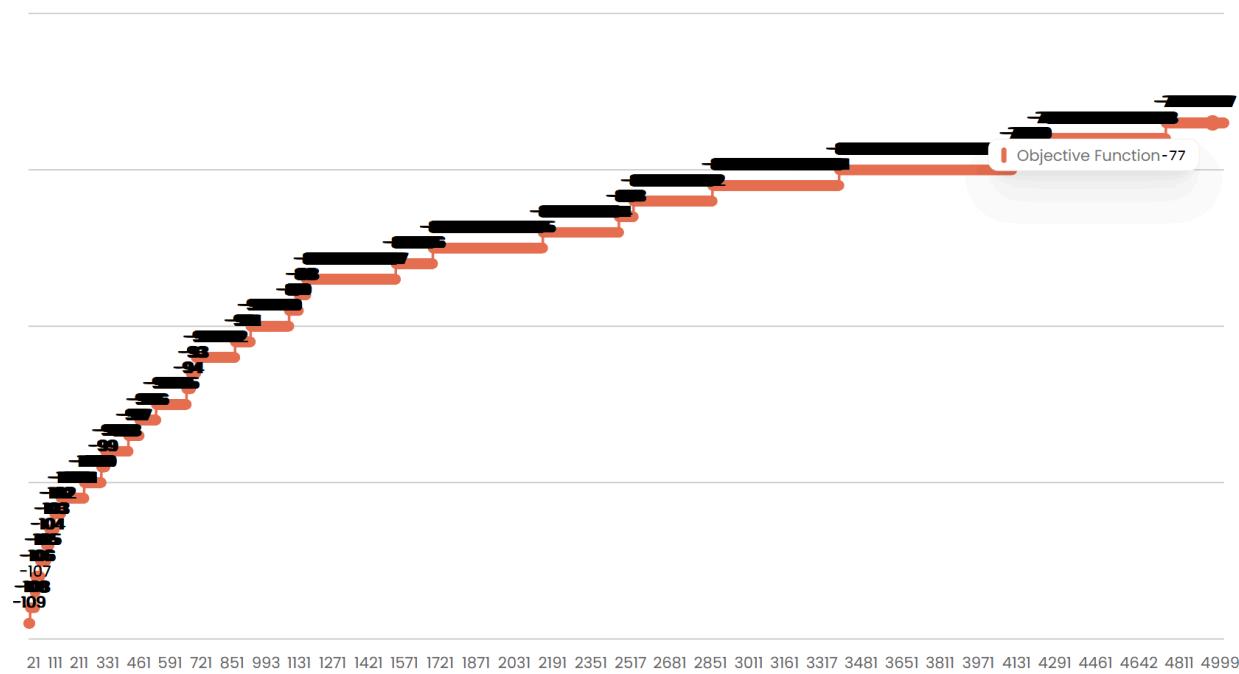
Nilai objective function akhir yang dicapai, durasi pencarian, dan jumlah iterasi :

Duration: 0.83s Final Objective Function Value: -77 Iteration Count: 5000

Plot :

Objective Function vs Iteration

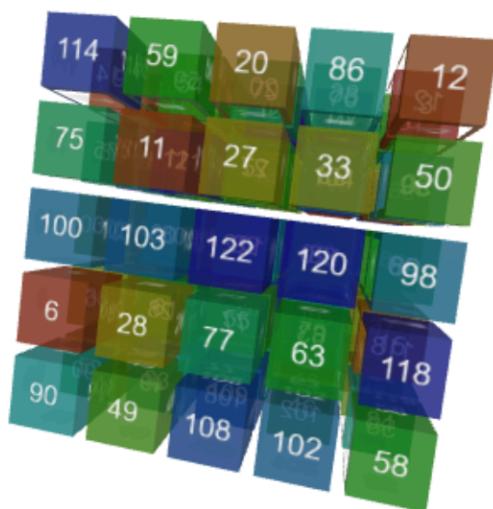
Algorithm progression over time



2.7.3.2 Eksperimen 2

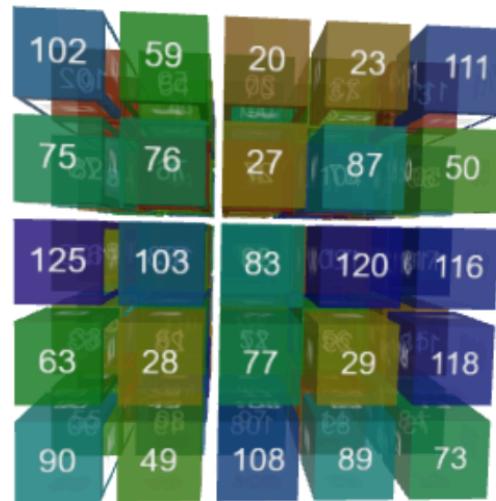
Nmax : 20000

State awal :



94	97	32	71	72	91	24	116	17	78	64	41	125	117	51	9	54	70	76	3	114	59	20	86	12
60	106	130	7	111	87	62	69	1	2	115	44	34	110	35	82	121	57	104	96	75	11	27	33	50
40	74	31	4	113	101	56	123	14	81	13	29	119	39	53	36	80	92	83	65	100	103	122	120	98
8	95	73	66	88	99	21	45	67	25	23	107	112	105	37	38	61	55	79	15	6	28	77	63	118
19	47	48	22	26	68	124	43	16	5	10	89	85	18	42	52	93	109	46	84	90	49	108	102	58

State akhir :



94	46	32	71	72	34	115	93	2	78	5	41	100	65	51	9	54	70	37	3	102	59	20	23	111
60	106	130	107	12	80	62	11	121	122	18	44	39	7	35	82	1	4	104	96	75	76	27	87	50
40	74	31	57	113	101	79	123	109	81	13	6	119	91	33	36	53	92	17	117	125	103	83	120	116
8	95	66	58	88	99	21	45	67	25	86	110	112	105	69	38	61	52	56	15	63	28	77	29	118
19	24	48	22	26	68	124	43	16	64	10	114	85	47	42	55	98	97	14	84	90	49	108	89	73

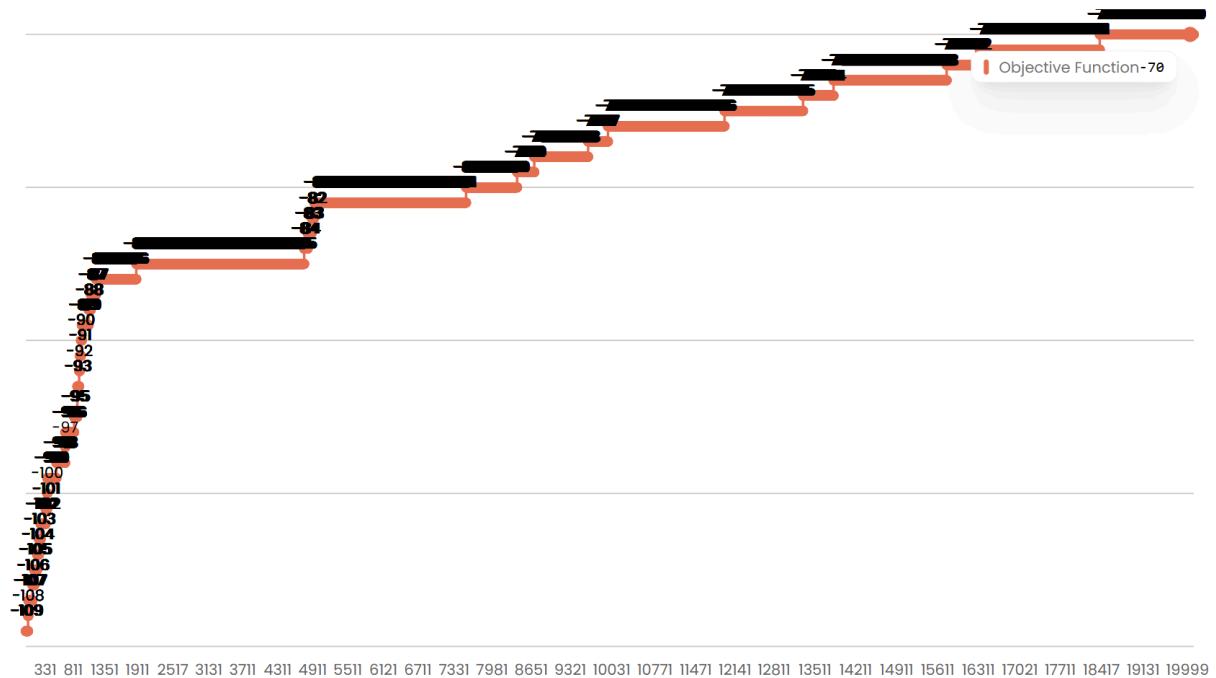
Nilai objective function akhir yang dicapai, durasi pencarian, dan jumlah iterasi :

S

Plot :

Objective Function vs Iteration

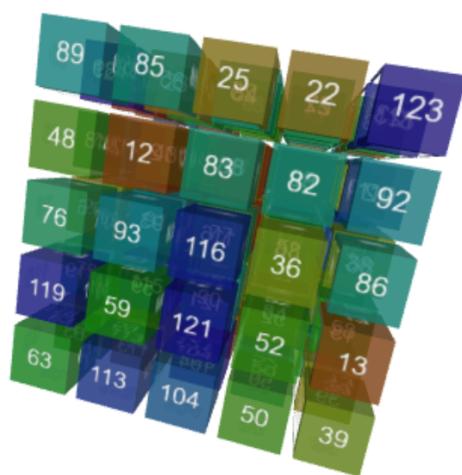
Algorithm progression over time



2.7.3.3 Eksperimen 3

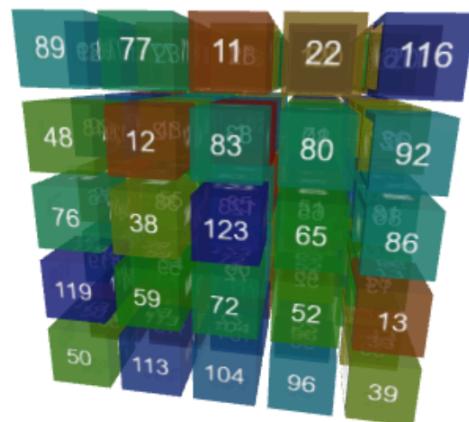
Nmax : 50000

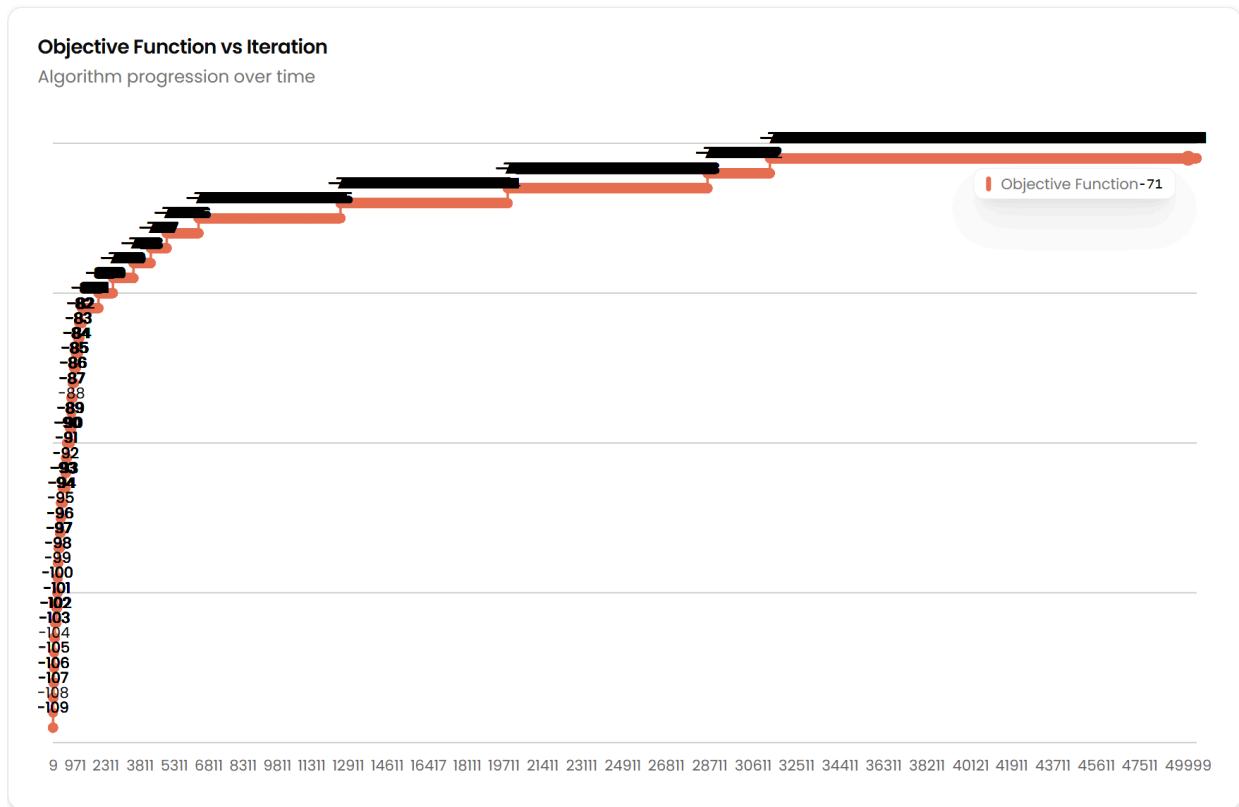
State awal :



