

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



Disusun oleh:

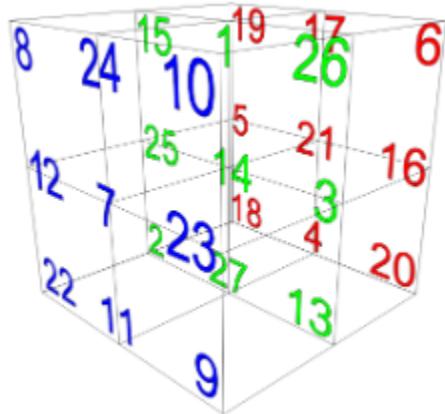
1. Maulana Muhamad Susetyo 13522127
2. Ahmad Rafi Maliki 13522137
3. Andi Marihot Sitorus 13522138
4. Nicholas Reymond Sihite 13522144

INSTITUT TEKNOLOGI BANDUNG
2023

Deskripsi Persoalan

Magic cube adalah kubus berukuran $n \times n \times n$ yang setiap bloknya diwakilkan oleh angka dari 1 sampai n^3 . Angka-angka pada *magic cube* disusun sedemikian rupa sehingga memenuhi dua properti, yaitu terdapat sebuah *magic number* dan jumlah angka untuk setiap baris, kolom, tiang, diagonal sisi (termasuk sisi pada bagian tengah kubus) harus sama dengan *magic number* tersebut.

Tugas Besar 1 IF3170 Inteligensi Artifisial meminta mahasiswa untuk memanfaatkan berbagai jenis *local search* untuk membentuk sebuah *magic cube* dari sebuah kondisi awal dan mengimplementasikannya dengan bahasa pemrograman Go dalam aplikasi GUI berbasis web. Kondisi awal yang dimaksud adalah kubus berukuran $5 \times 5 \times 5$ yang angka-angkanya acak. Setiap langkah perubahan yang boleh dilakukan adalah menukar posisi 2 angka pada kubus tersebut.



Gambar 1. Contoh Kubus Berukuran $3 \times 3 \times 3$

(sumber: <https://docs.google.com/document/d/1QDj9Pi3HrBr2VdFIvsnrA8KXaISpEr4JaGIYRxOUPWw>)

Pembahasan

1. Pemilihan Fungsi Objektif

Fungsi objektif yang kami gunakan pada pemodelan algoritma *local search* adalah dengan menghitung negatif dari jumlah selisih antara *magic number* untuk *magic cube* dengan ukuran $5 \times 5 \times 5$, yaitu 315, dengan jumlah angka tiap kolom, baris, tiang, dan diagonal sisi. Semakin bagus *magic cube*, semakin besar pula skor fungsi objektifnya. Skor maksimal yang dapat dicapai dengan perhitungan ini adalah 0, artinya tidak ada selisih antara seluruh jumlah baris, kolom, tiang, dan diagonal sisi dengan *magic number*.

Magic number dapat diperoleh dengan menggunakan rumus berikut.

$$n \times \frac{n^3 + 1}{2}, \text{ dengan } n \text{ adalah panjang sisi kubus.}$$

Pada permasalahan *magic cube*, jumlah angka untuk setiap baris, kolom, tiang, dan diagonal sisi harus sama dengan *magic number*. Maka, semakin jauh jumlah suatu kolom, baris, tiang, atau diagonal sisi dari *magic number*, semakin jauh pula *state* kubus saat itu dari *goal state*. Itulah dasar kami memilih cara ini untuk dijadikan fungsi objektif dalam permasalahan *magic cube*.

Sebagai contoh, misalkan angka-angka pada sebuah kubus $5 \times 5 \times 5$ adalah [[[1, 15, 7, 120, 4], ..., ..., [[97, 88, 3, 2, 41]]]. Skor fungsi objektif kubus ini dihitung dengan cara $-1 \times [| (1 + 15 + 7 + 120 + 4) - 315 | + \dots + | (97 + 88 + 3 + 2 + 4) - 315 | + \text{jumlah kolom} + \text{jumlah diagonal sisi}]$, yaitu $-(289 + \text{jumlah lainnya})$.

Berikut merupakan implementasi fungsi objektif yang telah dibuat dalam bahasa Go untuk sebuah konfigurasi *magic cube*.

```
func (mc *MagicCube) ObjectiveFunction() int {
    dimension := mc.Size
    score := 0
    magicNumber := dimension * (dimension*dimension*dimension + 1) / 2

    for y := 0; y < dimension; y++ {
        for z := 0; z < dimension; z++ {
            rowSum := 0
            for x := 0; x < dimension; x++ {
                rowSum += mc.Buffer[x][y][z]
            }
            score += AbsoluteVal(rowSum - magicNumber)
        }
    }

    for x := 0; x < dimension; x++ {
        for y := 0; y < dimension; y++ {
            columnSum := 0
            for z := 0; z < dimension; z++ {
                columnSum += mc.Buffer[x][y][z]
            }
            score += AbsoluteVal(columnSum - magicNumber)
        }
    }

    for x := 0; x < dimension; x++ {
        for z := 0; z < dimension; z++ {
            diagonalSum := 0
            for y := 0; y < dimension; y++ {
                diagonalSum += mc.Buffer[x][y][z]
            }
            score += AbsoluteVal(diagonalSum - magicNumber)
        }
    }

    for z := 0; z < dimension; z++ {
        for y := 0; y < dimension; y++ {
            diagonalSum := 0
            for x := 0; x < dimension; x++ {
                diagonalSum += mc.Buffer[x][y][z]
            }
            score += AbsoluteVal(diagonalSum - magicNumber)
        }
    }
}
```

```

        for z := 0; z < dimension; z++ {
            columnSum += mc.Buffer[x][y][z]
        }
        score += AbsoluteVal(columnSum - magicNumber)
    }
}

for x := 0; x < dimension; x++ {
    for z := 0; z < dimension; z++ {
        pillarSum := 0
        for y := 0; y < dimension; y++ {
            pillarSum += mc.Buffer[x][y][z]
        }
        score += AbsoluteVal(pillarSum - magicNumber)
    }
}

for z := 0; z < dimension; z++ {
    diag1XY := 0
    diag2XY := 0

    for i := 0; i < dimension; i++ {
        diag1XY += mc.Buffer[i][i][z]
        diag2XY += mc.Buffer[i][dimension-i-1][z]
    }

    score += AbsoluteVal(diag1XY - magicNumber)
    score += AbsoluteVal(diag2XY - magicNumber)
}

for y := 0; y < dimension; y++ {
    diag1XZ := 0
    diag2XZ := 0

    for i := 0; i < dimension; i++ {
        diag1XZ += mc.Buffer[i][y][i]
        diag2XZ += mc.Buffer[i][y][dimension-i-1]
    }

    score += AbsoluteVal(diag1XZ - magicNumber)
    score += AbsoluteVal(diag2XZ - magicNumber)
}

for x := 0; x < dimension; x++ {
    diag1YZ := 0
    diag2YZ := 0

    for i := 0; i < dimension; i++ {
        diag1YZ += mc.Buffer[x][i][i]
        diag2YZ += mc.Buffer[x][i][dimension-i-1]
    }

    score += AbsoluteVal(diag1YZ - magicNumber)
    score += AbsoluteVal(diag2YZ - magicNumber)
}

```

```

mainDiag3D := 0
antiDiag1_3D := 0
antiDiag2_3D := 0
antiDiag3_3D := 0

for i := 0; i < dimension; i++ {
    mainDiag3D += mc.Buffer[i][i][i]
    antiDiag1_3D += mc.Buffer[i][dimension-i-1][i]
    antiDiag2_3D += mc.Buffer[dimension-i-1][i][i]
    antiDiag3_3D += mc.Buffer[dimension-i-1][dimension-i-1][i]
}

score += AbsoluteVal(mainDiag3D - magicNumber)
score += AbsoluteVal(antiDiag1_3D - magicNumber)
score += AbsoluteVal(antiDiag2_3D - magicNumber)
score += AbsoluteVal(antiDiag3_3D - magicNumber)

return -score
}

```

2. Penjelasan Implementasi Algoritma *Local Search*

Algoritma *local search* yang digunakan pada program ini untuk mencari solusi permasalahan *diagonal magic cube* adalah *hill-climbing* (keempat varian), *simulated annealing*, dan *genetic algorithm*. Representasi *magic cube* dalam program ini adalah sebuah *struct* yang bernama MagicCube dengan spesifikasi sebagai berikut.

```

package MagicCube

type MagicCube struct {
    Size      int
    Score    int
    Buffer   [][]][]int
}

type Response struct {
    // semua fungsi
    Buffer           [][]][]int
    IndexChange     [][]][]int
    ObjectiveFunctions []int
    ExecutionTimeInMS int
    CubeStates       [][]][][]int

    //Steepest Ascent, Sideways Move HC, Simulated Annealing
    Iterations int

    // Random Restart HC
    RestartCount      int
    RestartPerIteration []int

    // simulated annealing
    LocalOptimum int
}

```

```

DeltaE      []int

// genetic algorithm
ObjectiveFunctionsMean []int
}

// fungsi untuk membuat magic cube baru
func New(size ...int) *MagicCube

// prosedur untuk menampilkan konfigurasi magic cube
func (mc *MagicCube) Print()

// prosedur untuk mengacak angka-angka pada magic cube
func (mc *MagicCube) Shuffle()

// fungsi untuk menyalin kubus
func (mc *MagicCube) Copy() *MagicCube

// fungsi untuk menghitung nilai mutlak
func AbsoluteVal(n int) int

// fungsi untuk mengubah integer menjadi koordinat pada kubus
func IntToThreeDee(n int) [3]int

// fungsi untuk menghitung skor fungsi objektif kubus
func (mc *MagicCube) ObjectiveFunction() int

// fungsi untuk mengambil indeks acak
func (mc *MagicCube) GetRandomIdx() [3]int

// prosedur untuk menukar angka pada indeks idx1 dengan idx2
func (mc *MagicCube) SwapValues(idx1 [3]int, idx2 [3]int)

// prosedur untuk mengubah skor magic cube
func (mc *MagicCube) SetScore(score int)

```

Berikut merupakan penjelasan implementasi seluruh algoritma tersebut dalam program yang telah dibuat.

2.1. *Steepest Ascent Hill-Climbing*

Berikut adalah algoritma Steepest Ascent Hill-Climbing, diimplementasikan dalam fungsi SteepestAscentHillClimbing(cube *MagicCube.MagicCube) dan mengembalikan struct MagicCube.Response berisikan hasil, urutan swap, nilai objective function, waktu eksekusi, state magic cube, dan jumlah iterasi.

Selain variabel untuk mencatat hasil untuk Response, dibuat map dengan key pair [2]int dan value int, ini diisi dalam loop yang mengiterasikan setiap pasangan dua titik pada magic cube dan menghitung nilai objective function setelah swap (mencari neighbour). Kemudian Dilakukan swap pada koordinat yang sama untuk mengembalikan magic cube ke state sebelum di swap sekaligus mencari nilai tertinggi dari neighboring states. Jika semua sudah teriterasikan dan nilai tertinggi sudah

ditemukan, maka akan dicari neighbor dengan nilai yang sama dengan nilai tertinggi dan dipilih secara random.

Ini akan terus dilakukan sampai tidak ada neighbour yang memiliki objective value lebih besar dari objective value sekarang. Berikut adalah potongan kodennya

```
func SteepestAscentHillClimbing(cube *MagicCube.MagicCube)
(MagicCube.Response, error) {
    start := time.Now()
    pairs := make(map[[2]int]int)
    indexChange := [][][]int{}
    objectiveFunctions := []int{}
    currentObjectiveValue := cube.ObjectiveFunction()
    objectiveFunctions = append(objectiveFunctions,
    currentObjectiveValue)
    maxObjectiveValue := currentObjectiveValue
    cubeStates := [][][][]int{}
    cubeStates = append(cubeStates, cube.Buffer)
    iter := 0
    for {
        res := 0
        for i := 0; i < 125; i++ {
            for j := i + 1; j < 125; j++ {
                cube.SwapValues(MagicCube.IntToThreeDee(i),
                MagicCube.IntToThreeDee(j))
                res = cube.ObjectiveFunction()
                cube.SwapValues(MagicCube.IntToThreeDee(i),
                MagicCube.IntToThreeDee(j))
                pairs[[2]int{i, j}] = res

                if res > maxObjectiveValue {
                    maxObjectiveValue = res
                }
            }
        }
        iter++
        if currentObjectiveValue >= maxObjectiveValue {
            break
        }

        var maxPairs [][]int

        for pair, value := range pairs {
            if value == maxObjectiveValue {
                maxPairs = append(maxPairs, pair)
            }
        }
        randomIndex := rand.Intn(len(maxPairs))
        randomPair := maxPairs[randomIndex]

        swapOne := MagicCube.IntToThreeDee(randomPair[0])
        swapTwo := MagicCube.IntToThreeDee(randomPair[1])
```

```

        indexChange = append(indexChange, [][]int{swapOne[:],
swapTwo[:]})
        objectiveFunctions = append(objectiveFunctions,
maxObjectiveValue)
        cube.SwapValues(swapOne, swapTwo)
        cubeStates = append(cubeStates, cube.Buffer)
        currentObjectiveValue = maxObjectiveValue
    }

    elapsed := time.Since(start)
    executionTimeInMS := int(elapsed.Milliseconds())

    Result := MagicCube.Response{
        Buffer:           cube.Buffer,
        IndexChange:     indexChange,
        ObjectiveFunctions: objectiveFunctions,
        ExecutionTimeInMS: executionTimeInMS,
        CubeStates:       cubeStates,
        Iterations:      iter,
    }
    return Result, nil
}

```

2.2. Hill-Climbing with Sideways Move

Berikut adalah implementasi algoritma Hill-Climbing with Sideways Move. Fungsi SideWaysMoveHillClimbing(cube *MagicCube.MagicCube, maxOcc int) adalah *Carbon Copy* dari fungsi SteepestAscentHillClimbing() dengan beberapa modifikasi.

Pertama, sesuai spek, jumlah maksimum kemunculan nilai objective function tertinggi dijadikan sebagai parameter. Ini digunakan untuk mencegah terjadinya infinite loop saat terjebak di local optima.