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

State akhir :





2.7.4 Hasil Analisis

Pada algoritma stochastic, terdapat parameter yang harus diinput oleh user yaitu jumlah iterasi. Parameter ini menentukan seberapa banyak *random successor* yang akan dibangkitkan dalam proses pencarian solusi. Dari hasil eksperimen dengan tiga nilai iterasi yang berbeda yaitu 5000, 20000, dan 50000 iterasi, dapat dilihat bahwa jumlah iterasi memiliki pengaruh signifikan terhadap durasi eksekusi dan nilai fungsi objektif yang dihasilkan. Dengan 5000 iterasi, algoritma berjalan sangat cepat dalam waktu 0.83 detik namun menghasilkan nilai fungsi objektif -77. Ketika iterasi ditingkatkan menjadi 20000, waktu eksekusi meningkat menjadi 2.57 detik dengan nilai fungsi objektif -70. Pada eksperimen ketiga dengan 50000 iterasi, durasi eksekusi menjadi yang terpanjang yaitu 5.46 detik dan menghasilkan nilai fungsi objektif -71. Hal ini menunjukkan adanya trade-off antara waktu komputasi dan kualitas solusi, dimana semakin banyak iterasi akan memberikan kesempatan lebih besar untuk menemukan solusi yang lebih baik namun membutuhkan waktu komputasi yang lebih lama. Oleh karena itu, user perlu mempertimbangkan dalam menentukan jumlah iterasi yang diinginkan.

2.8 Algoritma Simulated Annealing

2.8.1 Alur Algoritma Simulated Annealing

Alur algoritma *Simulated Annealing* mirip dengan algoritma *Stochastic Hill-climbing* (*successor* dipilih secara random sebagai *neighbor*), namun *downhill moves* diperbolehkan dengan aturan probabilitas tertentu. Misalkan terdapat sebuah fungsi $T(t)$ yang memetakan waktu ke temperatur dimana $T(t) \geq 0$ dan ada suatu nilai t sehingga $T(t) = 0$. Berikut merupakan algoritma *simulated annealing*:

1. Inisialisasi *initial state* dimana kubus terdiri dari suatu susunan angka 1 hingga 125 secara acak sebagai *current state*. Hitung juga value dari *initial state* tersebut dengan fungsi objektif.
2. Inisialisasikan nilai waktu t dimulai dari 1.
3. Hitung nilai temperatur pada waktu sekarang sebagai variable T .
4. Terdapat 2 kasus:
 - Jika temperatur sekarang $T = 0$, maka langsung kembalikan *current state*.
 - Jika tidak,
 - a. Bangkitkan sebuah nilai *successor* acak dari *current state* dengan menukar posisi 2 angka pada kubus dan hitung juga *value* dari *successornya* dengan fungsi objektif. Jadikan *successor* tersebut sebagai *next*. Hitung selisih *next value* dengan *current value* sebagai ΔE .
 - b. Terdapat 2 kasus:
 - Jika $\Delta E > 0$, maka ubah *current* menjadi *next* kemudian ulangi lagi pada step nomor 3,
 - Selain itu, hanya ubah *current* menjadi *next* **hanya jika** $P(\text{move}) = e^{\Delta E/T} > L$. Dimana L bisa ditetapkan sebagai nilai statis (misalnya 0.5) atau digenerate dinamis (misalnya RAND(0,1)). Kemudian ulangi lagi pada step nomor 3.

2.8.2 Implementasi Algoritma Simulated Annealing

Berikut merupakan implementasi algoritma simulated annealing dalam bahasa TypeScript.

```
import { LocalSearch } from "./LocalSearch";
import { MagicCube } from "./MagicCube";
import { Plot } from "./Plot";

export class SimulatedAnnealing extends LocalSearch {
    // Simulated annealing specific attributes
```

```
private minimumTemperature: number;
private temperatureFunction;
private staticProbabilityValue: number | null;

// Current state
private currentState: MagicCube;

// Plot result
private probabilityPlot: Plot<number, number>;

// Stuck local optima counter
private stuckLocalOptimaCounter: number;

// Constructor
constructor(
    initialCube: MagicCube,
    temperatureFunction: (t: number) => number,
    minimumTemperature: number = 0,
    staticProbabilityValue: number | null = null
) {
    super(initialCube);
    this.temperatureFunction = temperatureFunction;
    this.minimumTemperature = minimumTemperature;
    this.staticProbabilityValue = staticProbabilityValue;
    this.currentState = initialCube;
    this.probabilityPlot = {
        labelX: "Iterasi",
        labelY: "e^(dE/T)",
        data: [],
    };
    this.stuckLocalOptimaCounter = 0;
}

// Solver
public solve() {
    this.startTimer();

    let t = 1;
```

```
while (true) {
    let temperature = this.temperatureFunction(t);

    // Stop
    if (temperature < this.minimumTemperature) break;

    const currentStateValue = this.currentState.calculateObjectiveFunction();
    const nextState = this.currentState.generateRandomSuccessor();
    const nextStateValue = nextState.calculateObjectiveFunction();

    // Update plot data + history states
    this.addStateEntry(this.currentState);
    this.addIterationCount();
    this.addObjectiveFunctionPlotEntry(
        this.iterationCount,
        currentStateValue
    );

    const deltaE = nextStateValue - currentStateValue;
    if (deltaE > 0) {
        this.currentState = nextState;
    } else {
        const probability = this.staticProbabilityValue ?? Math.random();
        const edET = Math.exp(deltaE / temperature);
        if (edET > probability) {
            // Do some bad moves
            this.currentState = nextState;

            // Local optima is visited when doing some bad moves
            if (deltaE < 0) {
                console.log(this.iterationCount, currentStateValue, nextStateValue);
                this.addStuckLocalOptimaCounter();
            }
        }
    }

    // Update probability plot
    this.addProbabilityPlotEntry(this.getIterationCount(), edET);
}
```

```
// Update search state
t++;
temperature = this.temperatureFunction(t);
}

this.endTimer();
}

// Get plot data
public getProbabilityPlot(): Plot<number, number> {
    return this.probabilityPlot;
}

// Get aggregated plot data
public getAggregatedProbabilityPlot(n: number): Plot<number, number> {
    const aggregatedData = this.aggregatePlotData(this.probabilityPlot.data, n);

    return {
        labelX: this.probabilityPlot.labelX,
        labelY: this.probabilityPlot.labelY,
        data: aggregatedData,
    };
}

// Get stuck local optima counter
public getStuckLocalOptimaCounter(): number {
    return this.stuckLocalOptimaCounter;
}

// Add plot entry
private addProbabilityPlotEntry(x: number, y: number) {
    this.probabilityPlot.data.push({ x, y });
}

// Add local optima counter
private addStuckLocalOptimaCounter() {
    this.stuckLocalOptimaCounter++;
}
}
```

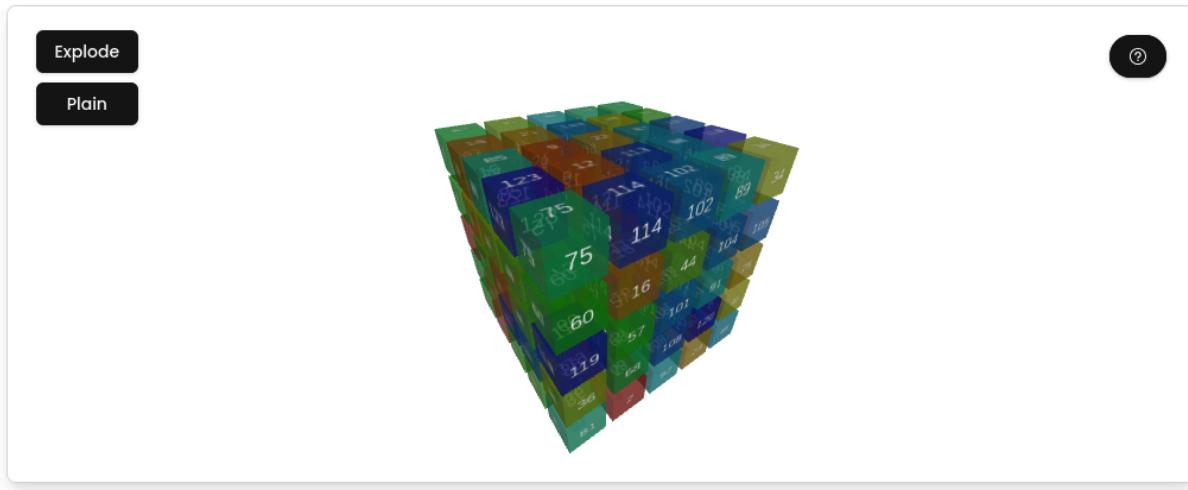
```
export class TemperatureFactory {  
    // T(t) = T0 * alpha^t  
    static exponentialDecay(initialTemperature: number, alpha: number) {  
        return (t: number) => initialTemperature * Math.pow(alpha, t);  
    }  
  
    static fastExponentialDecay(initialTemperature: number, alpha: number) {  
        return (t: number) => initialTemperature * Math.pow(alpha, t * t);  
    }  
  
    // T(t) = T0 * (1 - t/tmax)  
    static linearDecay(initialTemperature: number, tMax: number) {  
        return (t: number) => initialTemperature * (1 - t / tMax);  
    }  
  
    // T(t) = T0/log(1 + t)  
    static logarithmicDecay(initialTemperature: number) {  
        return (t: number) => initialTemperature / Math.log(1 + t);  
    }  
}
```

2.8.3 Hasil Eksperimen

2.8.3.1 Eksperimen 1

Initial state cube:

Duration: 70.51s Final Objective Function Value: -45 Iteration Count: 1000002 Stuck Frequency: 509

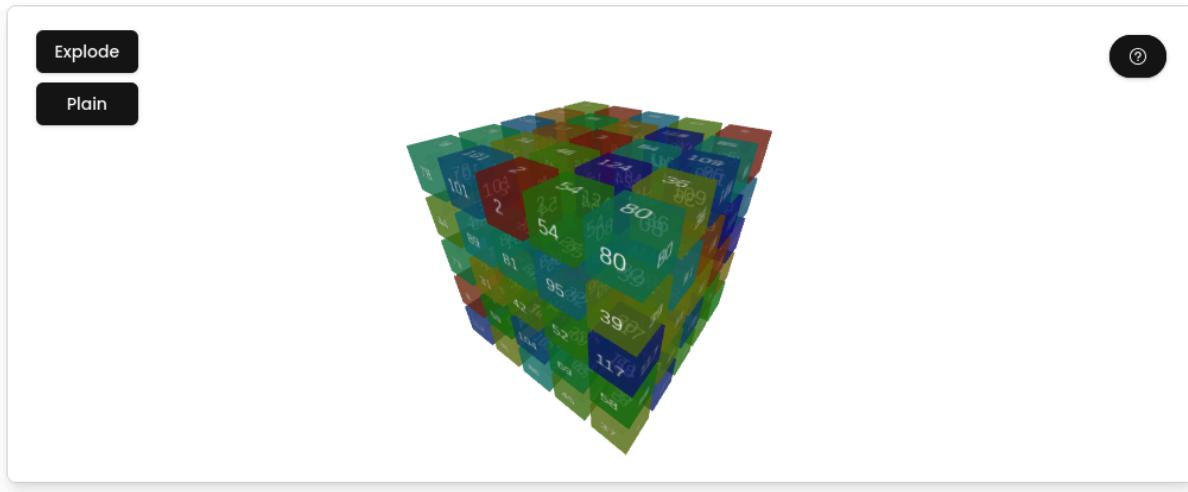


Duration: 70.51s Final Objective Function Value: -45 Iteration Count: 1000002 Stuck Frequency: 509