Kemudian, yang membedakannya dari Steepest Ascent Hill-Climbing adalah, sesuai namanya, Hill-climbing with Sideways Move tidak langsung berhenti ketika tidak ada neighbor dengan objective value yang lebih besar, tetapi memperbolehkan untuk bergerak ke neighbor dengan objective value yang sama dengan nilai sekarang.

```

func SideWaysMoveHillClimbing(cube *MagicCube.MagicCube, maxOcc int)
(MagicCube.Response, error) {
    start := time.Now()
    pairs := make(map[[2]int]int)
    countOcc := 0
    iter := 0
    indexChange := [][][]int{}
    objectiveFunctions := []int{}
    currentObjectiveValue := cube.ObjectiveFunction()
    objectiveFunctions = append(objectiveFunctions,
currentObjectiveValue)
    maxObjectiveValue := currentObjectiveValue
}

```

```

cubeStates := [][][][]int{}
cubeStates = append(cubeStates, cube.Buffer)
for {
    res := 0
    for i := 0; i < 125; i++ {
        for j := i + 1; j < 125; j++ {
            cube.SwapValues(MagicCube.IntToThreeDee(i),
MagicCube.IntToThreeDee(j))
            res = cube.ObjectiveFunction()
            cube.SwapValues(MagicCube.IntToThreeDee(i),
MagicCube.IntToThreeDee(j))
            pairs[[2]int{i, j}] = res
            if res > maxObjectiveValue {
                maxObjectiveValue = res
                countOcc = 0
            }
        }
    }
    countOcc++
    iter++
    if currentObjectiveValue > maxObjectiveValue || countOcc >
maxOcc {
        break
    }
}

var maxPairs [][]int

for pair, value := range pairs {
    if value == maxObjectiveValue {
        maxPairs = append(maxPairs, pair)
    }
}

if len(maxPairs) > 0 {
    randomIndex := rand.Intn(len(maxPairs))
    randomPair := maxPairs[randomIndex]

    swapOne := MagicCube.IntToThreeDee(randomPair[0])
    swapTwo := MagicCube.IntToThreeDee(randomPair[1])

    objectiveFunctions = append(objectiveFunctions,
maxObjectiveValue)
    indexChange = append(indexChange, [][]int{swapOne[:],
swapTwo[:]})
    cube.SwapValues(swapOne, swapTwo)
    cubeStates = append(cubeStates, cube.Buffer)
}
}

elapsed := time.Since(start)
executionTimeInMS := int(elapsed.Milliseconds())

Result := MagicCube.Response{
    Buffer:                 cube.Buffer,
    IndexChange:           indexChange,
}

```

```

        ObjectiveFunctions: objectiveFunctions,
        ExecutionTimeInMS: executionTimeInMS,
        CubeStates: cubeStates,
        Iterations: iter,
    }

    return Result, nil
}

```

2.3. Random Restart Hill-Climbing

Berikut adalah algoritma Random Restart Hill-Climbing. Algoritma ini menggunakan basis yang sama dengan algoritma Steepest Ascent Hill-Climbing, Dimana akan dilakukan iterasi berulang sampai meraih batas jumlah iterasi (ditetapkan sebagai parameter), atau mendapatkan objective value 0.

```

func RandomRestartHillClimbing(cube *MagicCube.MagicCube, numOfRestart
int) (MagicCube.Response, error) {
    start := time.Now()

    indexChange := [][][]int{}
    objectiveFunctions := []int{}
    currentObjectiveValue := cube.ObjectiveFunction()
    objectiveFunctions = append(objectiveFunctions,
currentObjectiveValue)
    maxObjectiveValue := currentObjectiveValue
    cubeStates := [][]][]int{}
    cubeStates = append(cubeStates, cube.Buffer)
    restartCount := 0
    iteration := 0
    restartPerIteration := []int{}

    for {
        for {
            pairs := make(map[[2]int]int)
            currentObjectiveValue = cube.ObjectiveFunction()
            maxObjectiveValue = currentObjectiveValue
            res := 0
            for i := 0; i < 125; i++ {
                for j := i + 1; j < 125; j++ {
                    cube.SwapValues(MagicCube.IntToThreeDee(i),
MagicCube.IntToThreeDee(j))
                    res = cube.ObjectiveFunction()
                    cube.SwapValues(MagicCube.IntToThreeDee(i),
MagicCube.IntToThreeDee(j))

                    if res > maxObjectiveValue {
                        pairs[[2]int{i, j}] = res
                        maxObjectiveValue = res
                    }
                }
            }
        }
    }
}

```

```

        if currentObjectiveValue >= maxObjectiveValue {
            break
        }

        var maxPairs [][]int

        for pair, value := range pairs {
            if value == maxObjectiveValue {
                maxPairs = append(maxPairs, pair)
            }
        }
        randomIndex := rand.Intn(len(maxPairs))
        randomPair := maxPairs[randomIndex]
        swapOne := MagicCube.IntToThreeDee(randomPair[0])
        swapTwo := MagicCube.IntToThreeDee(randomPair[1])

        indexChange = append(indexChange, [][]int{swapOne[:],
        swapTwo[:]})
        objectiveFunctions = append(objectiveFunctions,
        maxObjectiveValue)
        cube.SwapValues(swapOne, swapTwo)
        cubeStates = append(cubeStates, cube.Buffer)
        iteration++
    }

    restartPerIteration = append(restartPerIteration, iteration)

    sz := MagicCube.IntToThreeDee(0)
    indexChange = append(indexChange, [][]int{sz[:], sz[:]})
    iteration = 0
    if restartCount >= numRestart || maxObjectiveValue == 0 {
        break
    }
    restartCount++
    cube.Shuffle()
}

elapsed := time.Since(start)
executionTimeInMS := int(elapsed.Milliseconds())

Result := MagicCube.Response{
    Buffer:                 cube.Buffer,
    IndexChange:           indexChange,
    ObjectiveFunctions:   objectiveFunctions,
    ExecutionTimeInMS:    executionTimeInMS,
    CubeStates:            cubeStates,
    RestartCount:          restartCount,
    RestartPerIteration:  restartPerIteration,
}
return Result, nil
}

```

2.4. Stochastic Hill-Climbing

Berikut adalah algoritma Stochastic Hill-Climbing. Ditetapkan iterasi sebanyak 100000 di awal. Dilakukan iterasi sebanyak 100000 kali dengan tiap iterasi melakukan hal berikut:

- Mencari objective value salah satu tetangga dengan swap (dipilih secara random)
- Membandingkannya dengan objective value saat ini
- Jika lebih besar, swap tetap. Jika tidak, akan di-swap pada koordinat yang sama

Berikut adalah potongan kodennya.

```
func StochasticHillClimbing(cube *MagicCube.MagicCube)
(MagicCube.Response, error) {
    start := time.Now()
    cur := cube.ObjectiveFunction()
    iter := 100000
    indexChange := [][][]int{}
    objectiveFunctions := []int{}
    currentObjectiveValue := cube.ObjectiveFunction()
    objectiveFunctions = append(objectiveFunctions,
    currentObjectiveValue)
    cubeStates := [][][]int{}
    cubeStates = append(cubeStates, cube.Buffer)
    for i := 0; i < iter; i++ {
        num1 := rand.Intn(125)
        var num2 int
        for {
            num2 = rand.Intn(125)
            if num1 != num2 {
                break
            }
        }
        swapOne := MagicCube.IntToThreeDee(num1)
        swapTwo := MagicCube.IntToThreeDee(num2)
        cube.SwapValues(swapOne, swapTwo)
        res := cube.ObjectiveFunction()

        if res > cur {
            cur = res
            objectiveFunctions = append(objectiveFunctions, res)
            indexChange = append(indexChange, [][]int{swapOne[:],
            swapTwo[:]})
            cubeStates = append(cubeStates, cube.Buffer)
        } else {
            cube.SwapValues(swapOne, swapTwo)
        }
    }
    elapsed := time.Since(start)

    executionTimeInMS := int(elapsed.Milliseconds())
    Result := MagicCube.Response{
        Buffer:                 cube.Buffer,
        IndexChange:           indexChange,
```

```

    ObjectiveFunctions: objectiveFunctions,
    ExecutionTimeInMS: executionTimeInMS,
    CubeStates: cubeStates,
    Iterations: iter,
}
return Result, nil
}

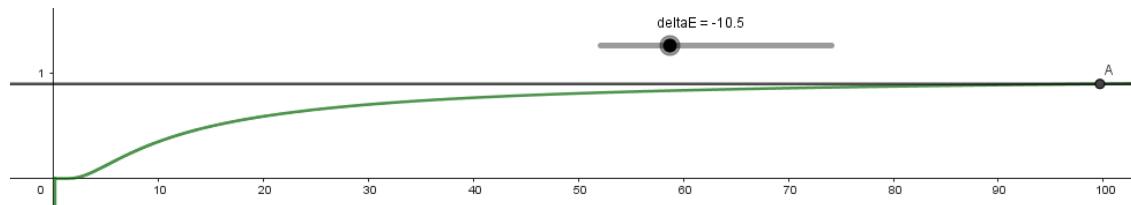
```

2.5. Simulated Annealing

Algoritma *simulated annealing* diimplementasikan dalam sebuah fungsi yang bernama `SimulatedAnnealing()` yang menerima masukan konfigurasi kubus awal dan menghasilkan sebuah *struct* bernama `Response` yang terdiri dari: Buffer yang menyimpan konfigurasi kubus akhir; `IndexChange` yang menyimpan urutan perubahan indeks untuk setiap langkah, dan `ObjectiveFunctions` yang menyimpan skor fungsi objektif untuk setiap langkah.

Di awal, fungsi akan melakukan inisialisasi nilai-nilai variabel yang diperlukan untuk menjalankan algoritma dan menampilkan hasilnya. Setelah itu, fungsi memeriksa apakah kubus saat ini sudah mencapai maksimum global atau belum. Jika belum, program akan menjalankan algoritma *simulated annealing*. Temperatur pada awal iterasi diatur menjadi 100 dan akan berkurang sebanyak 0.001 setiap iterasi sehingga akan ada maksimal sebanyak 100.000 iterasi.

Selama melakukan iterasi, fungsi akan memeriksa apakah temperatur sudah mencapai 0 atau belum. Jika sudah, hentikan proses pencarian. Jika belum, lakukan penukaran indeks secara acak (menandakan perpindahan *state*). Jika skor fungsi objektif konfigurasi yang baru sudah maksimum global (0), hentikan pencarian. Jika belum, bandingkan skornya dengan skor kubus saat ini. Jika lebih baik, pilih kubus yang baru sebagai kubus terbaik. Jika tidak lebih baik, kalkulasikan probabilitas terpilihnya *state* tersebut dengan rumus $\exp(\Delta E/T)$. Jika probabilitas lebih dari 0.9, *state* tersebut dipilih menjadi kubus terbaik. *Threshold* 0.9 dipilih sebab berdasarkan kalkulasi, kami ingin penurunan hanya bisa dilakukan apabila perbedaan skor maksimal 10. Berikut merupakan ilustrasinya dengan sumbu X mewakili temperatur dan sumbu Y mewakili kalkulasi $\exp(\Delta E/T)$.



Gambar 2. Ilustrasi Probabilitas dengan Threshold 0.9 di Berbagai Temperatur

Setelah proses selesai, temperatur akan dikurangi dan iterasi akan diulang kembali. Berikut merupakan potongan kodanya.

```
func SimulatedAnnealing(cube *MagicCube.MagicCube)
(MagicCube.Response, error) {
    bestCube := cube.Copy()
    temperature := 100.0
    indexChange := [][][]int{}
    objectiveFunctions := []int{}
    countLocalOptimum := 0
    deltaEValues := []int{}
    iterations := 0
    cubeStates := [][][][]int{}

    start := time.Now()
    initialScore := bestCube.ObjectiveFunction()
    objectiveFunctions = append(objectiveFunctions, initialScore)
    cubeStates = append(cubeStates, bestCube.Buffer)

    if initialScore != 0 {
        for {
            iterations++

            if temperature <= 0 {
                break
            }

            swapSourceIdx := bestCube.GetRandomIdx()
            swapTargetIdx := bestCube.GetRandomIdx()

            for {
                if swapSourceIdx != swapTargetIdx {
                    break
                }
                swapTargetIdx = bestCube.GetRandomIdx()
            }

            newCube := bestCube.Copy()
            newCube.SwapValues(swapSourceIdx, swapTargetIdx)

            if newCube.ObjectiveFunction() == 0 {
                bestCube = newCube
                objectiveFunctions = append(objectiveFunctions,
bestCube.ObjectiveFunction())
                cubeStates = append(cubeStates, bestCube.Buffer)
                break
            }

            deltaE := newCube.ObjectiveFunction() -
bestCube.ObjectiveFunction()

            if deltaE > 0 {
                deltaEValues = append(deltaEValues, 1)
                bestCube = newCube
                indexChange = append(indexChange,

```

```

    [] [] int{swapSourceIdx[:, swapTargetIdx[:]]}
        objectiveFunctions = append(objectiveFunctions,
bestCube.ObjectiveFunction())
    } else {
        countLocalOptimum++
        deltaEVValues = append(deltaEVValues, deltaE)
        probability := math.Exp(float64(deltaE) / temperature)
        goDown := probability > 0.9

        if goDown {
            indexChange = append(indexChange,
    [] [] int{swapSourceIdx[:, swapTargetIdx[:]]})
            objectiveFunctions = append(objectiveFunctions,
bestCube.ObjectiveFunction())
            bestCube = newCube
        } else {
            indexChange = append(indexChange, [] [] int{{0, 0,
0}, {0, 0, 0}})
        }
    }

    objectiveFunctions = append(objectiveFunctions,
bestCube.ObjectiveFunction())
    cubeStates = append(cubeStates, bestCube.Buffer)
    temperature -= 0.001
}
}

elapsed := time.Since(start)
executionTimeInMS := int(elapsed.Milliseconds())

Result := Response{
    Buffer:           bestCube.Buffer,
    IndexChange:      indexChange,
    ObjectiveFunctions: objectiveFunctions,
    LocalOptimum:     countLocalOptimum,
    Iterations:       iterations,
    DeltaE:           deltaEVValues,
    ExecutionTimeInMS: executionTimeInMS,
}
}

return Result, nil
}

```

2.6. Genetic Algorithm

Algoritma Genetik diimplementasikan sebagai fungsi yang menerima masukan konfigurasi kubus awal, banyaknya populasi awal, dan jumlah iterasi yang dilakukan.

Pertama, algoritma akan menggenerate populasi awal yaitu objek konfigurasi kubus sebanyak masukan. Lalu dalam setiap iterasi akan dilakukan penghitungan fitness function yang kemudian digunakan untuk memilih secara random. 2 kubus yang terpilih akan dimasukkan ke fungsi crossover sebagai parameter untuk menghasilkan 2

kubus baru melalui crossover dan mutation. Proses ini dilakukan hingga jumlah kubus yang dihasilkan sama dengan jumlah kubus di populasi awal. Di akhir iterasi, kubus generasi baru akan menggantikan generasi lama sebagai penghuni populasi. Hal tersebut diulang hingga mencapai jumlah iterasi yang diinginkan.

```
func GeneticAlgorithm(cube *MagicCube.MagicCube, startpopulation int,
iteration int) (MagicCube.Response, error) {

    response := MagicCube.Response{}
    idx := 0
    start := time.Now()
    population := [] (MagicCube.MagicCube){}
    for i := 0; i < startpopulation; i++ {
        cube.Shuffle()
        newcube := MagicCube.MagicCube{}

        newcube.Buffer = make([][][]int, len(cube.Buffer))
        for j := range cube.Buffer {
            newcube.Buffer[j] = make([][]int, len(cube.Buffer[j]))
            for k := range cube.Buffer[j] {
                newcube.Buffer[j][k] = make([]int,
len(cube.Buffer[j][k]))
                copy(newcube.Buffer[j][k], cube.Buffer[j][k])
            }
        }

        score := cube.ObjectiveFunction()
        newcube.SetScore(score)
        population = append(population, newcube)
    }

    for i := 0; i < iteration; i++ {
        newpopulation := [] (MagicCube.MagicCube){}
        listfitness := []float64{}
        listprobability := []float64{}
        totalfitness := float64(0)

        for j := 0; j < len(population); j++ {
            listfitness = append(listfitness,
1.0/float64(population[j].Score))
            totalfitness += 1.0/float64(population[j].Score)

        }
        for j := 0; j < len(listfitness); j++ {
            if j == 0 {
                listprobability = append(listprobability, 100.0 *
(listfitness[j]/totalfitness))
            } else {
                listprobability = append(listprobability,
listprobability[j-1]+ 100.0 * (listfitness[j]/totalfitness))
            }
        }
        for k:=0; k<len(population); k++ {
```

```

        selector := float64(rand.Intn(100))
        selector1 := float64(rand.Intn(100))

        use1 := MagicCube.MagicCube{}
        for j:=0; j < len(listprobability); j++ {
            if selector < listprobability[j] {
                cube.Buffer = population[j].Buffer
                use1.Buffer = cube.Buffer
                break
            }
        }

        use2 := MagicCube.MagicCube{}
        for j:=0; j < len(listprobability); j++ {
            if selector1 < listprobability[j] {
                use2.Buffer = population[j].Buffer
                break
            }
        }

        crossover(cube, use1, use2, &newpopulation)
    }
    population = newpopulation

    totalscore := 0
    score := -1000000
    for j:=0; j< len(population); j++ {
        if score < population[j].Score {
            score = population[j].Score
            idx = j
        }
        totalscore += population[j].Score
    }
    response.CubeStates = append(response.CubeStates,
population[idx].Buffer)
    response.ObjectiveFunctions =
append(response.ObjectiveFunctions, score)
    response.ObjectiveFunctionsMean =
append(response.ObjectiveFunctionsMean, totalscore/len(population))

    if score == 0 {
        break
    }
}

elapsed := time.Since(start)
response.ExecutionTimeInMS = int(elapsed.Milliseconds())

response.Buffer = population[idx].Buffer

return response, nil
}

func crossover(cube *MagicCube.MagicCube, e MagicCube.MagicCube, f
MagicCube.MagicCube, newpopulation *[]MagicCube.MagicCube) {

```

```

var tracknumw [126]int
var tracknumx [126]int

w := make([][][][]int, 5)
for i := range w {
    w[i] = make([][]int, 5)
    for j := range w[i] {
        w[i][j] = make([]int, 5)
    }
}

x := make([][][][]int, 5)
for i := range x {
    x[i] = make([][]int, 5)
    for j := range x[i] {
        x[i][j] = make([]int, 5)
    }
}

cp := rand.Intn(125)
tracknumidx := 0

// Crossover logic
for j := 0; j < 5; j++ {
    for k := 0; k < 5; k++ {
        for l := 0; l < 5; l++ {
            if tracknumidx < cp {
                w[j][k][l] = e.Buffer[j][k][l]
                tracknumw[e.Buffer[j][k][l]]++
                x[j][k][l] = f.Buffer[j][k][l]
                tracknumx[f.Buffer[j][k][l]]++
                tracknumidx++
            } else {
                w[j][k][l] = f.Buffer[j][k][l]
                tracknumw[f.Buffer[j][k][l]]++
                x[j][k][l] = e.Buffer[j][k][l]
                tracknumx[e.Buffer[j][k][l]]++
            }
        }
    }
}

for j := 0; j < 5; j++ {
    for k := 0; k < 5; k++ {
        for l := 0; l < 5; l++ {
            if tracknumw[w[j][k][l]] > 1 {
                for m := 0; m < len(tracknumw); m++ {
                    if tracknumw[m] == 0 {
                        w[j][k][l] = m
                        tracknumw[m]++
                        break
                    }
                }
            }
            if tracknumx[x[j][k][l]] > 1 {
                for m := 0; m < len(tracknumx); m++ {

```

```

        if tracknumx[m] == 0 {
            x[j][k][l] = m
            tracknumx[m]++
            break
        }
    }
}
}

temp := w[2][2][2]
w[2][2][2] = w[4][4][4]
w[4][4][4] = temp

temp = x[2][2][2]
x[2][2][2] = x[4][4][4]
x[4][4][4] = temp

newcube := MagicCube.MagicCube{}
newcube.Buffer = w
cube.Buffer = w
score := cube.ObjectiveFunction()
newcube.SetScore(score)
*newpopulation = append(*newpopulation, newcube)

newcube2 := MagicCube.MagicCube{}
newcube2.Buffer = x
cube.Buffer = x
score = cube.ObjectiveFunction()
newcube2.SetScore(score)
*newpopulation = append(*newpopulation, newcube2)
}

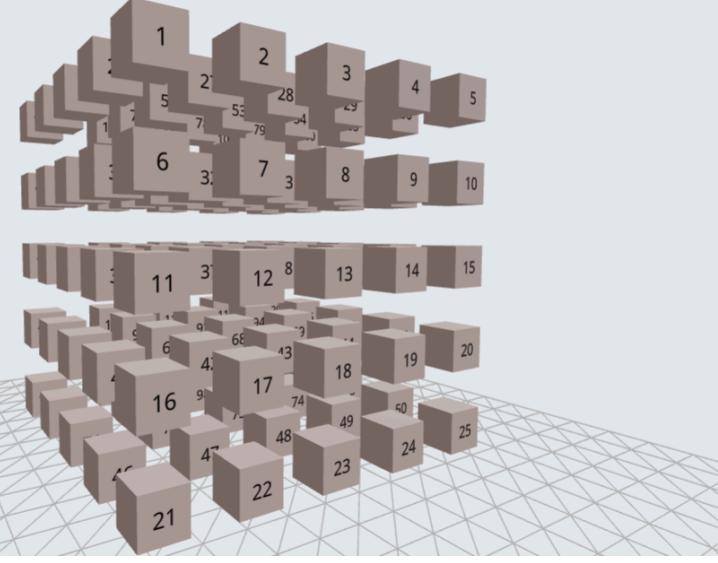
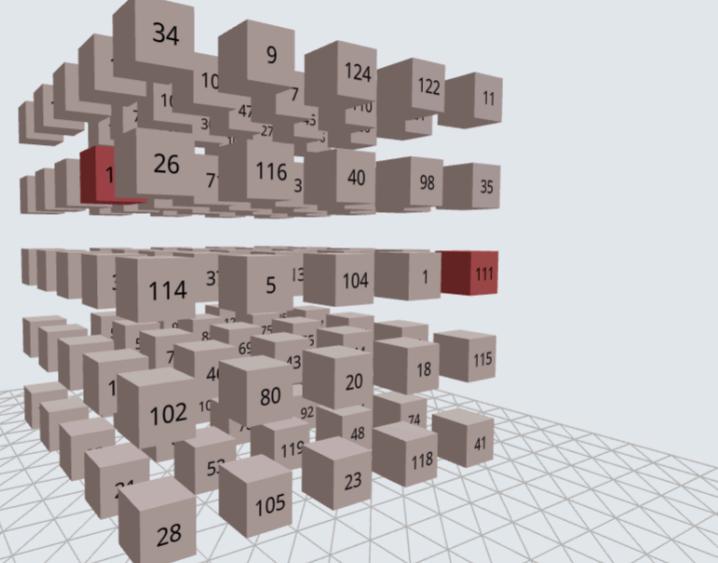
```

3. Hasil Eksperimen dan Analisis

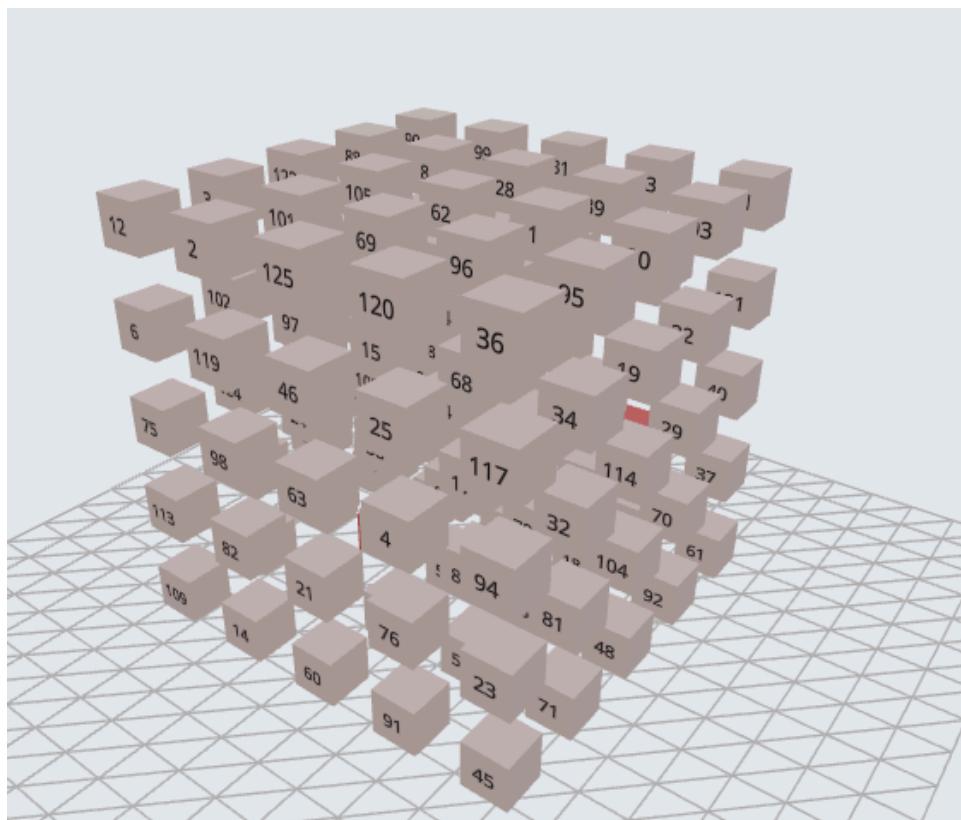
Berikut merupakan data hasil eksperimen untuk setiap algoritma dan analisisnya. Data disajikan dalam tabel.

3.1. Steepest Ascent Hill-Climbing

Tabel 1. Hasil Eksperimen Steepest Ascent Hill-Climbing

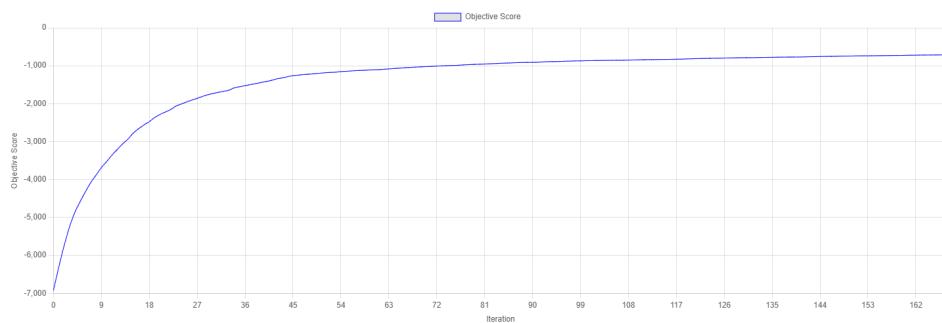
No.	Hasil
	<p><i>Initial state:</i></p>  <p>Skor fungsi objektif: -10.320</p>
1	<p><i>Final state:</i></p>  <p>Skor fungsi objektif: -593</p>
	Grafik objective function:

	<p>Objective Score per Iteration</p> <table border="1"> <thead> <tr> <th>Iteration</th> <th>Objective Score</th> </tr> </thead> <tbody> <tr><td>0</td><td>-10,000</td></tr> <tr><td>8</td><td>-5,000</td></tr> <tr><td>16</td><td>-3,000</td></tr> <tr><td>24</td><td>-2,000</td></tr> <tr><td>32</td><td>-1,800</td></tr> <tr><td>40</td><td>-1,600</td></tr> <tr><td>48</td><td>-1,550</td></tr> <tr><td>56</td><td>-1,520</td></tr> <tr><td>64</td><td>-1,500</td></tr> <tr><td>72</td><td>-1,480</td></tr> <tr><td>80</td><td>-1,470</td></tr> <tr><td>88</td><td>-1,460</td></tr> <tr><td>96</td><td>-1,450</td></tr> <tr><td>104</td><td>-1,440</td></tr> <tr><td>112</td><td>-1,430</td></tr> <tr><td>120</td><td>-1,420</td></tr> <tr><td>128</td><td>-1,410</td></tr> <tr><td>136</td><td>-1,400</td></tr> <tr><td>144</td><td>-1,400</td></tr> </tbody> </table>	Iteration	Objective Score	0	-10,000	8	-5,000	16	-3,000	24	-2,000	32	-1,800	40	-1,600	48	-1,550	56	-1,520	64	-1,500	72	-1,480	80	-1,470	88	-1,460	96	-1,450	104	-1,440	112	-1,430	120	-1,420	128	-1,410	136	-1,400	144	-1,400
Iteration	Objective Score																																								
0	-10,000																																								
8	-5,000																																								
16	-3,000																																								
24	-2,000																																								
32	-1,800																																								
40	-1,600																																								
48	-1,550																																								
56	-1,520																																								
64	-1,500																																								
72	-1,480																																								
80	-1,470																																								
88	-1,460																																								
96	-1,450																																								
104	-1,440																																								
112	-1,430																																								
120	-1,420																																								
128	-1,410																																								
136	-1,400																																								
144	-1,400																																								
	<table border="1"> <tr> <td>Iterasi</td> <td>149</td> </tr> <tr> <td>Durasi</td> <td>794 ms</td> </tr> </table>	Iterasi	149	Durasi	794 ms																																				
Iterasi	149																																								
Durasi	794 ms																																								
2	<p><i>Initial state:</i></p> <p><i>Final state:</i></p> <p>Skor fungsi objektif: -6922</p>																																								