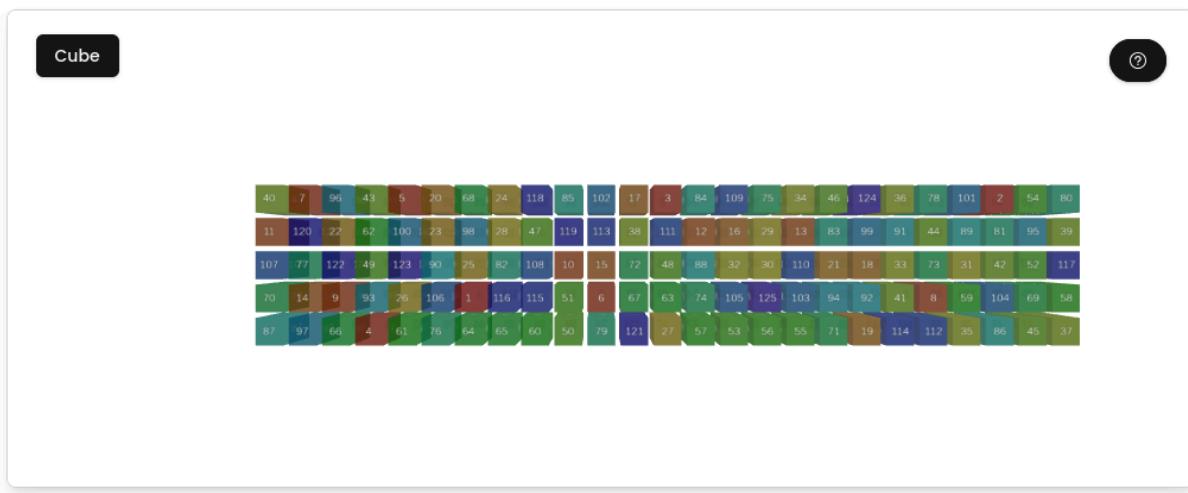
Final state cube:

Duration: 70.51s Final Objective Function Value: -45 Iteration Count: 1000002 Stuck Frequency: 509



Step: 1000002 / 1000002

Duration: 70.51s Final Objective Function Value: -45 Iteration Count: 1000002 Stuck Frequency: 509

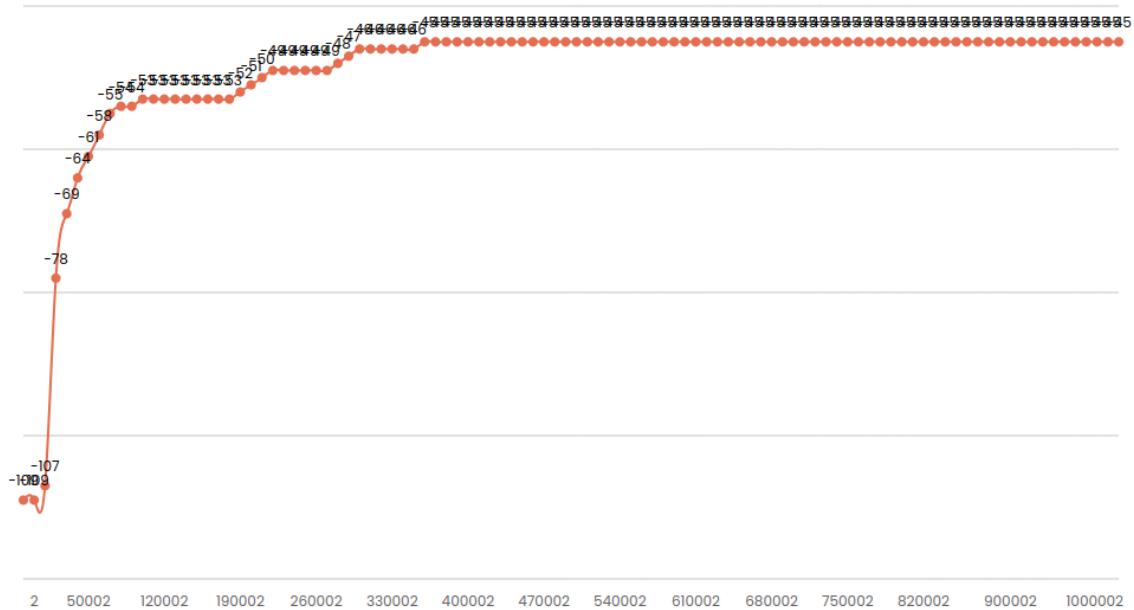


Step: 1000002 / 1000002

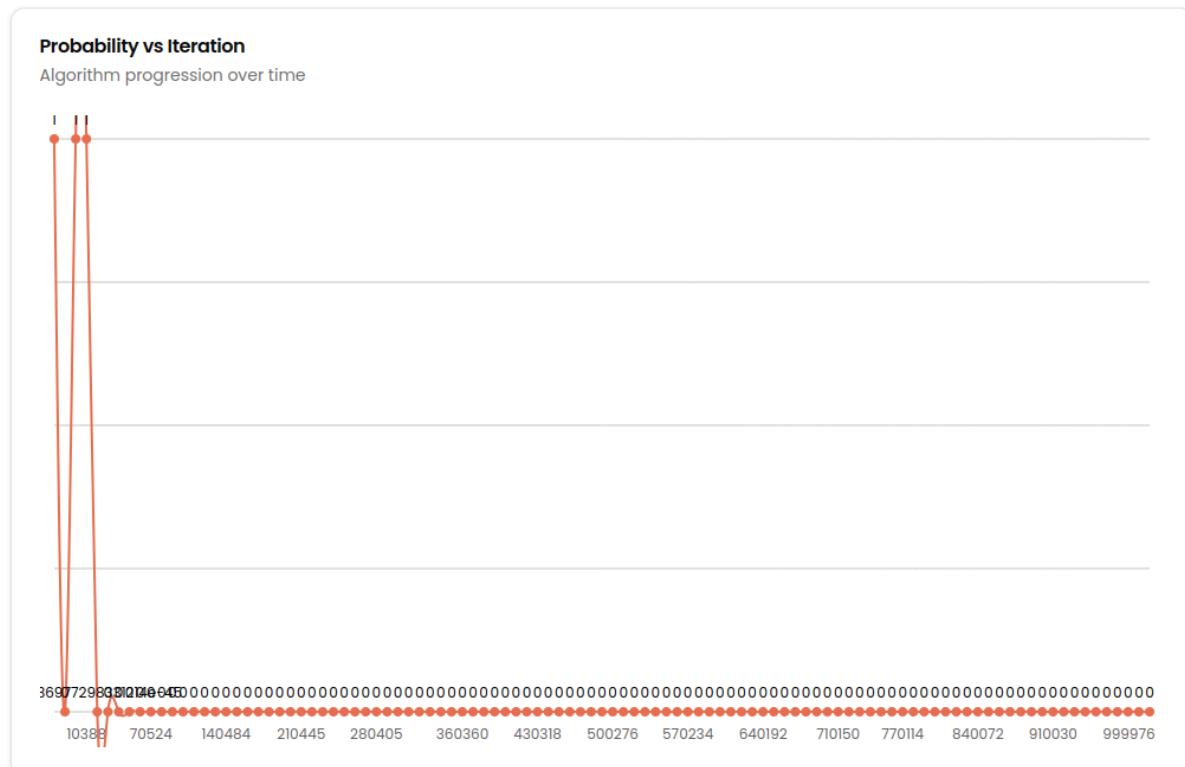
Objective Function vs Iteration Chart:

Objective Function vs Iteration

Algorithm progression over time



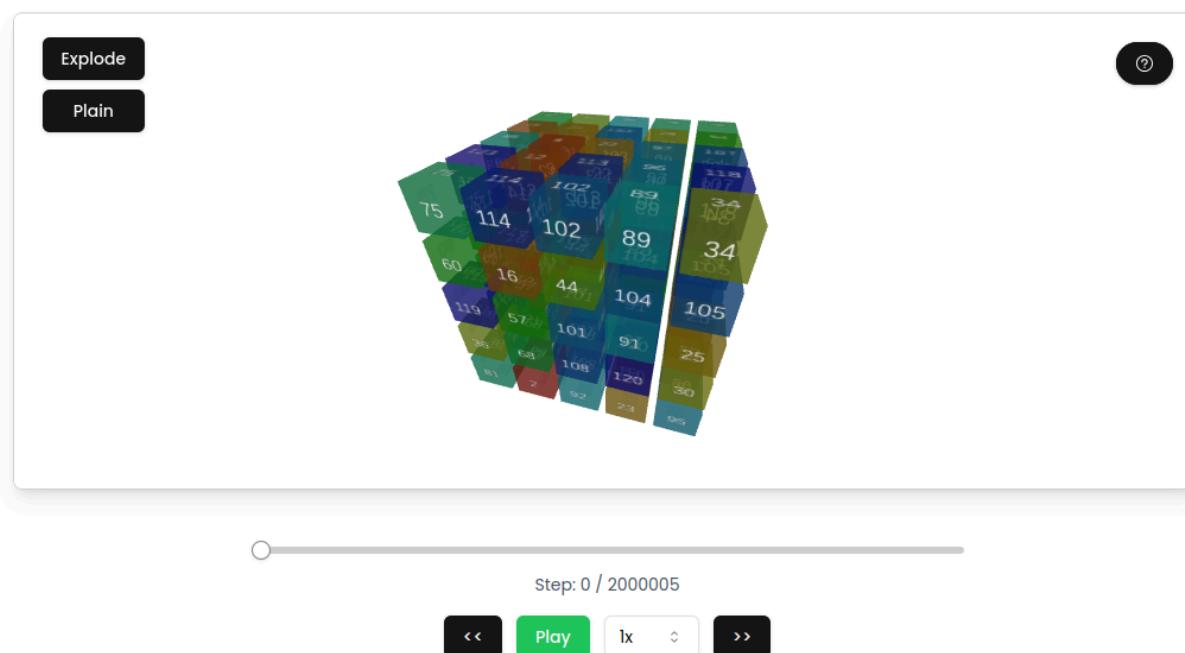
Probability vs Iteration Chart:



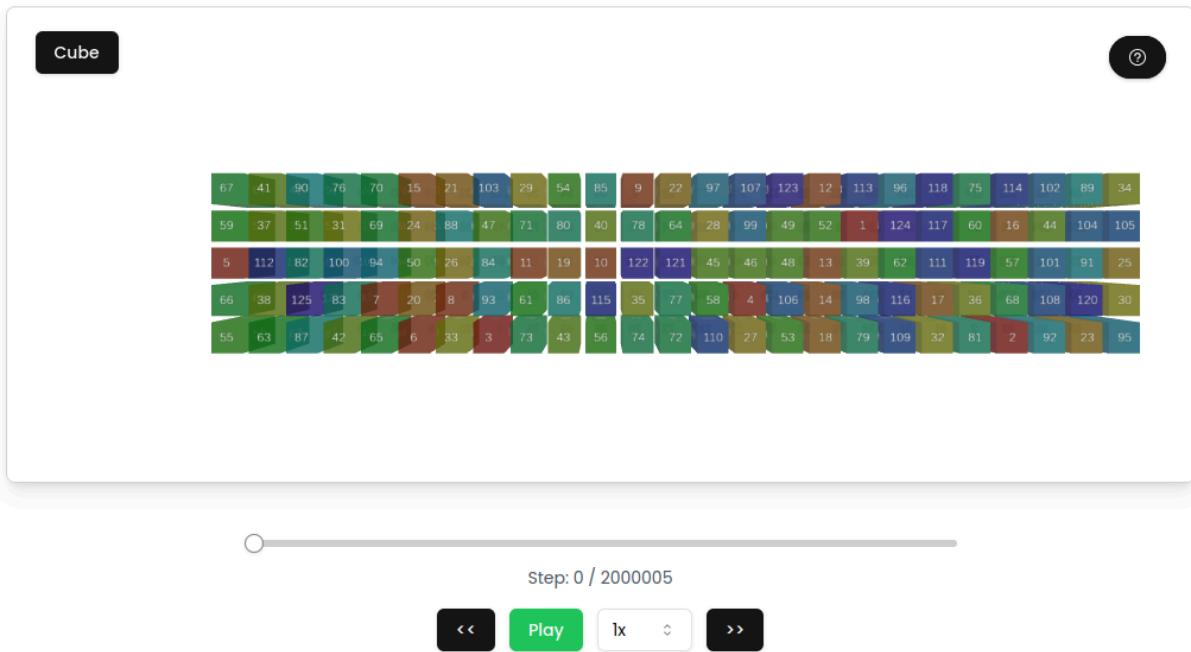
2.8.3.2 Eksperimen 2

Initial state cube:

Duration: 61.58s Final Objective Function Value: -48 Iteration Count: 1000002 Stuck Frequency: 534

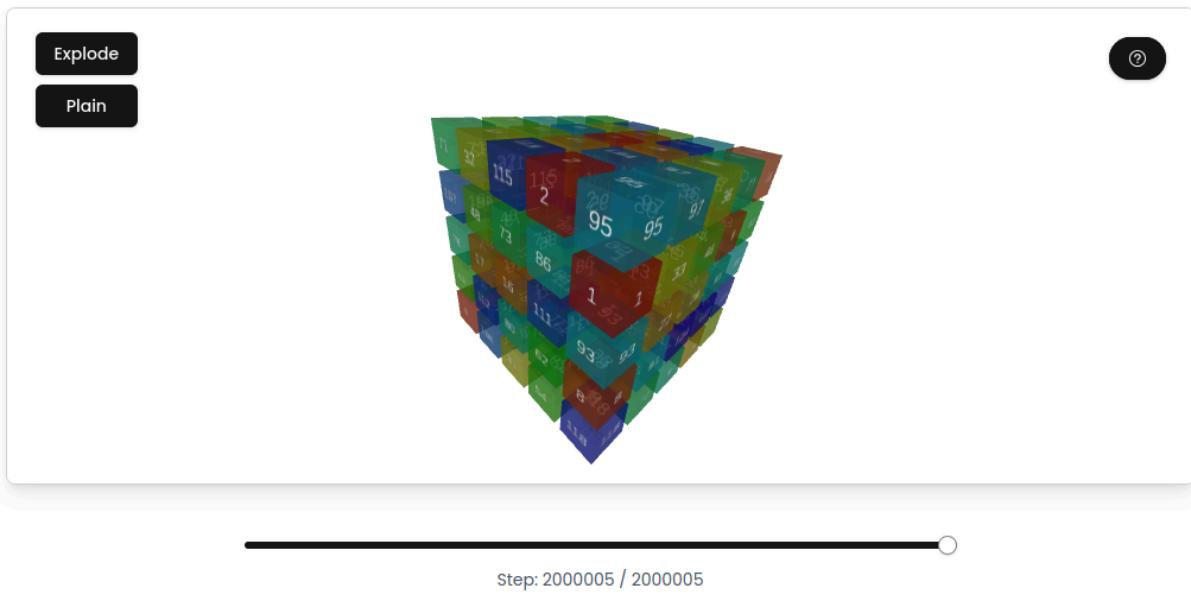


Duration: 61.58s Final Objective Function Value: -48 Iteration Count: 1000002 Stuck Frequency: 534

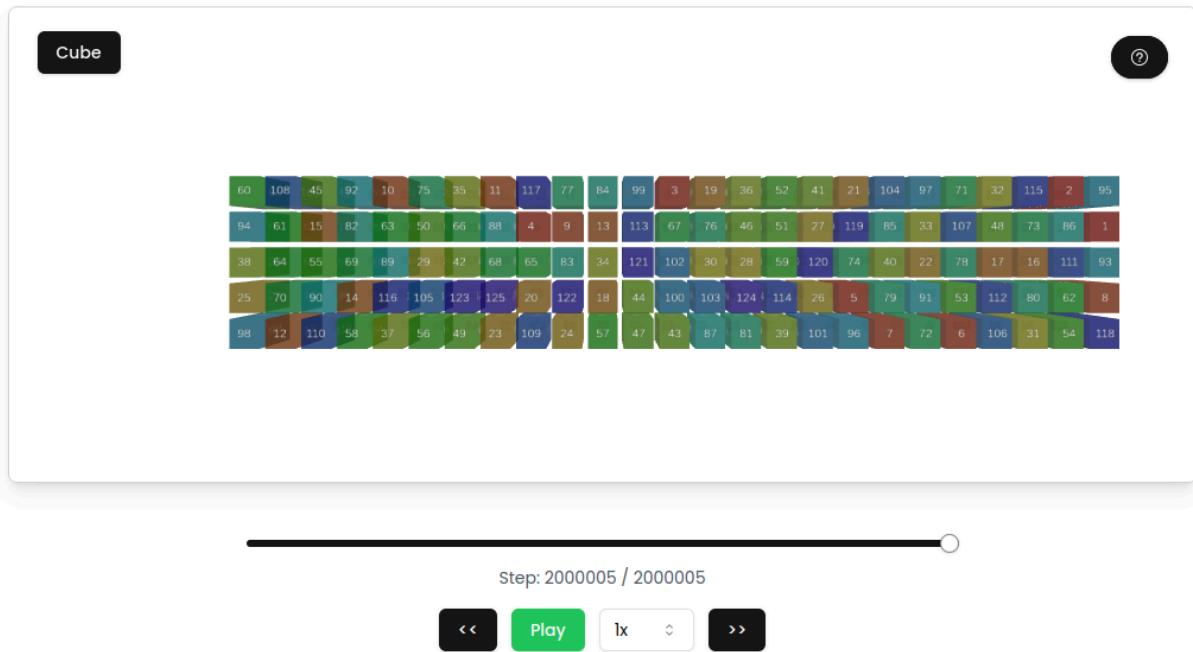


Final state cube:

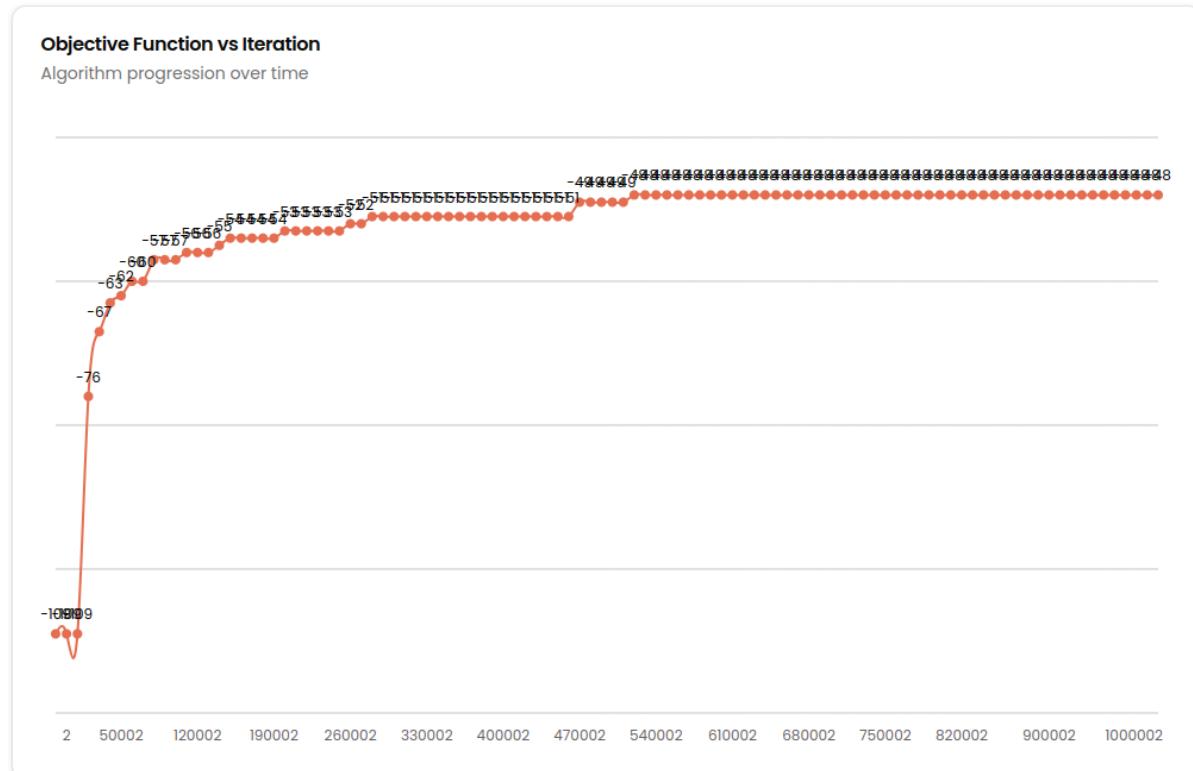
Duration: 61.58s Final Objective Function Value: -48 Iteration Count: 1000002 Stuck Frequency: 534



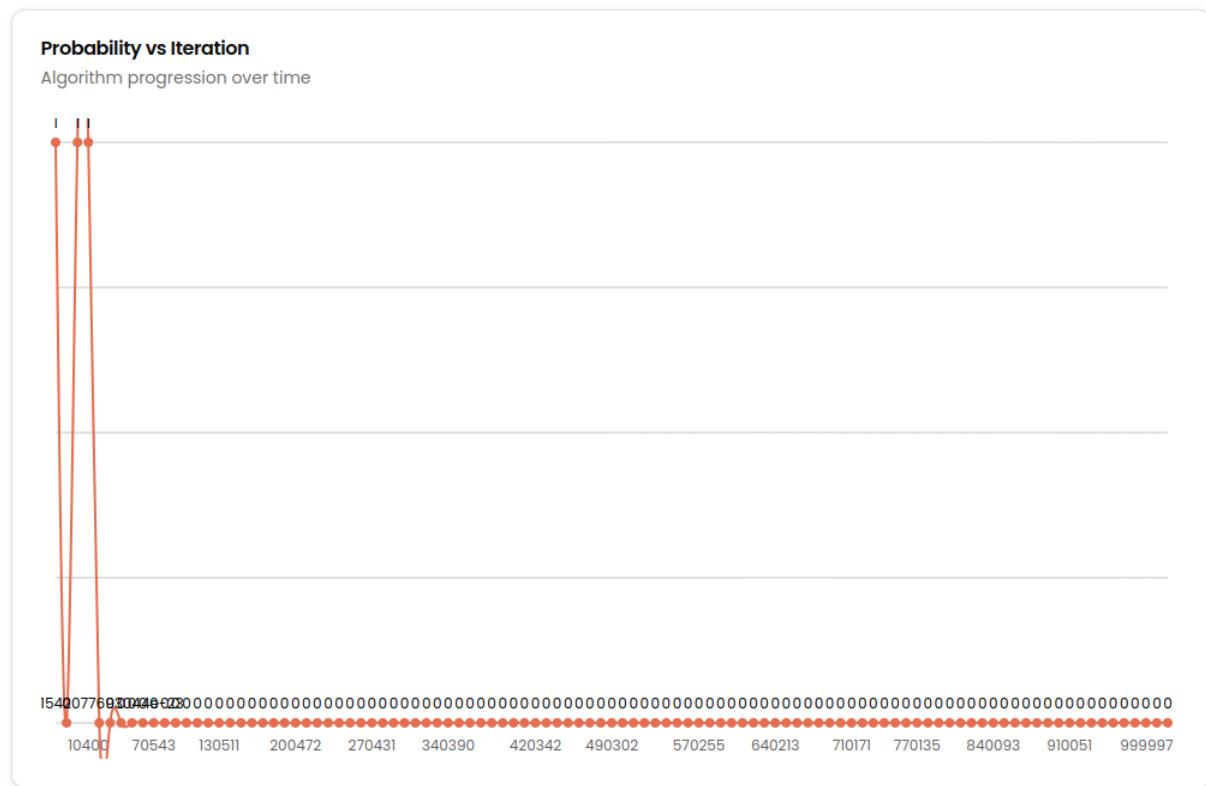
Duration: 61.58s Final Objective Function Value: -48 Iteration Count: 1000002 Stuck Frequency: 534



Objective Function vs Iteration Chart:



Probability vs Iteration Chart:



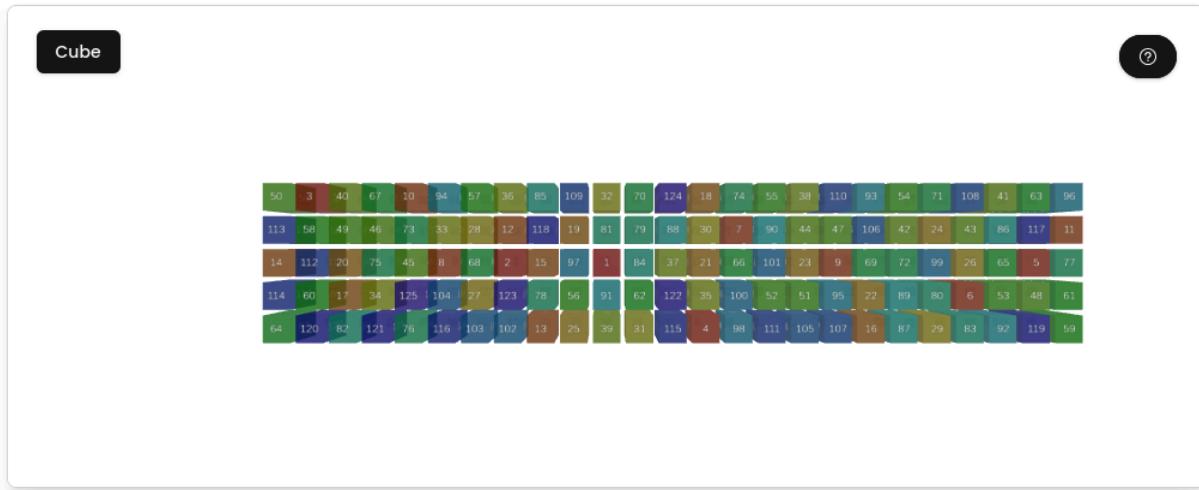
2.8.3.3 Eksperimen 3

Initial state cube:

Duration: 59.24s Final Objective Function Value: -48 Iteration Count: 1000002 Stuck Frequency: 505



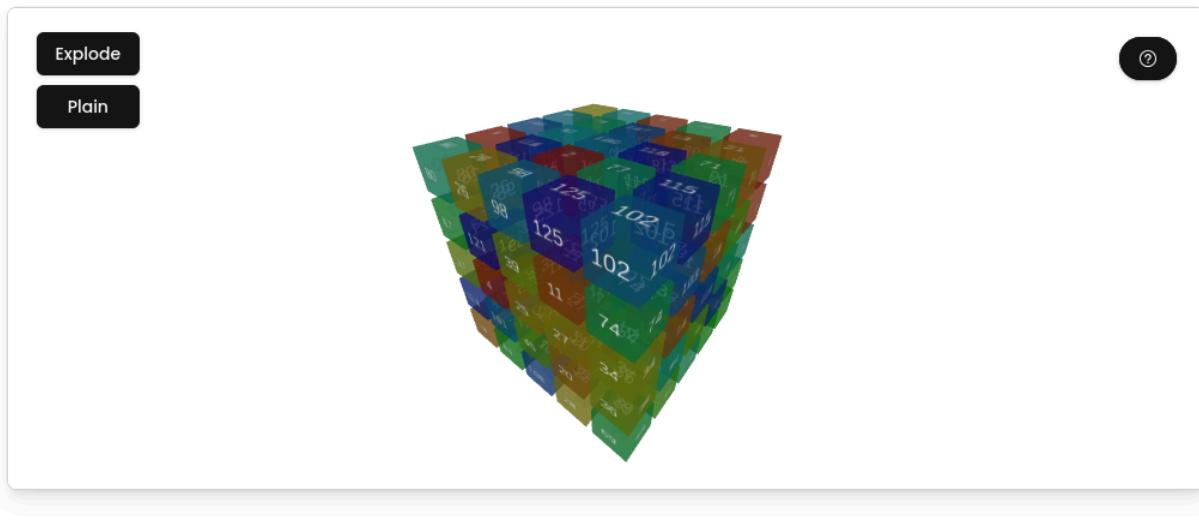
Duration: 59.24s Final Objective Function Value: -48 Iteration Count: 1000002 Stuck Frequency: 505



Step: 0 / 1000002

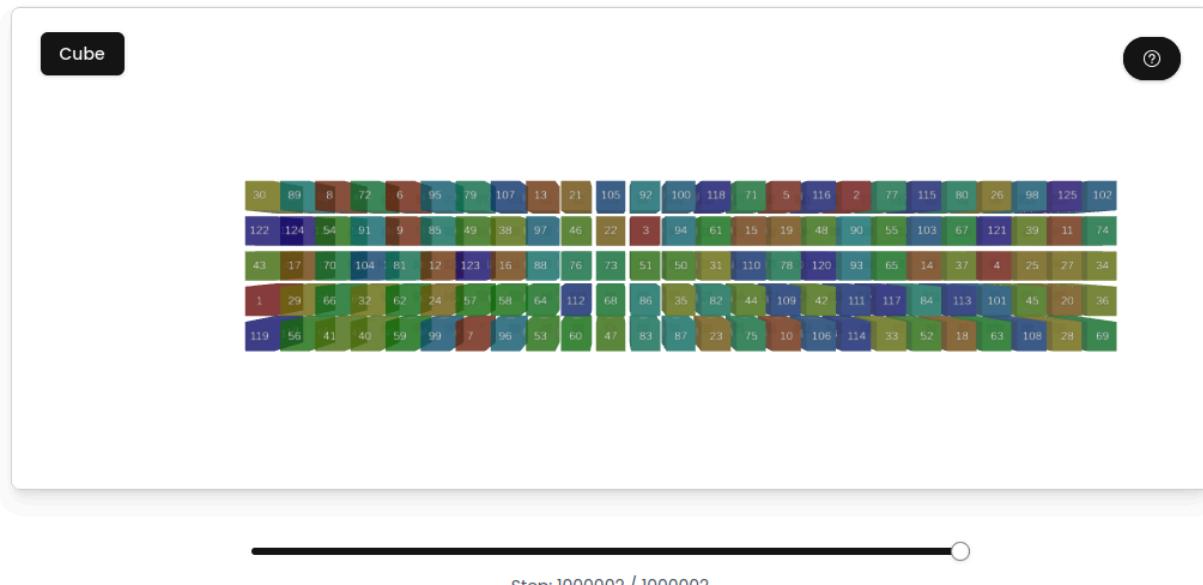
Final state cube:

Duration: 59.24s Final Objective Function Value: -48 Iteration Count: 1000002 Stuck Frequency: 505

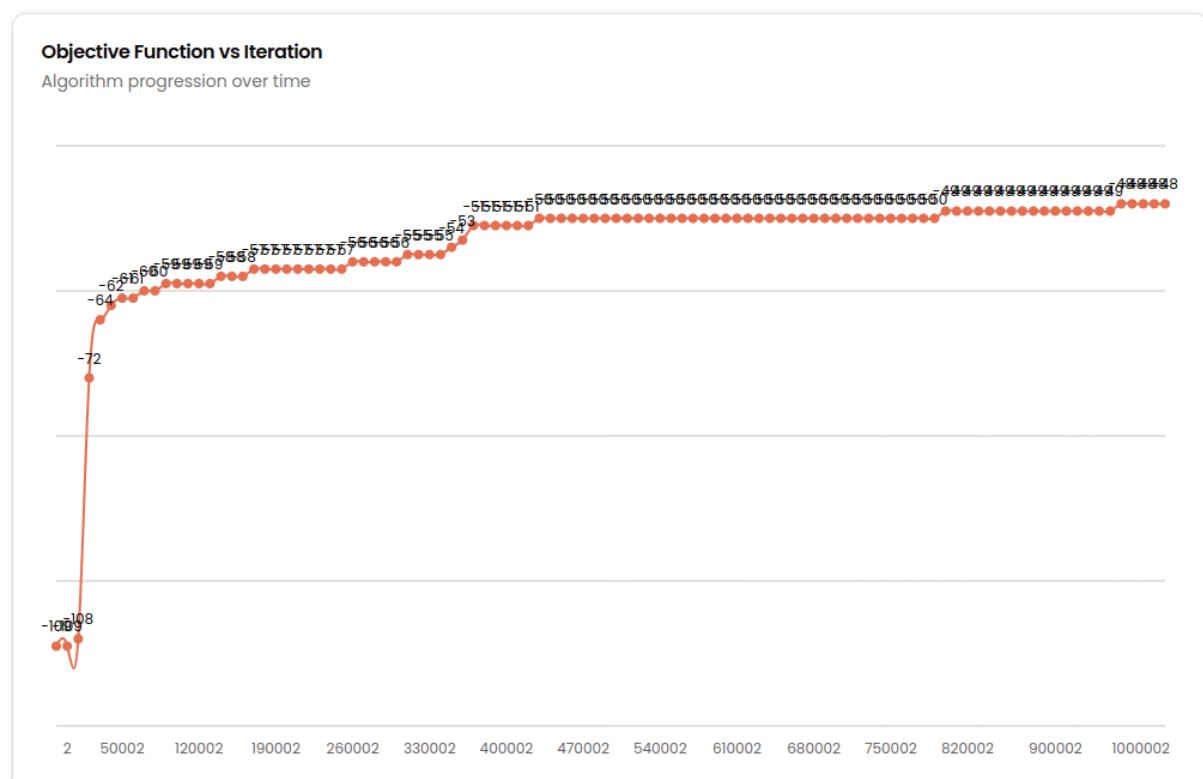


Step: 1000002 / 1000002

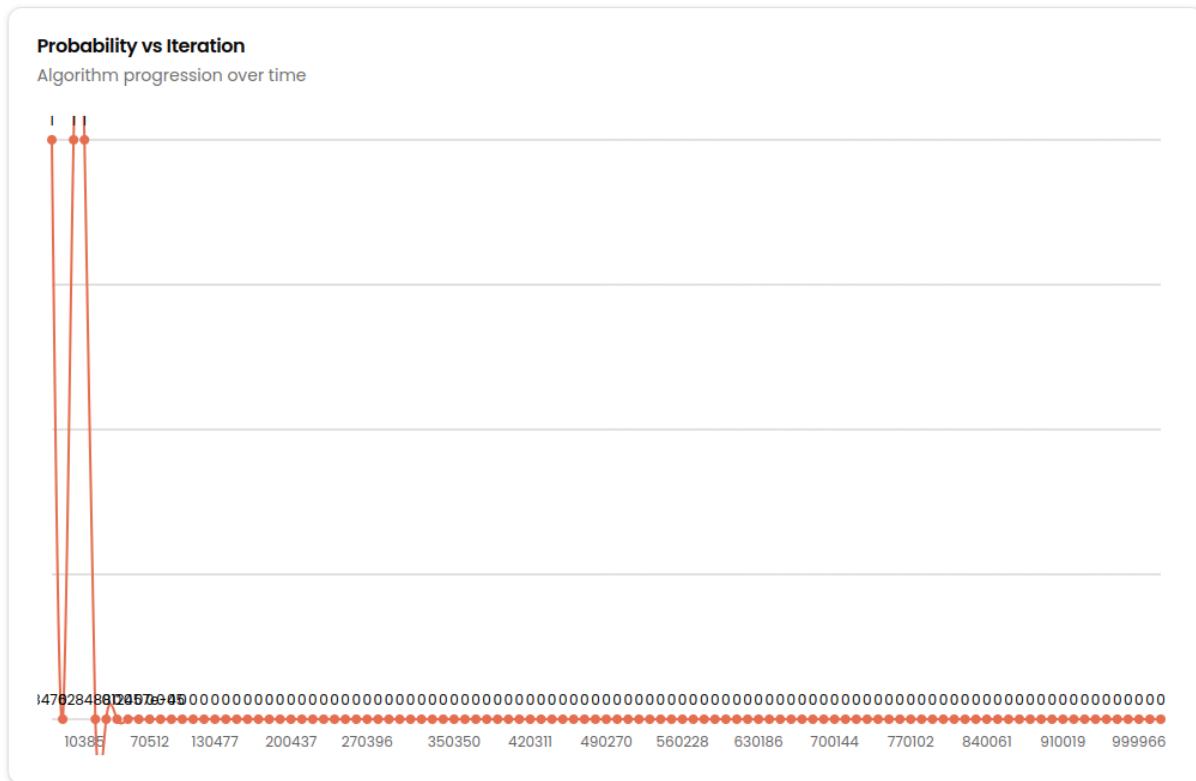
Duration: 59.24s Final Objective Function Value: -48 Iteration Count: 1000002 stuck Frequency: 505



Objective Function vs Iteration Chart:

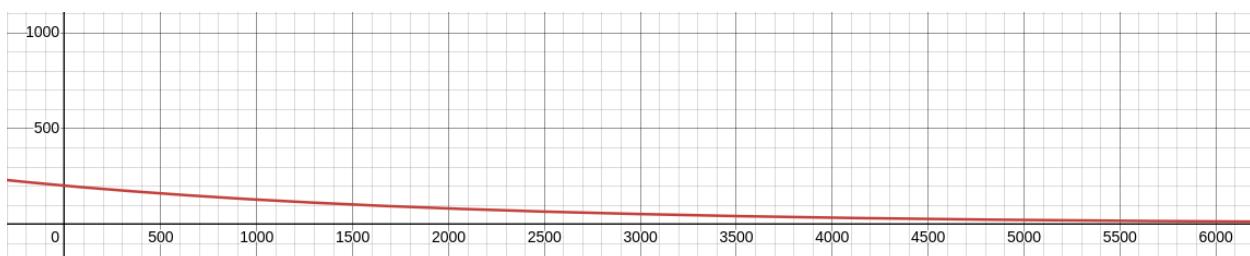


Probability vs Iteration Chart:



2.8.4 Hasil Analisis

Algoritma *simulated annealing* ditentukan oleh beberapa nilai tertentu, yaitu temperatur awal dan *scheduler function* yang memetakan waktu t dengan temperatur $T(t)$. Temperatur awal yang digunakan sebesar 200. Scheduler function yang digunakan adalah exponential decay $T(t) = T_0 \alpha^t$ dengan nilai $\alpha = 0.99955$.