Skor fungsi objektif: -692

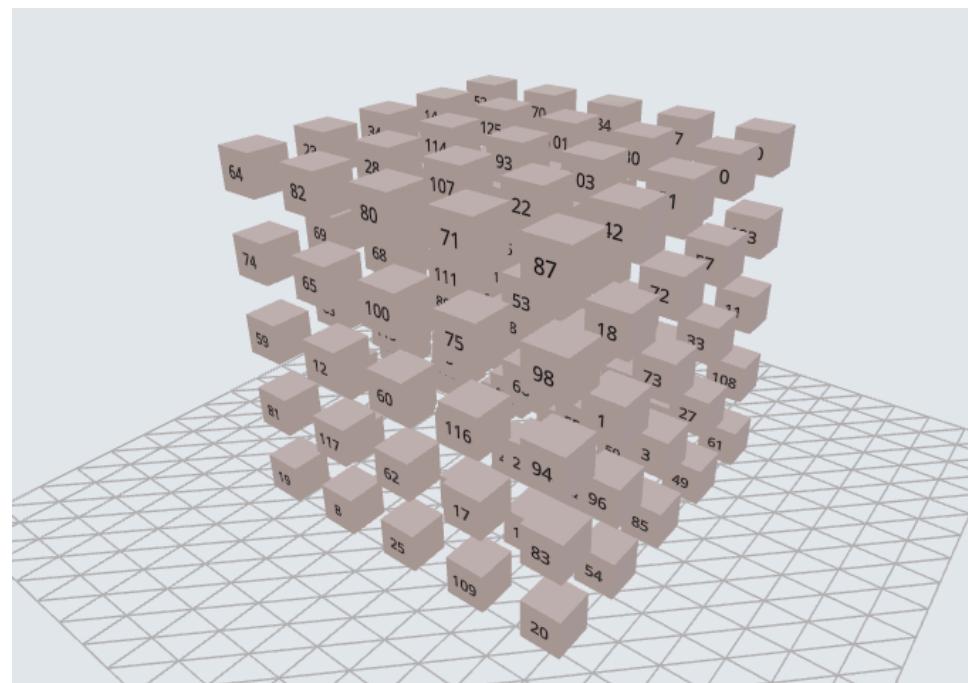
Grafik *objective function*:



Iterasi 168

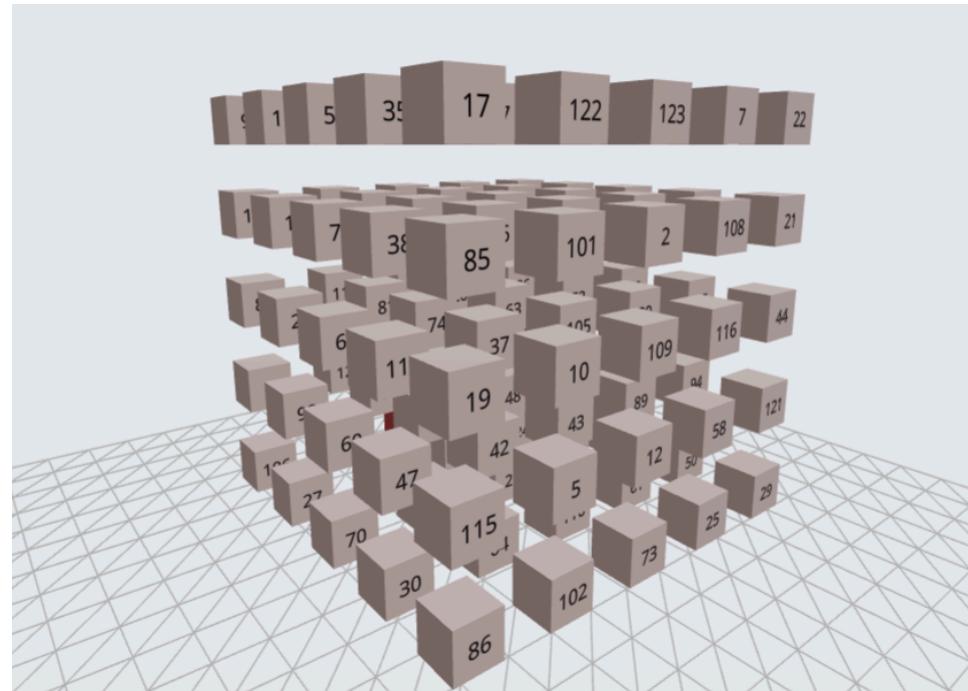
Durasi 938 ms

3 *Initial state:*



Skor Fungsi Objektif: -6.407

Final state:



Skor Fungsi Objektif: -800

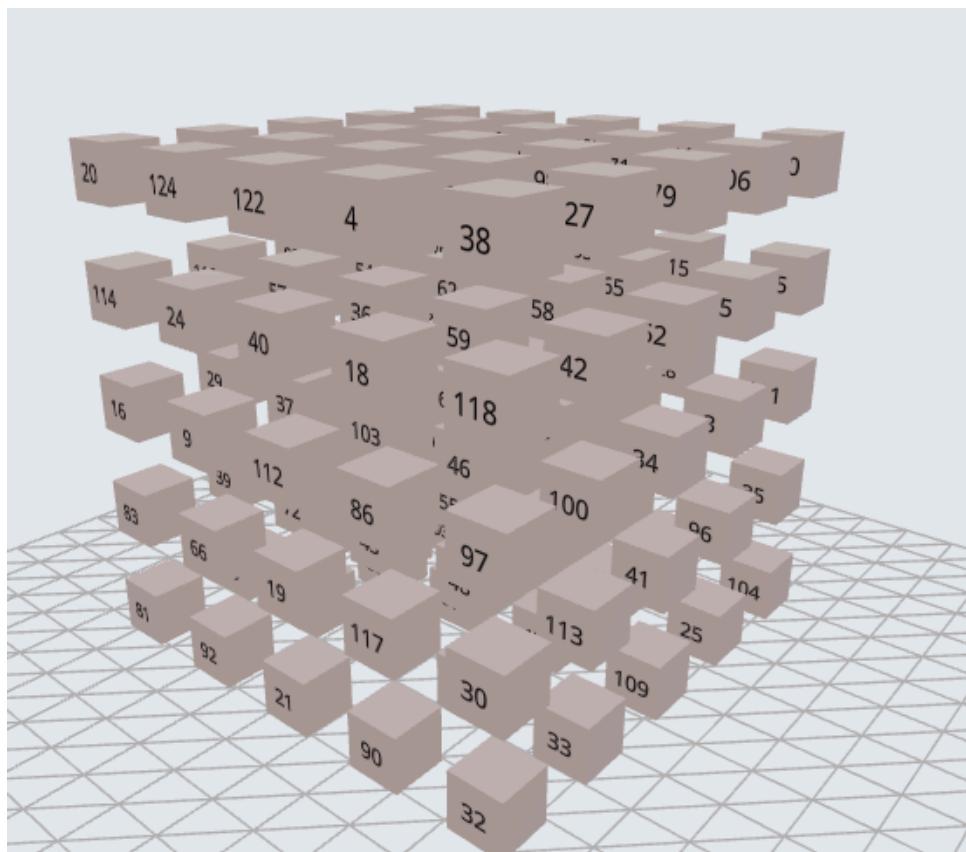
Grafik *objective function*:

	<p>Objective Score</p> <p>Iteration</p>
Iterasi	129
Durasi	730 ms

3.2. Hill-Climbing with Sideways Move

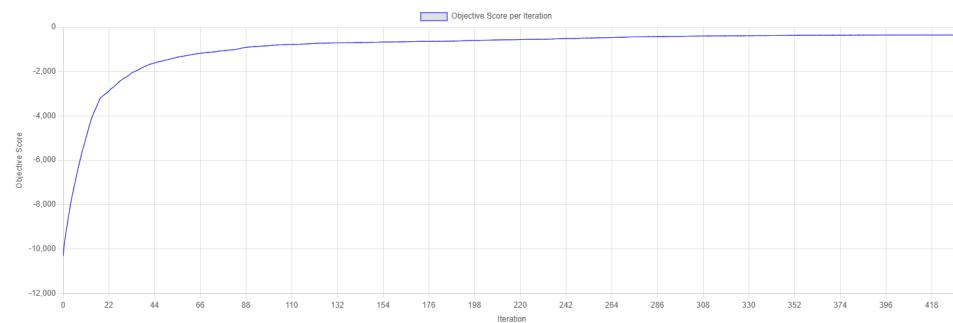
Tabel 2. Hasil Eksperimen Hill-Climbing with Sideways Move

No.	Hasil
1	<p><i>Initial state:</i></p> <p>Skor Fungsi Objektif: -10320</p>
	<p><i>Final state:</i></p>



Skor Fungsi Objektif: -351

Grafik *objective function*:



Sideways maksimum

40

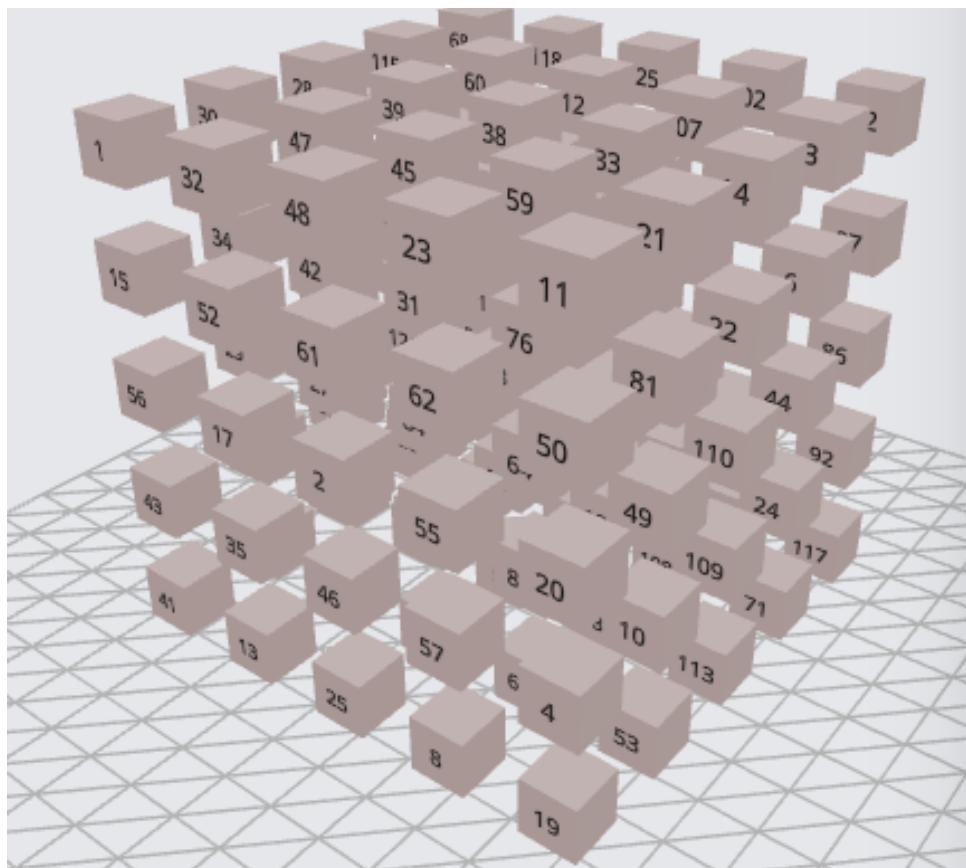
Iterasi

431

Durasi

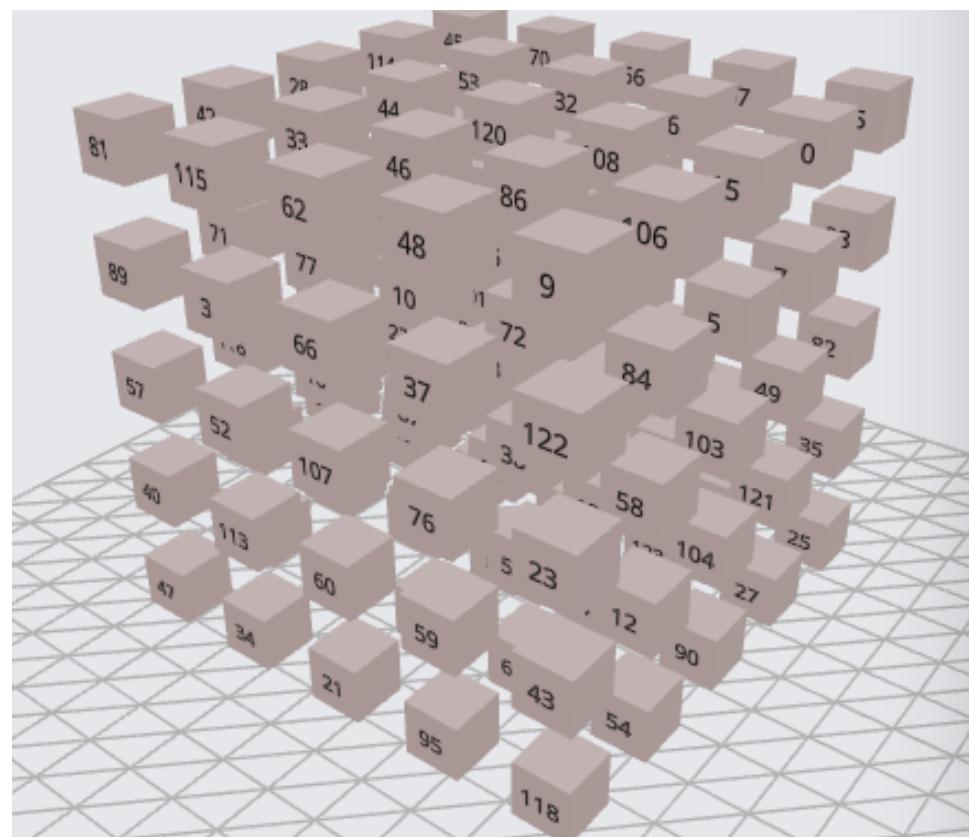
2445 ms

2 *Initial state:*



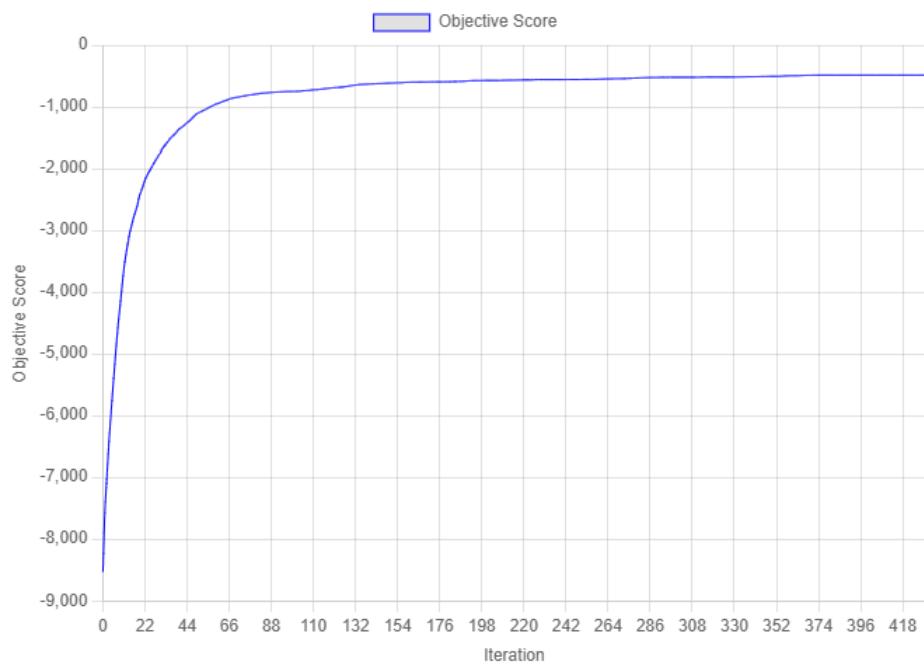
Skor Fungsi Objektif: -8531

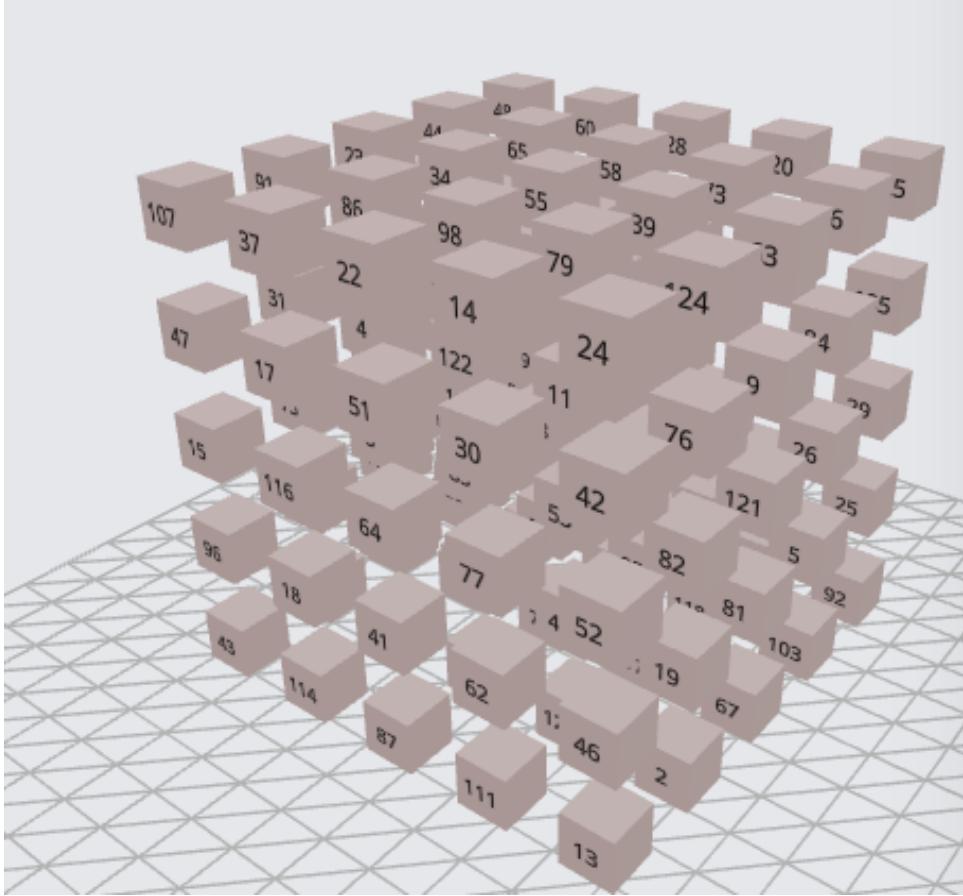
Final state:

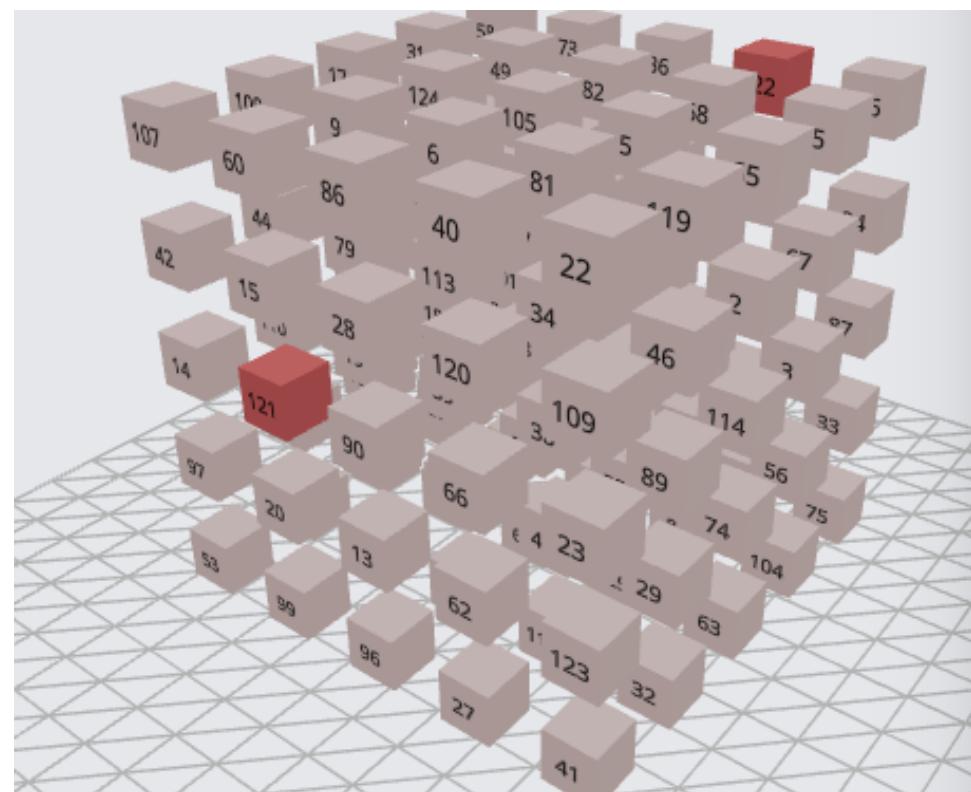


Skor Fungsi Objektif: -486

Grafik *objective function*:

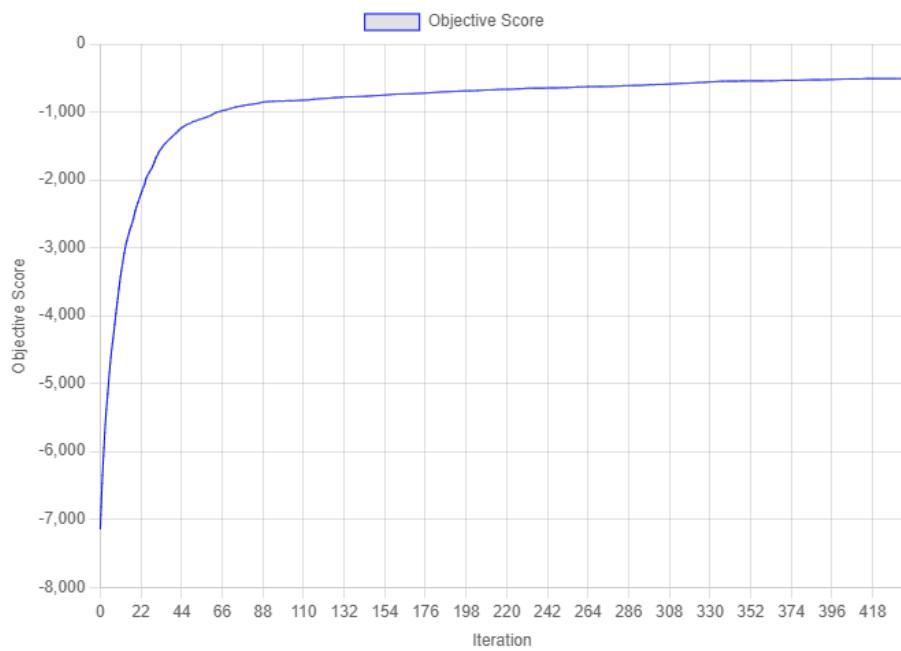


	<i>Sideways maksimum</i>	60
	Iterasi	430
	Durasi	2286 ms
<i>Initial state:</i>		
 Skor Fungsi Objektif: -7.147		
<i>Final state:</i>		



Skor Fungsi Objektif: -508

Grafik *objective function*:

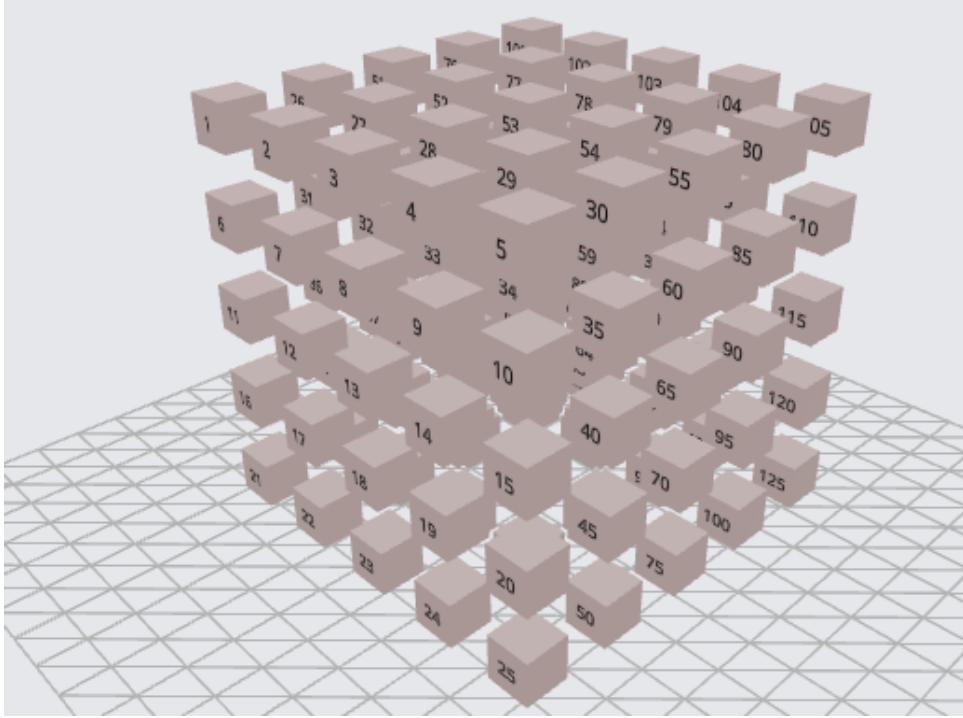


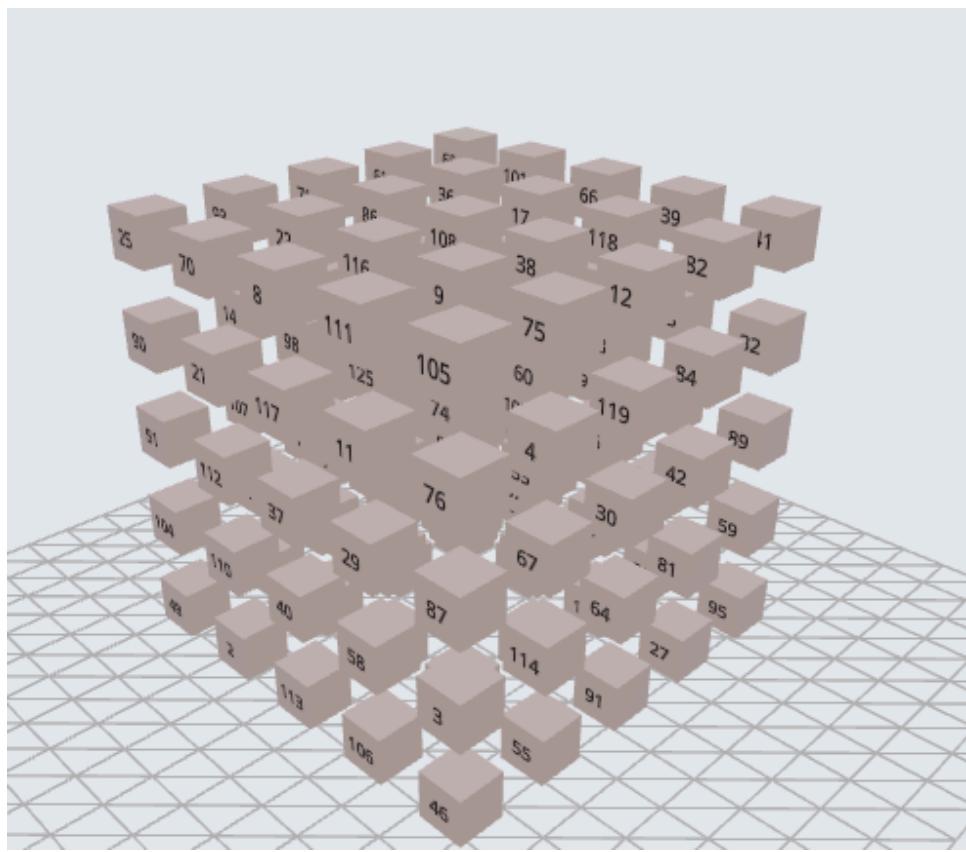
Sideways maksimum 150

	Iterasi	435
	Durasi	3058 ms

3.3. Random Restart Hill-Climbing

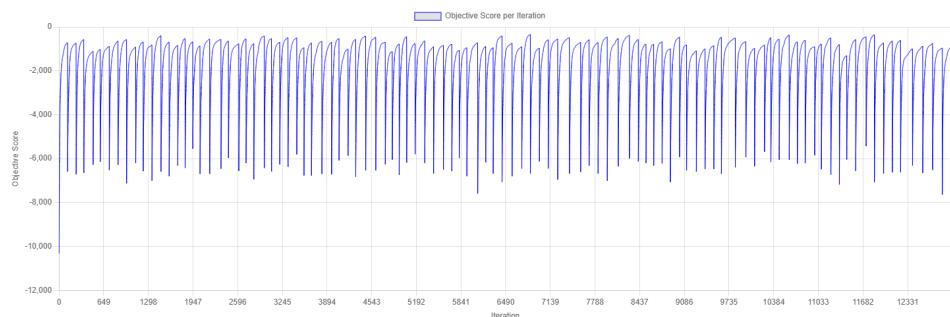
Tabel 3. Hasil Eksperimen Random Restart Hill-Climbing