Grafik temperatur terhadap waktu dari fungsi scheduler function

Seperti pada teorinya, algoritma simulated annealing mirip seperti stochastic hill climbing namun memperbolehkan untuk melakukan *bad moves* pada iterasi-iterasi awal (temperatur masih besar). Dapat dilihat pada grafik *Objective function* terhadap iterasi pada semau grafik, pada iterasi awal, *state* dapat berpindah ke *state* yang lebih buruk (melakukan *bad moves*)

sehingga nilai *objective function* dapat berkurang. Namun dapat dilihat pada grafik probabilitas terhadap iterasi, seiring bertambahnya waktu, probabilitas algoritma *simulated annealing* untuk melakukan *bad moves* berkurang, sehingga pada iterasi-iterasi yang lebih lanjut tidak ada *bad moves* yang dilakukan.

Dari ketiga percobaan yang dilakukan, dapat dilihat bahwa rata-rata waktu eksperimen berjalan sekitar 1 menit (~1 juta iterasi). Dari 1 juta iterasi yang dilakukan, rata-rata sebanyak ~ 500 *stuck* pada *local optimum*. Nilai *objective function* state terakhir terbaik yang didapat adalah -45 dan yang terburuk adalah -48. Hasil ini merupakan yang terbaik jika dibandingkan dengan algoritma-algoritma yang lain.

2.9 Algoritma Genetika

2.9.1 Penjelasan Algoritma Secara Keseluruhan

Algoritma genetika pada umumnya diimplementasi sesuai dengan salindia kuliah. *Fitness function* yang digunakan pun adalah *objective function* pada *hill climbing*. Beberapa detail implementasi yang penting dicatat adalah bahwa untuk setiap penghitungan populasi generasi berikutnya:

- $(N/2)$ individu terbaik selalu disimpan untuk populasi berikutnya. Alasan pemilihan strategi heuristik ini disebut di buku referensi perkuliahan: untuk iterasi yang tidak menggunakan heuristik ini bisa membuang individu-individu yang *fitness*-nya bagus.
- Pada potongan kode, dilakukan pemilihan dua individu *parent*, tetapi tidak ditulis karena panjang dan tidak terlalu relevan. Algoritma pemilihan adalah *roulette wheel* (sama dengan perkuliahan).

Berikut implementasinya.

```
public nextGeneration()
{
    // Here, we calculate chances for each individual (please note that
    // the actual calculation here is not written)
    const chances = this.population.map(
        cube => -cube.calculateObjectiveFunction());

    this.sortPopulation();
    this.addStateEntry(this.population[this.population_count - 1]);

    const nextPopulation = this.population.slice(this.population_count / 2,
```

```

        undefined); // keep the n/2 best parents

        for (let i = 0; i < this.population_count / 2; i++)
        {
... code here that chooses p1 and p2 from the population using a roulette wheel
algorithm omitted ...
        nextPopulation.push(this.combine(p1, p2));
    }
    this.population = nextPopulation;
}

```

Kemudian akan diiterasikan pada fungsi `run` berikut.

```

public run(): void
{
    while(this.iterations--)
    {
        this.nextGeneration();
    }
    this.sortPopulation(); // sort terakhir untuk mengambil yang paling bagus
}

```

2.9.2 Penjelasan Algoritma Penggabungan Individu

Perlu dijelaskan tentang algoritma penggabungan. Algoritma yang digunakan adalah ***partially mapped crossover***. Algoritma ini menggabungkan dua array permutasi menjadi sebuah array permutasi baru yang mempunyai karakteristik yang mirip dengan kedua individu awal.

Implementasi kodenya sebagai berikut.

```

private partiallyMappedCrossover(parent1: number[], parent2: number[]): number[] {
    const length = parent1.length;
    if (length !== 125 || parent2.length !== 125) {
        throw new Error("Both parents must be arrays of length 125");
    }

    // Initialize the child with -1 (indicating empty positions)

```

```
const child = new Array(length).fill(-1);

// Select two random crossover points
let crossoverPoint1 = Math.floor(Math.random() * length);
let crossoverPoint2 = Math.floor(Math.random() * length);
while (crossoverPoint2 === crossoverPoint1) {
    crossoverPoint2 = Math.floor(Math.random() * length);
}

// Ensure crossoverPoint1 is less than crossoverPoint2
if (crossoverPoint1 > crossoverPoint2) {
    [crossoverPoint1, crossoverPoint2]
    = [crossoverPoint2, crossoverPoint1];
}

// Copy the segment from parent1 to the child
for (let i = crossoverPoint1; i <= crossoverPoint2; i++) {
    child[i] = parent1[i];
}

// Map the elements from parent2 to the child
for (let i = crossoverPoint1; i <= crossoverPoint2; i++) {
    const element = parent2[i];
    if (!child.includes(element)) {
        let position = i;
        while (child[position] !== -1) {
            position = parent2.indexOf(parent1[position]);
        }
        child[position] = element;
    }
}

// Fill in the remaining positions from parent2
for (let i = 0; i < length; i++) {
    if (child[i] === -1) {
        child[i] = parent2[i];
    }
}

return child;
```

```
}
```

Kemudian untuk menggunakan fungsi ini, kubus harus diabstraksikan menjadi sebuah array permutasi sehingga perlu dibuat fungsi penggabungan resminya sebagai berikut.

```
public combine(p1: MagicCube, p2: MagicCube): MagicCube {
    const parent1 = [];
    const parent2 = [];
    for (let i = 0; i < 5; i++)
        for (let j = 0; j < 5; j++)
            for (let k = 0; k < 5; k++)
                { // convert parents to permutation arrays
                    parent1.push(p1.getElement([i, j, k]));
                    parent2.push(p2.getElement([i, j, k]));
                }

    // Get the child
    const child = this.partiallyMappedCrossover(parent1, parent2);

    // Convert back to cube and return
    const childCube = new MagicCube();
    for (let i = 0; i < 5; i++)
        for (let j = 0; j < 5; j++)
            for (let k = 0; k < 5; k++)
                childCube.setElement([i, j, k], child[i * 25 + j * 5 + k]);
    return childCube;
}
```

2.9.3 Hasil Eksperimen

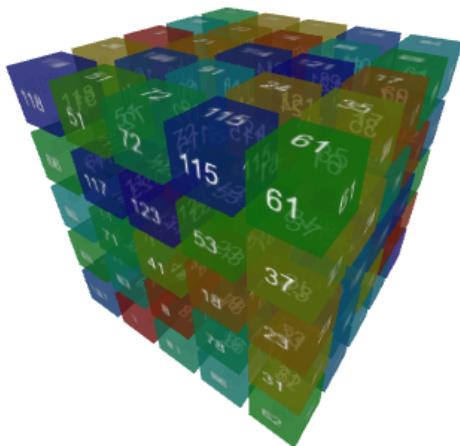
Untuk setiap variasi parameter **telah dilakukan 3 kali**, tetapi hanya diambil yang terbaik saja.

Iterasi	Populasi	Objective Function	Durasi
300	100	-92	2s
300	200	-82	4.14s

300	500	-74	10.8s
100	300	-93	2.06s
200	300	-82	4.2s
500	300	-81	10.58s

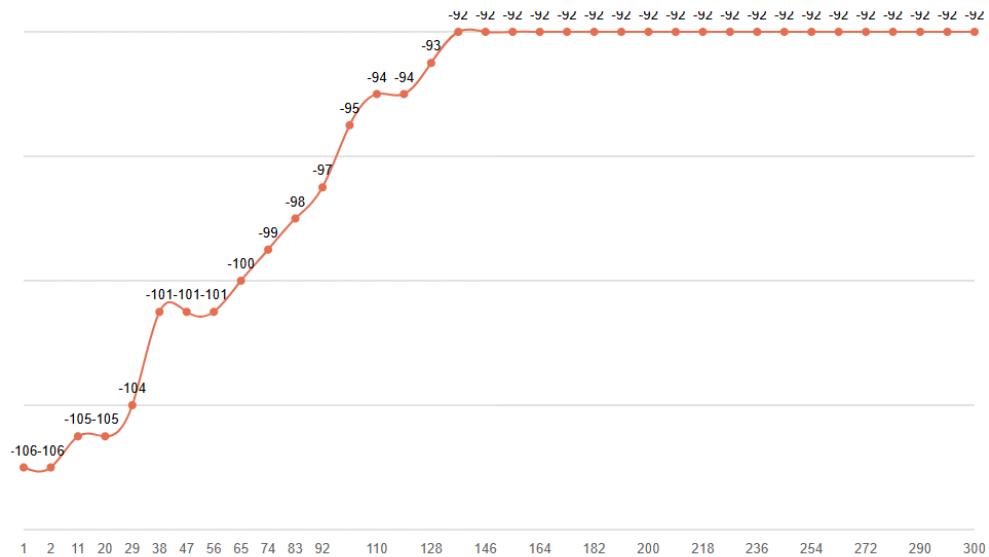
Eksperimen pertama:

Duration: 2s Final Objective Function Value: -92 Iteration Count: 300



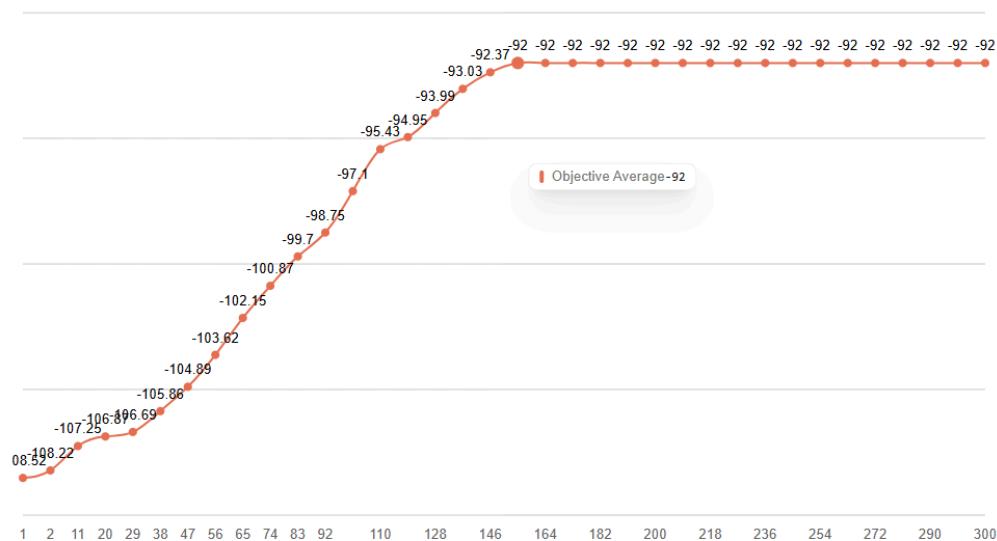
Objective Function vs Iteration

Algorithm progression over time



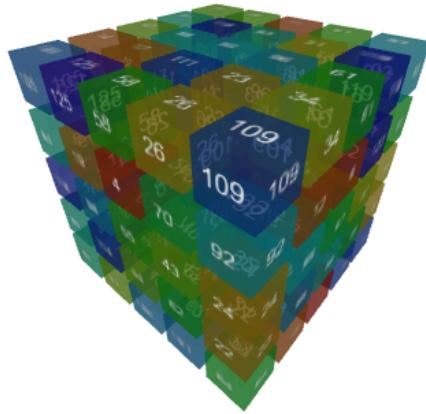
Objective Average vs Iterasi

Algorithm progression over time



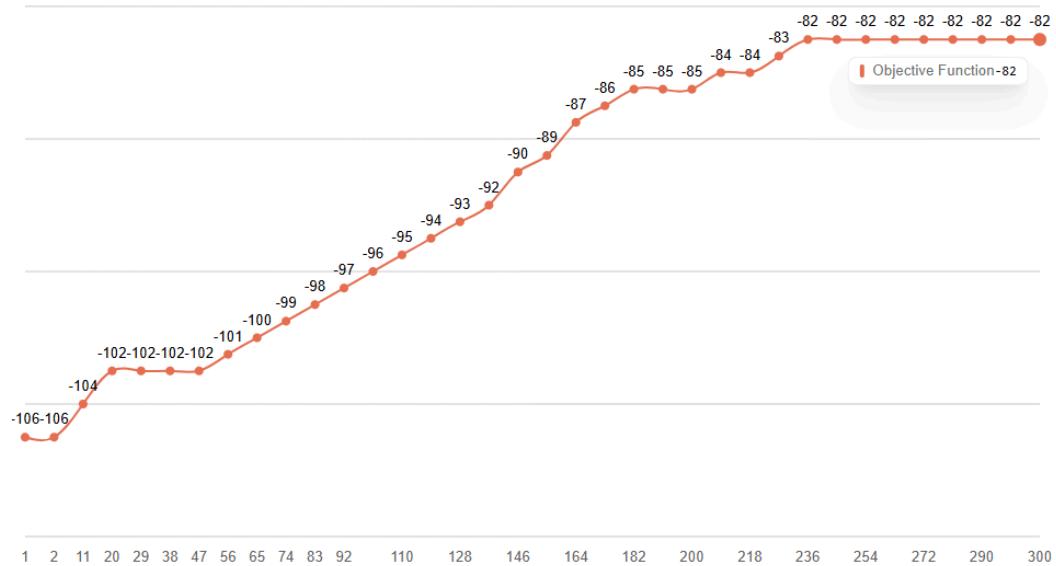
Eksperimen kedua:

Duration: 4.14s Final Objective Function Value: -82 Iteration Count: 300



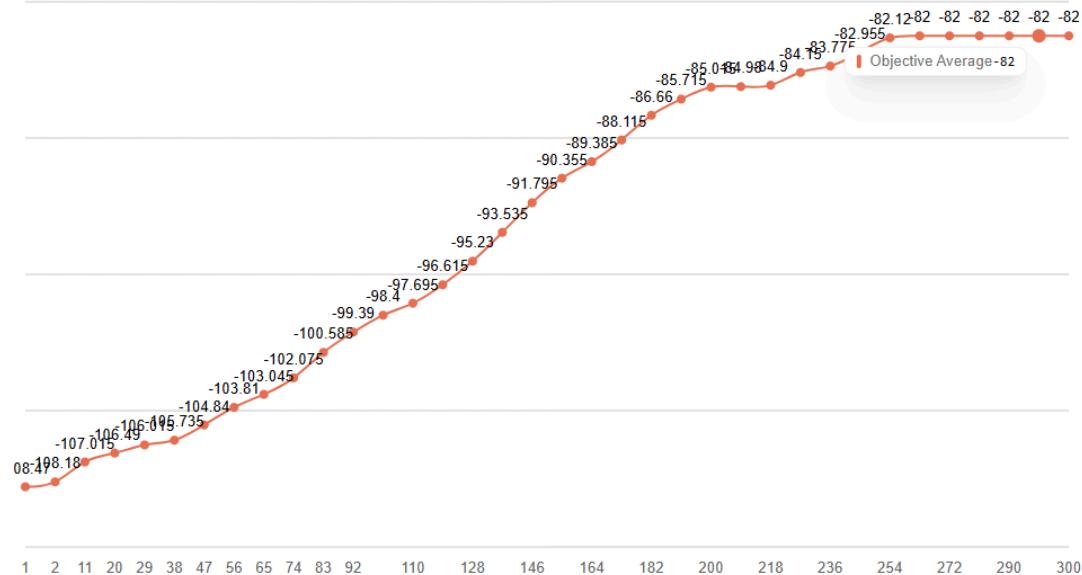
Objective Function vs Iteration

Algorithm progression over time



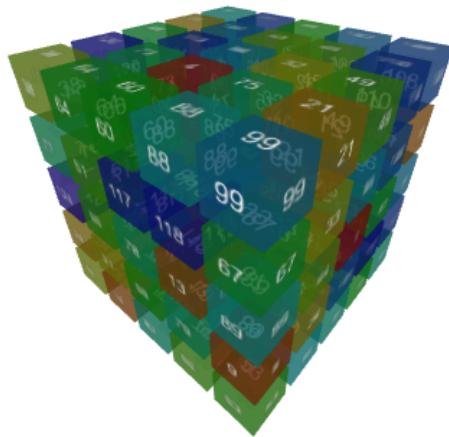
Objective Average vs Iterasi

Algorithm progression over time



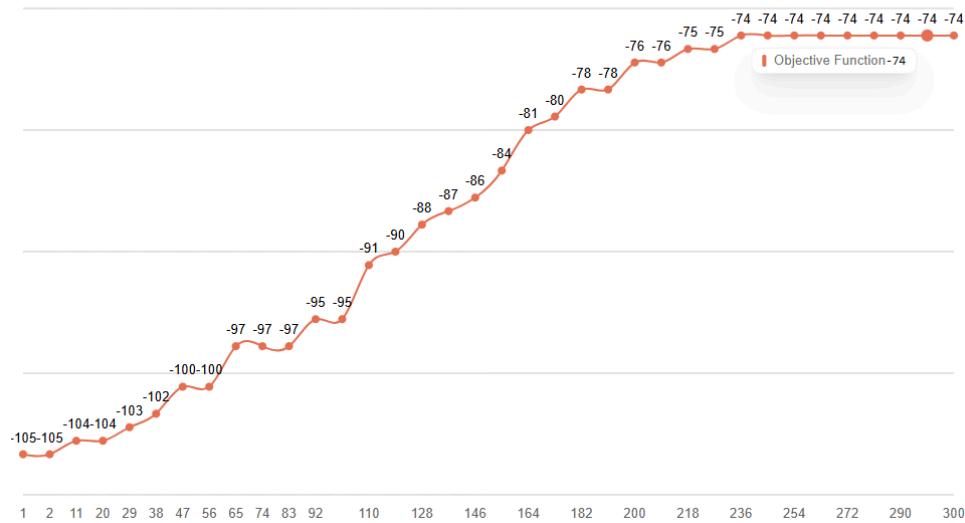
Eksperimen 3

Duration: 10.8s Final Objective Function Value: -74 Iteration Count: 300



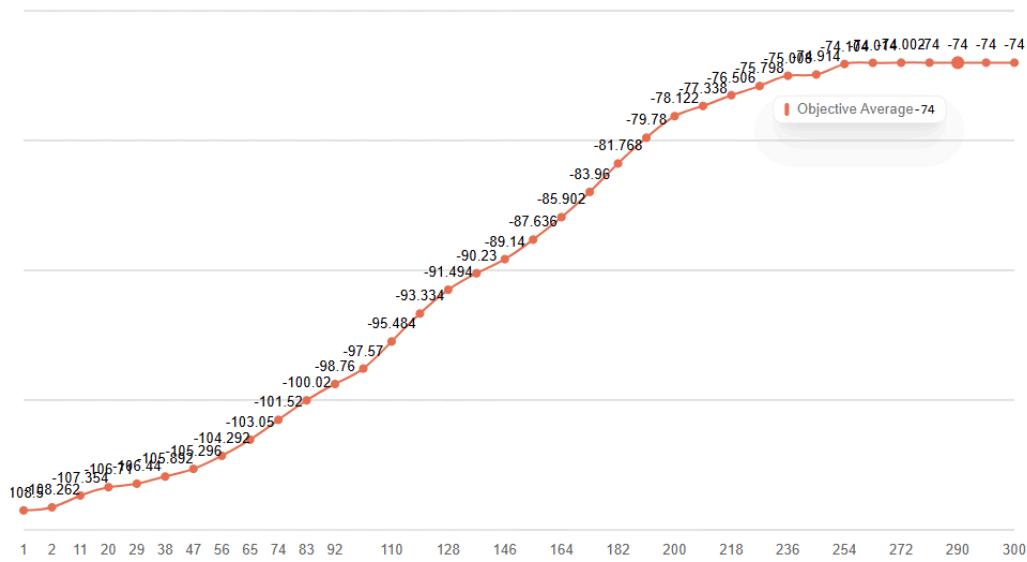
Objective Function vs Iteration

Algorithm progression over time



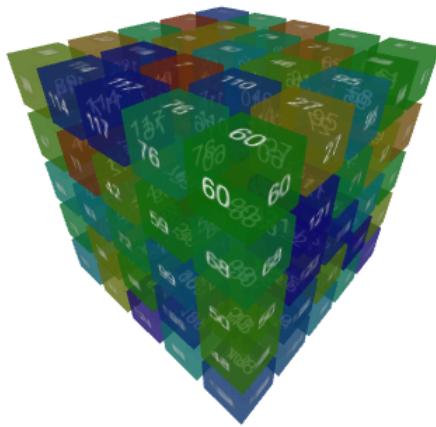
Objective Average vs Iterasi

Algorithm progression over time

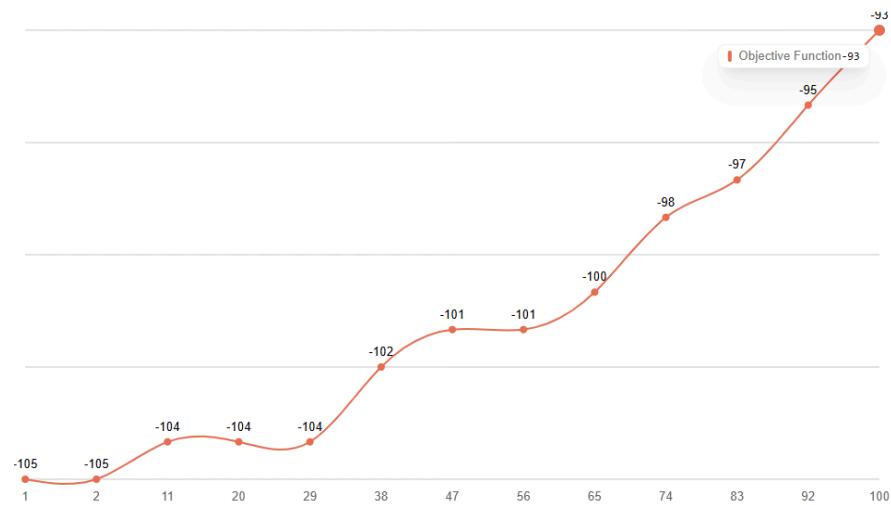


Eksperimen 4

Duration: 2.06s Final Objective Function Value: -93 Iteration Count: 100

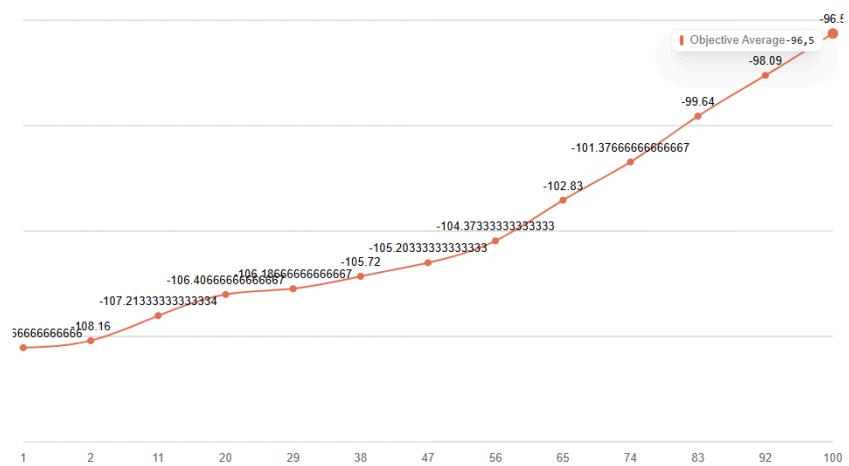


Objective Function vs Iteration
Algorithm progression over time



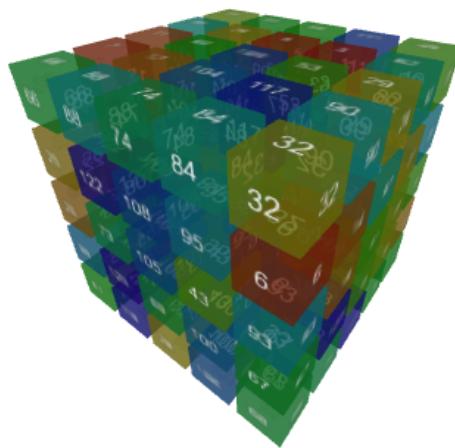
Objective Average vs Iterasi

Algorithm progression over time



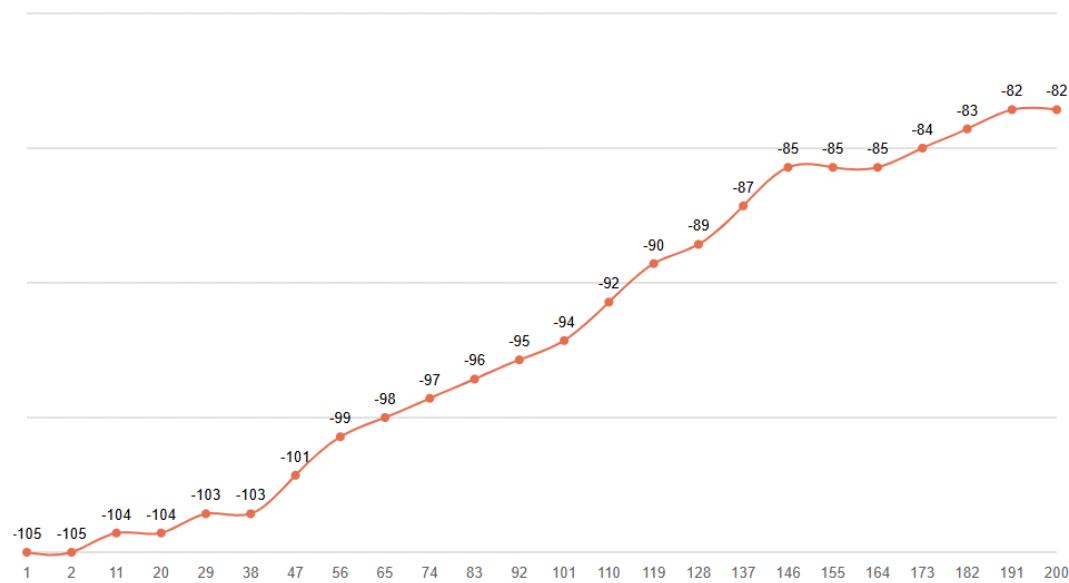
Eksperimen 5

Duration: 4.2s Final Objective Function Value: -82 Iteration Count: 200



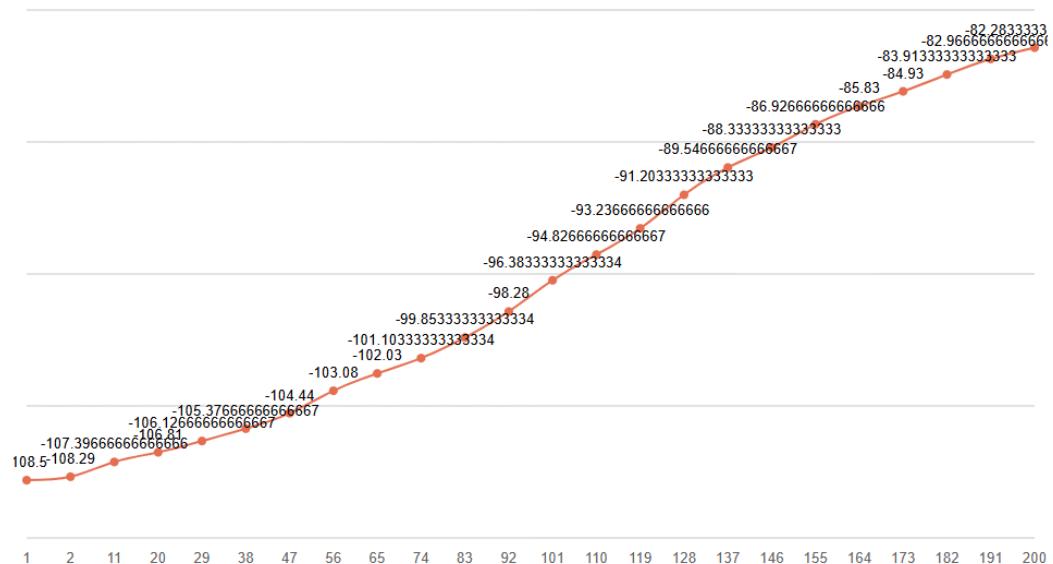
Objective Function vs Iteration

Algorithm progression over time



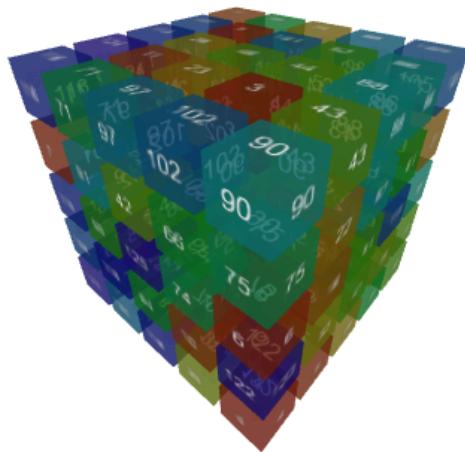
Objective Average vs Iterasi

Algorithm progression over time



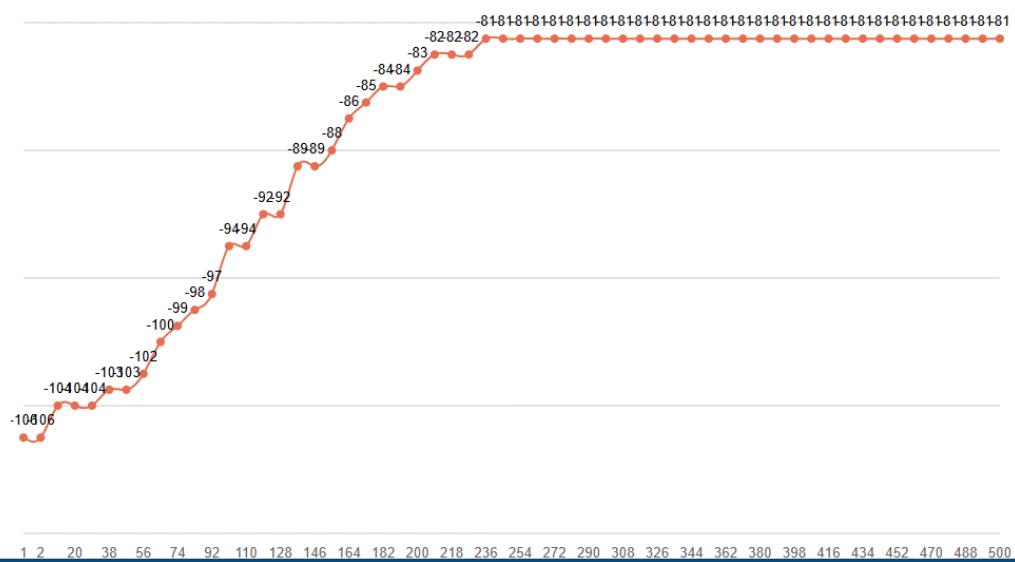
Eksperimen 6

Duration: 10.58s Final Objective Function Value: -81 Iteration Count: 500

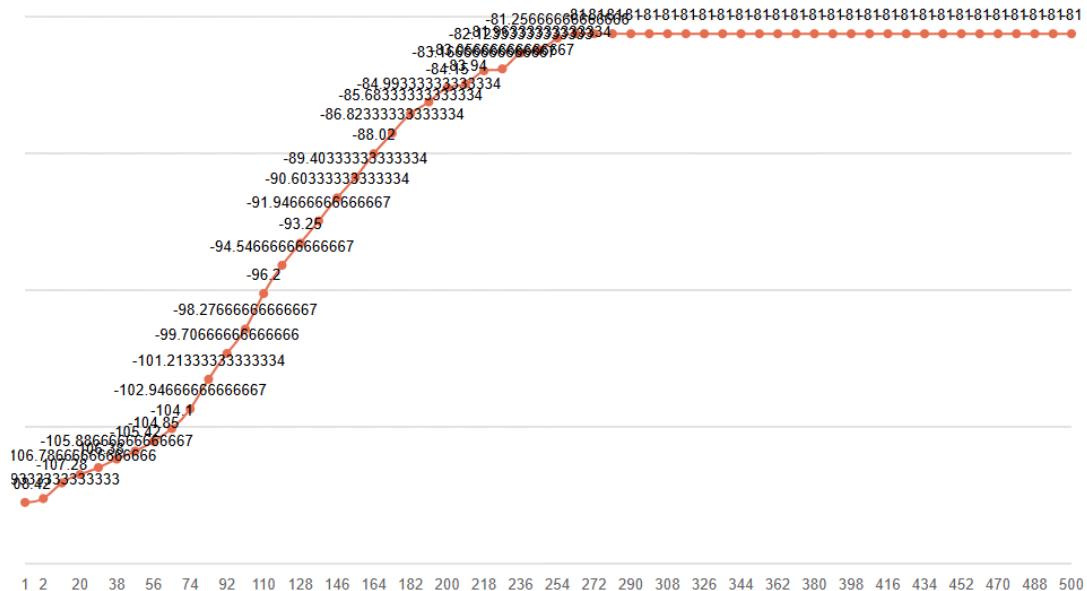


Objective Function vs Iteration

Algorithm progression over time



Objective Average vs Iterasi
Algorithm progression over time



2.9.4 Analisis Hasil Eksperimen

Dari hasil eksperimen di atas, dapat disimpulkan beberapa hal:

1. Algoritma genetika kami secara natural kurang efektif (dari segi solusi akhir) dibanding algoritma *hill climbing*. Hal ini sebenarnya bisa diperbaiki dengan menambah jumlah iterasi atau pun jumlah populasi.
2. Namun, menambah kedua parameter akan membutuhkan waktu *processing* yang lebih lama serta memori yang lebih besar, sesuai yang ditunjukkan hasil eksperimen.
3. Pada eksperimen 6, nilai akhir *objective function* kurang bertambah walaupun banyak iterasi bertambah sebanyak 2.6 kali lipat. Sebaliknya, pada eksperimen 1, 2, dan 3, nilai akhir *objective function* masih bertambah secara berbanding lurus dengan jumlah populasi. Ini menunjukkan bahwa **kedua faktor secara signifikan memengaruhi keefektifan algoritma**, tetapi parameter **Jumlah iterasi** lebih memengaruhi.
4. Algoritma genetika kami terbukti kurang baik dalam mengelola populasi secara baik. Hal ini ditunjukkan eksperimen 1, 2, 3, serta 6 yang nilai *objective function* dari individu terbaik tidak berubah dalam waktu yang signifikan (menuju waktu akhir). Hal ini mungkin karena algoritma penggabungan yang masih bisa diperbaiki.

Bab 3: Kesimpulan dan Saran

3.1 Kesimpulan

Kami telah mengimplementasi seluruh algoritma *local search* yang dipelajari pada perkuliahan, yaitu *Steepest Ascent Hill Climbing*, *Steepest Ascent with Sideways Move*, *Random Restart Hill Climbing*, *Stochastic Hill Climbing*, *Simulated Annealing*, dan *Genetic Algorithm*. Kami telah melakukan eksperimen terkait masing-masing algoritma dan menganalisis performa, durasi, dan seberapa efektif algoritma dalam menyelesaikan persoalan magic cube. Berdasarkan hasil eksperimen serta analisis, algoritma terbaik dalam menyelesaikan persoalan magic cube adalah *Simulated Annealing* yang hasil akhirnya paling mendekati state global optimum walaupun tidak sepenuhnya menyelesaikan persoalan magic cube (tidak mencapai global optimum).

3.2 Saran

Saran kedepannya dalam mencari solusi magic cube dengan local search seluruh state yang terlibat dalam proses pencarian tidak perlu disimpan dalam memori karena membuat *device* lambat (kehabisan memory). Di samping itu, algoritma dapat ditingkatkan dengan mencari strategi-strategi heuristik lain di luar dari yang sudah diimplementasi.

Pembagian Kerja

Pembagian Kerja

No	Nama	NIM	Tugas
1	Dewantoro Triatmojo	13522011	Simulated Annealing
2	Renaldy Arief Susanto	13522022	Genetic Algorithm
3	Moh Fairuz Alauddin Yahya	13522057	Frontend, Random Restart, Bonus Video
4	Rayhan Fadlan Azka	13522095	Steepest Ascent Hill Climbing, Sideways, Stochastic

Referensi

Russell, Stuart, and Peter Norvig. (2021). Artificial Intelligence a Modern Approach 4th Edition. (Diakses pada 27 September 2024).

Breedijk, Arie. "Features of the magic cube".
<https://www.magischvierkant.com/three-dimensional-eng/magic-features/>
(Diakses pada 27 September 2024).

Trump, Walter. (2003). "The Successful Search for the Smallest Perfect Magic Cube".
<https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>
(Diakses pada 27 September 2024).

Kontributor Wikipedia. "Magic Cube".
https://en.wikipedia.org/wiki/Magic_cube
(Diakses pada 27 September 2024).

Lampiran

Link Repository

<https://github.com/fairuzald/Tubes-AI-1>