No.	Hasil	
	<i>Restart maksimum</i>	100
1	<p><i>Initial state:</i></p>  <p>Skor Fungsi Objektif: -10.320</p>	
	<i>Final state:</i>	



Skor Fungsi Objektif = -907

Grafik *objective function*:

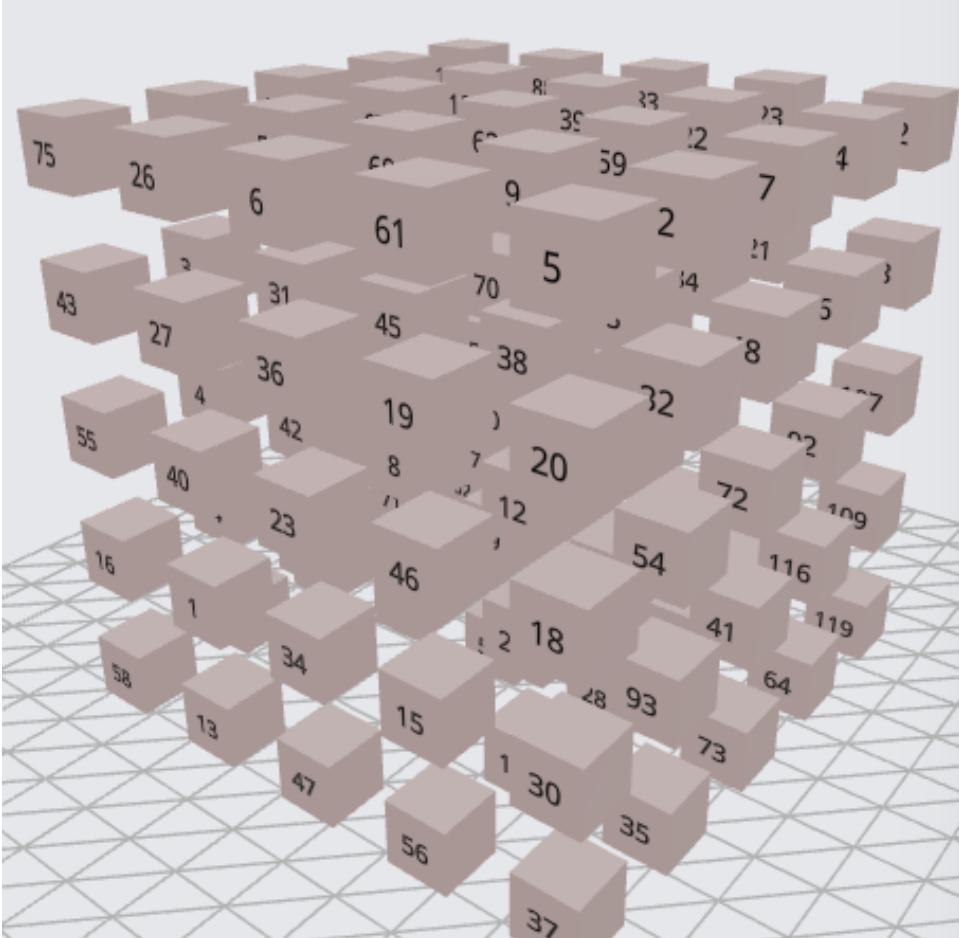


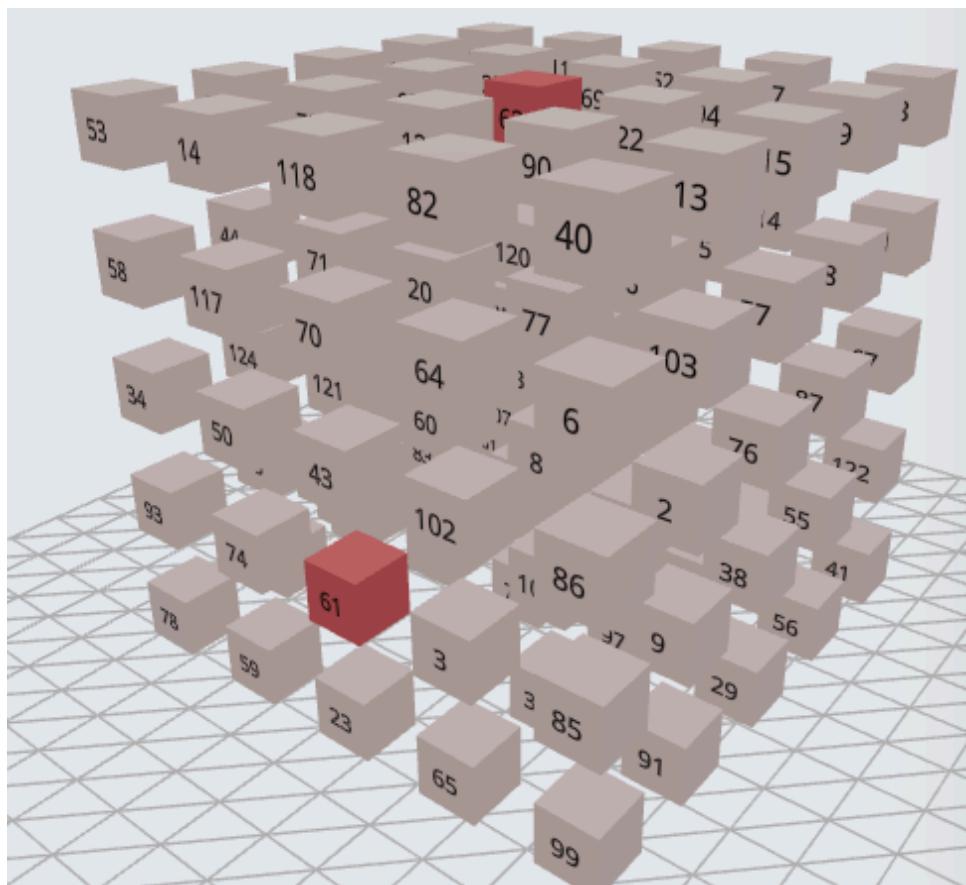
Iterasi per *restart*:

125, 123, 112, 135, 103, 131, 128, 124, 133, 109, 127, 131, 118, 126, 107, 111, 105, 141, 164, 108, 151, 105, 113, 152, 105, 120, 120, 127, 105, 109, 151, 141, 108, 131, 112, 145, 146, 138, 102, 100, 109, 123, 140, 128, 143, 122, 108, 111, 158, 116, 108, 129, 146, 141, 124, 132, 128, 138, 170, 163, 123, 130, 134, 159, 163, 129, 112, 112, 121, 121, 133, 102, 142, 127, 127, 110, 202, 154, 126, 144, 93, 121, 145, 119, 114, 138, 97, 138, 123, 108, 130, 151, 120, 134, 127, 120, 173, 149, 143, 141, 138

Banyak *restart*

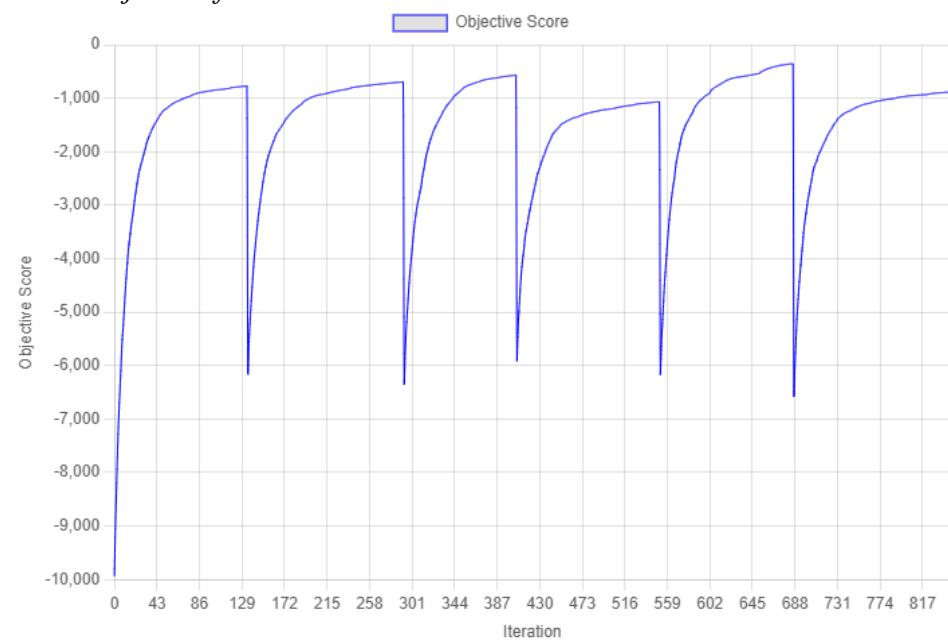
100

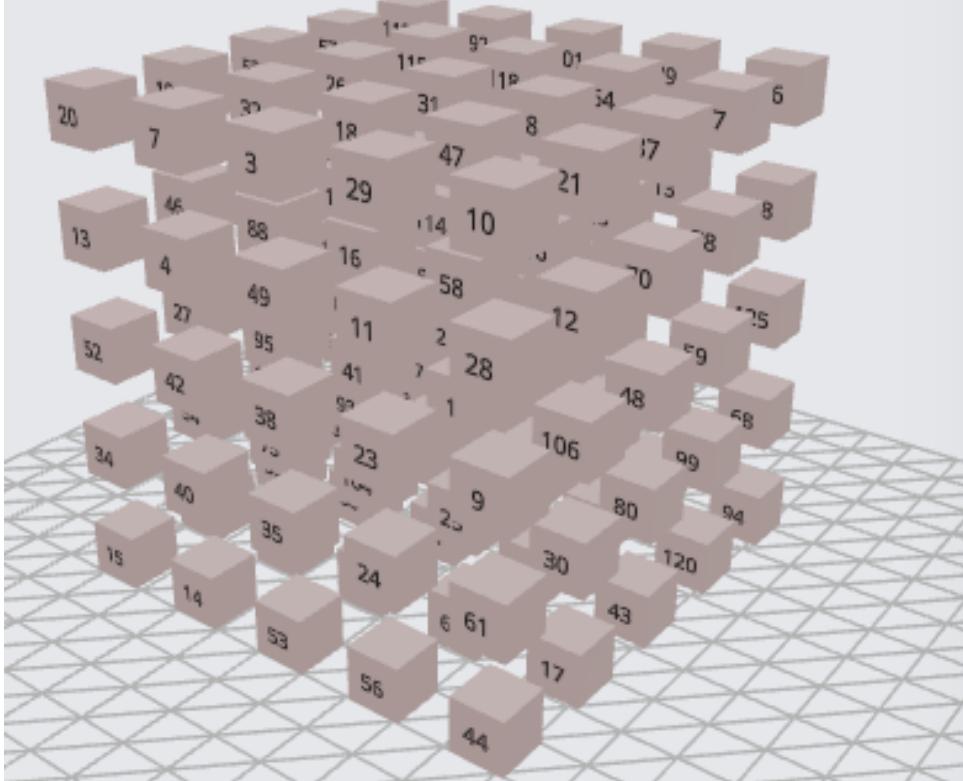
Durasi	72319 ms
Restart maksimum	5
<i>Initial state:</i>	
 <p>Skor Fungsi Objektif: -9.928</p>	

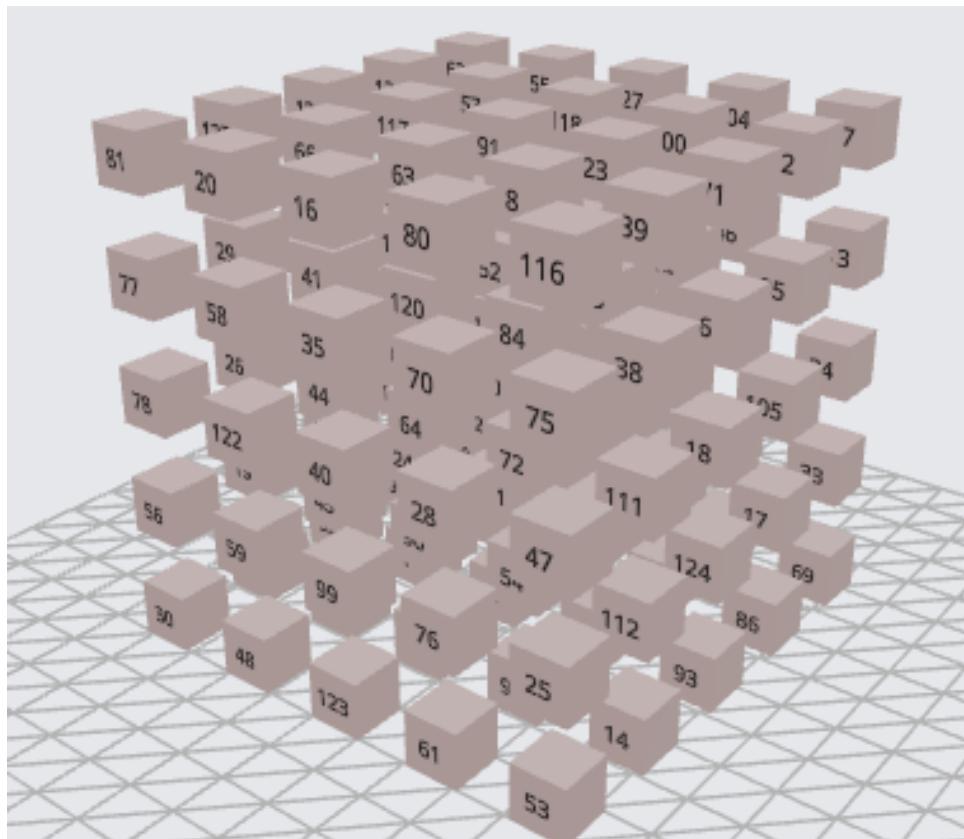


Skor Fungsi Objektif: -882

Grafik *objective function*:

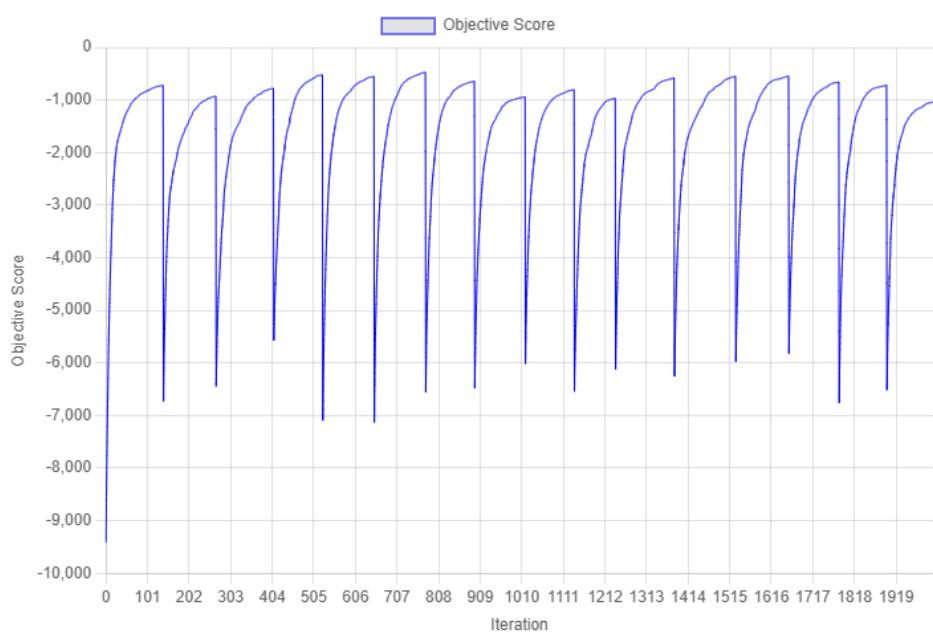


	Iterasi per <i>restart</i> : 134, 158, 114, 145, 135, 159
	Banyak <i>restart</i> 5
	Durasi 4145 ms
	<i>Restart maksimum</i> 15
3	<i>Initial state</i> :
	 <p style="text-align: center;">Skor Fungsi Objektif: -9.416</p>
	<i>Final state</i> :



Skor Fungsi Objektif: -1.040

Grafik *objective function*:

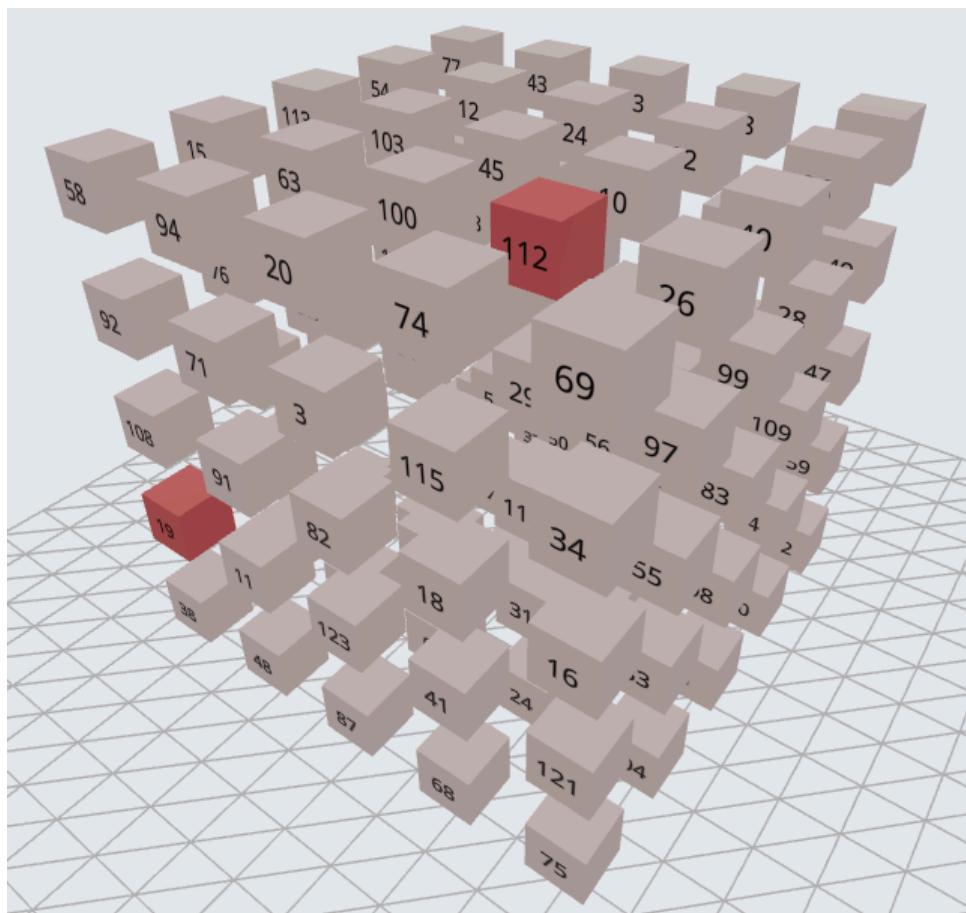


	Gambar banyak iterasi per <i>restart</i> : 139, 128, 140, 119, 125, 125, 119, 123, 119, 100, 143, 149, 129, 122, 116, 113
Banyak <i>restart</i>	15
Durasi	9928 ms

3.4. Stochastic Hill-Climbing

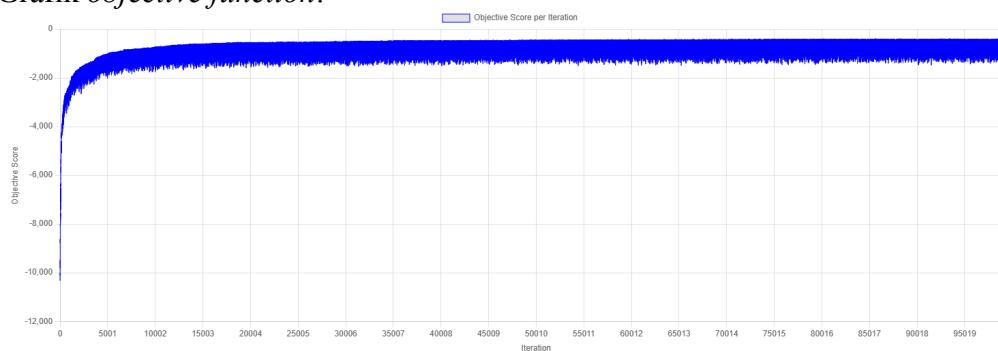
Tabel 4. Hasil Eksperimen Stochastic Hill-Climbing

No.	Hasil
1	<p>Initial state:</p> <p>Skor Fungsi Objektif: -10.320</p> <p>Final state:</p>



Skor Fungsi Objektif: -1.147

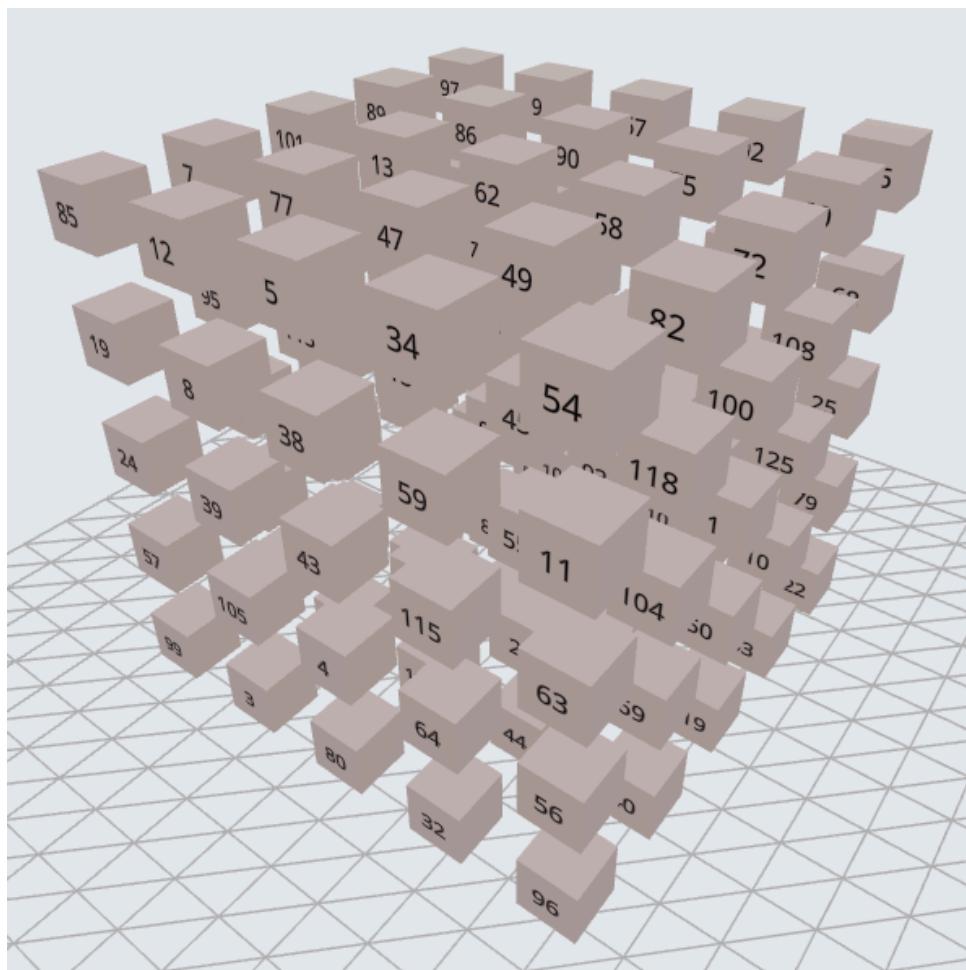
Grafik *objective function*:



Iterasi 100.000

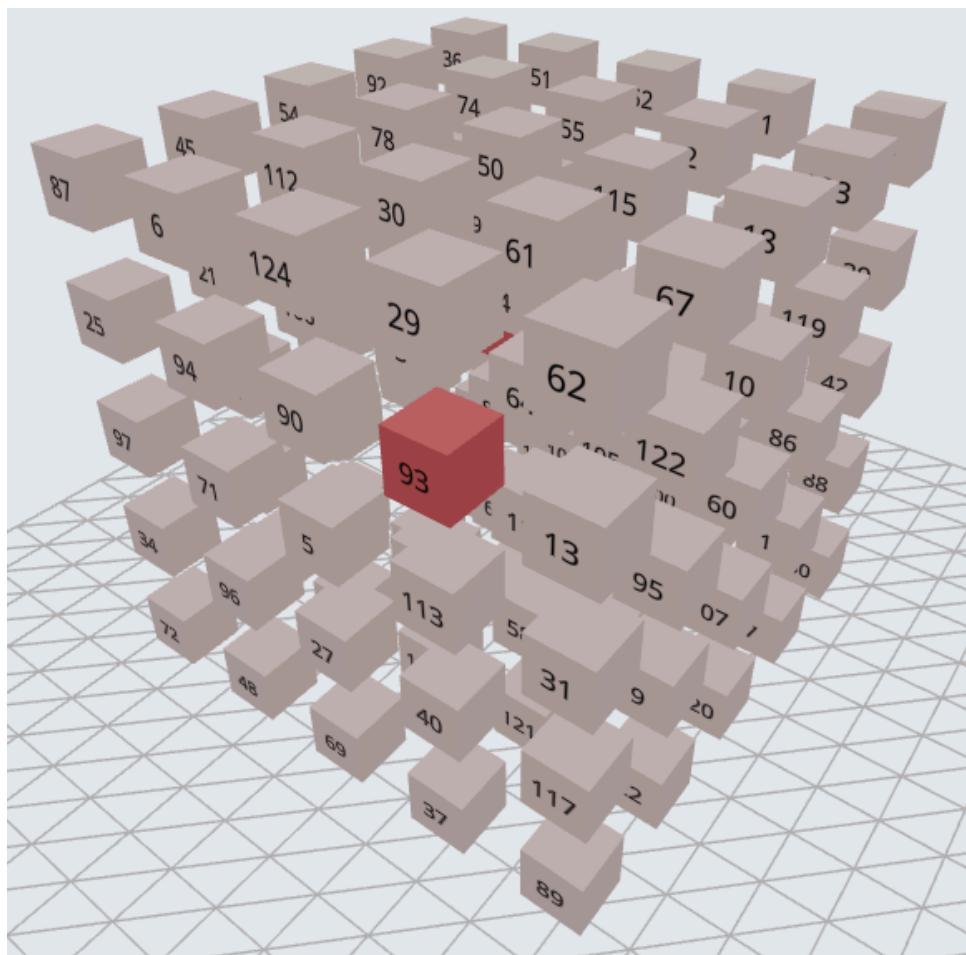
Durasi 382 ms

2 *Initial state:*

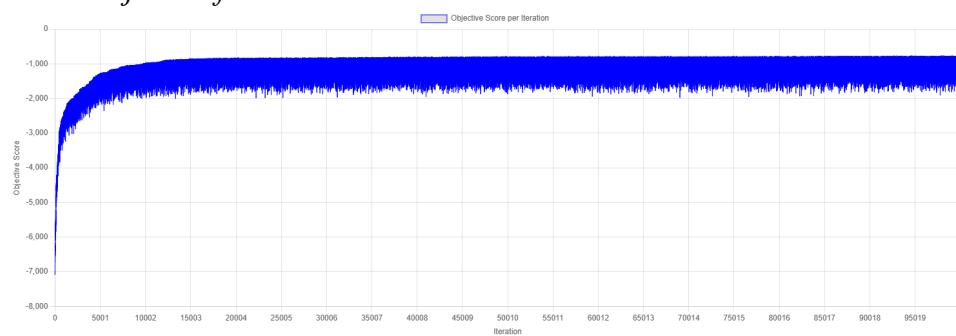


Skor Fungsi Objektif: -7.008

Final state:



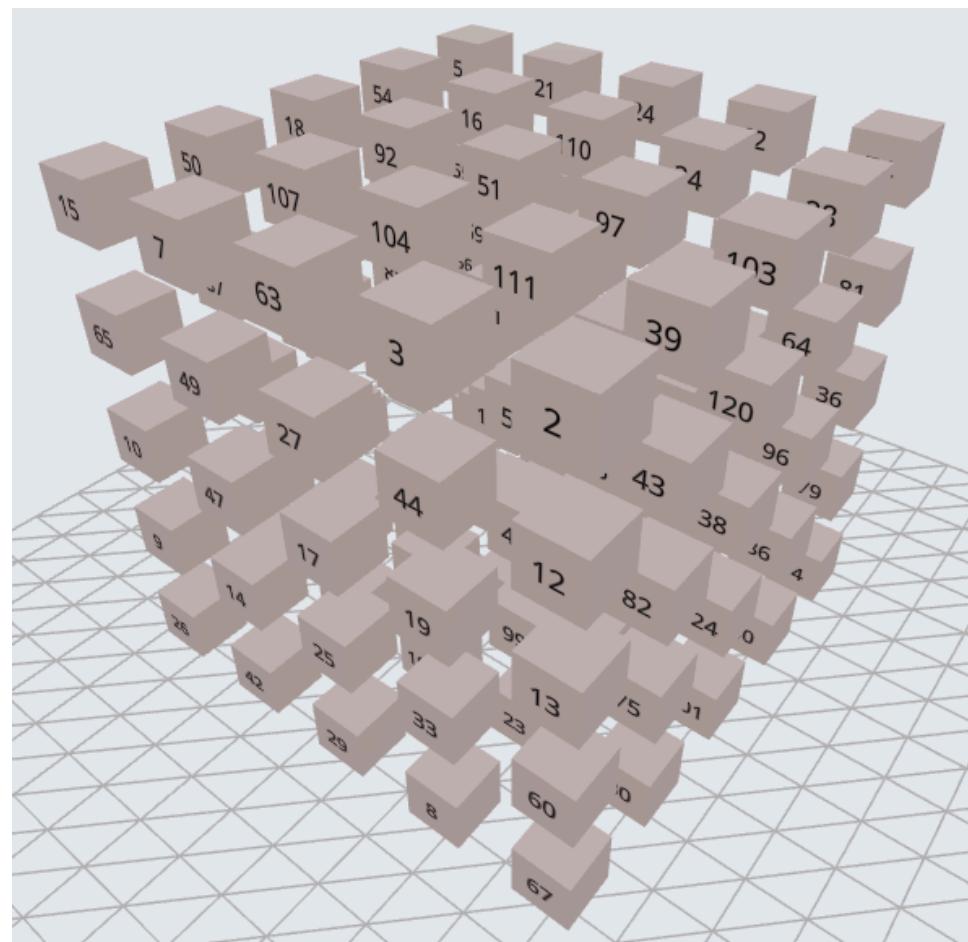
Grafik *objective function*:



Iterasi 100.000

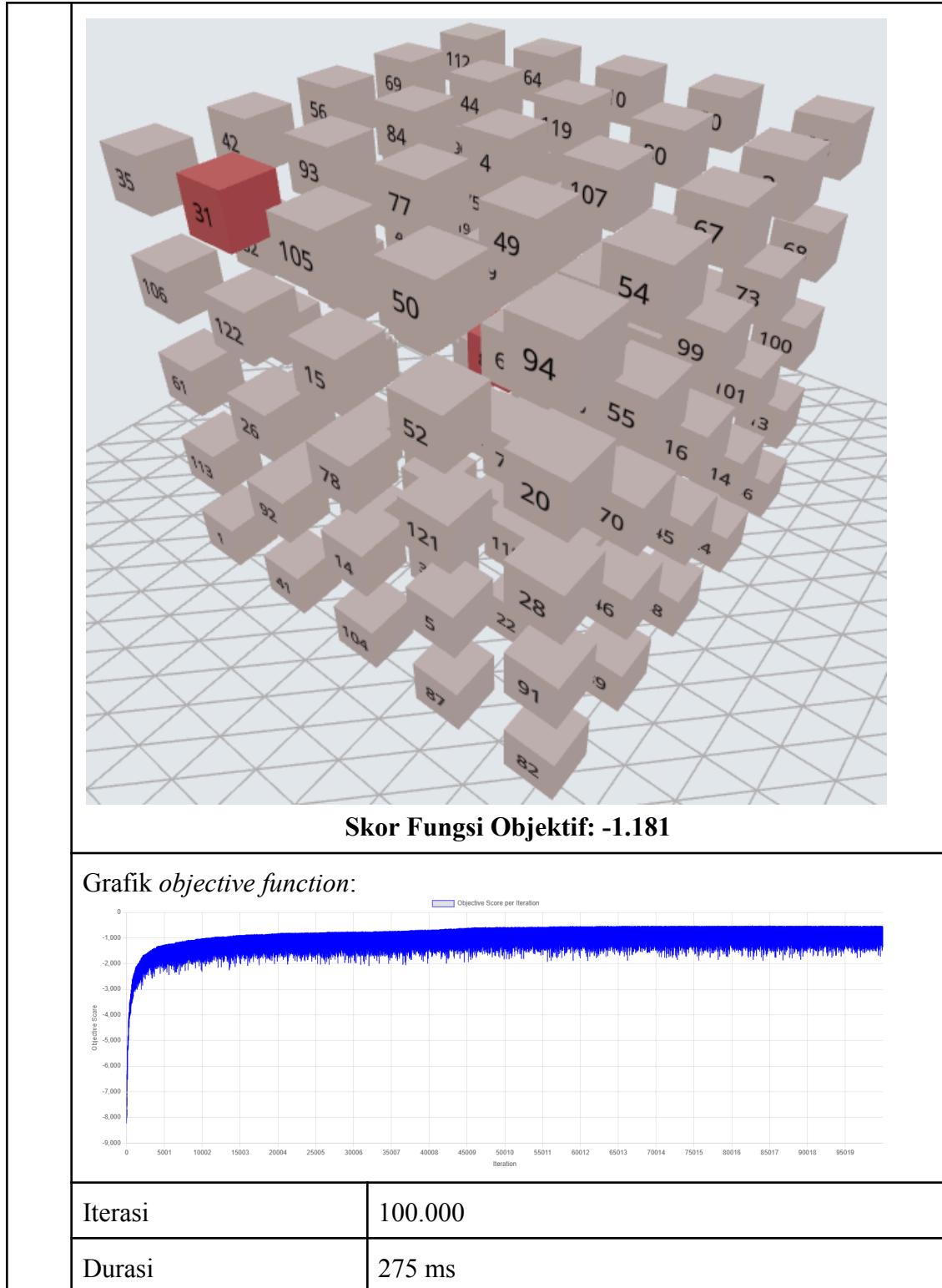
Durasi 284 ms

3 *Initial state:*



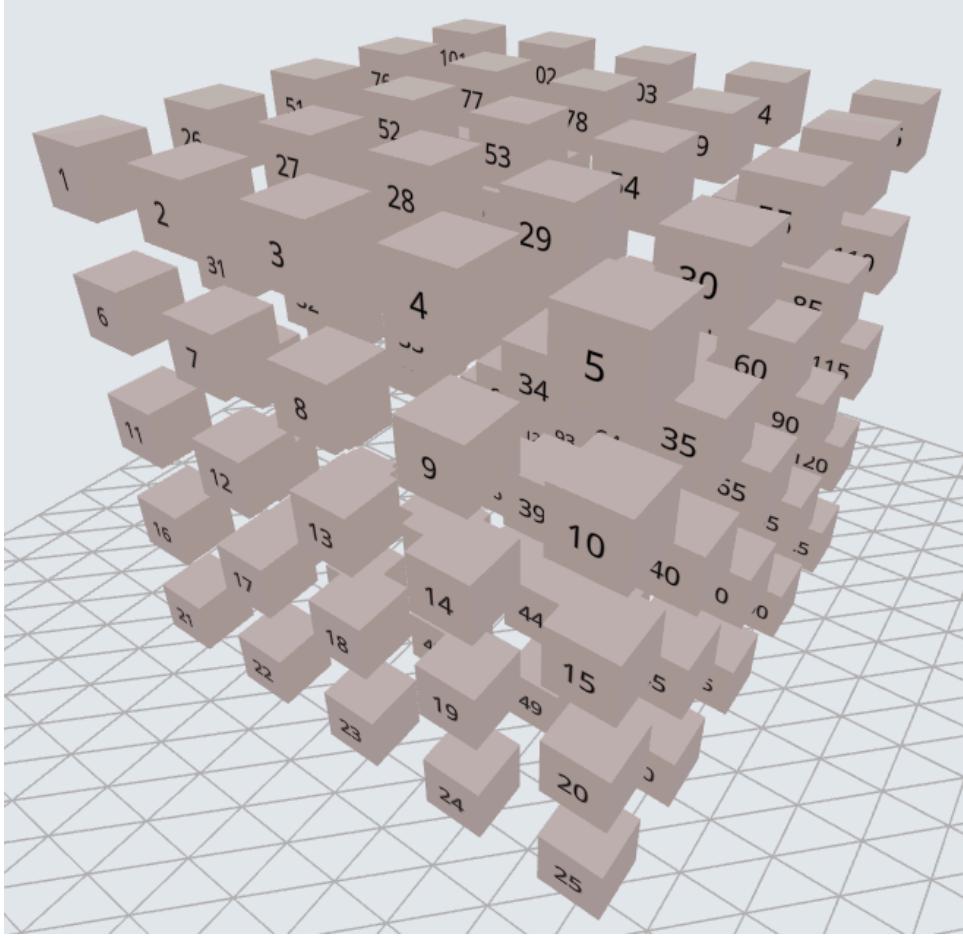
Skor Fungsi Objektif: -8.105

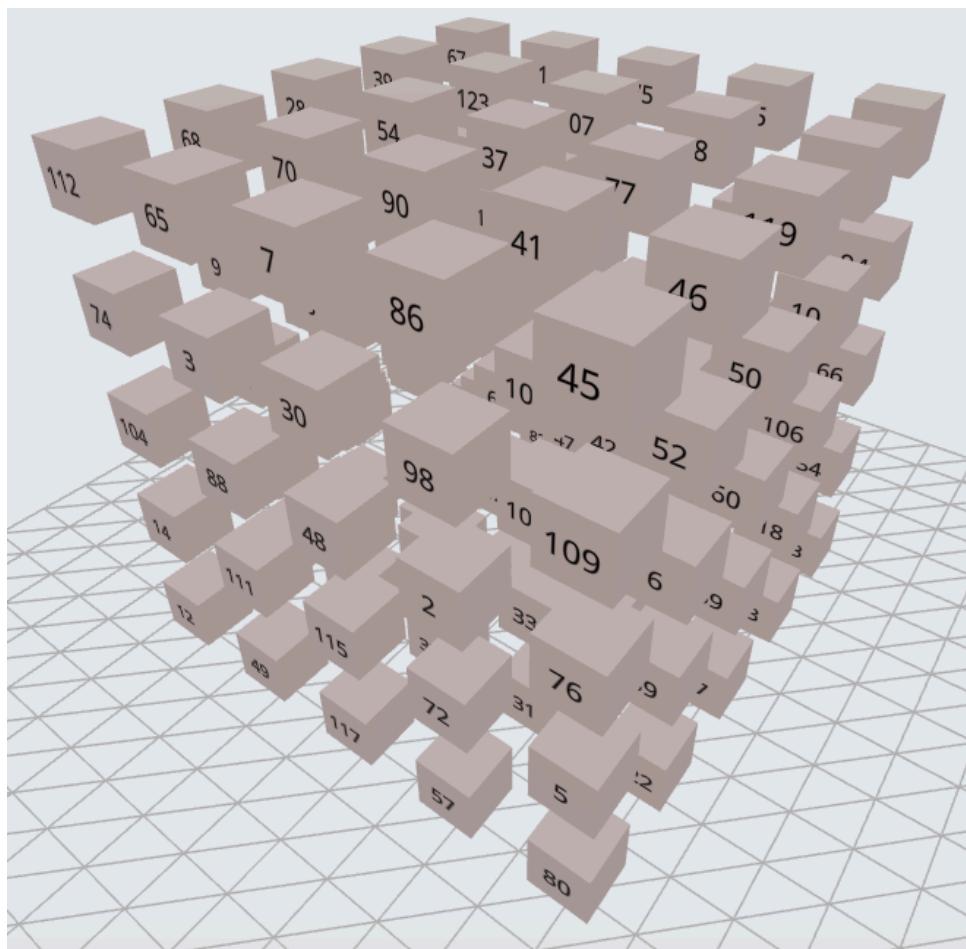
Final state:



3.5. Simulated Annealing

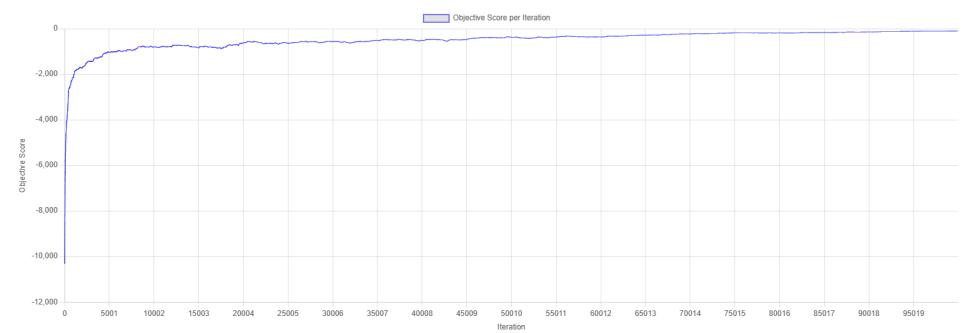
Tabel 5. Hasil Eksperimen Simulated Annealing

No.	Hasil
1	<p><i>Initial state:</i></p>  <p>Skor fungsi objektif: -10.320</p>
	<i>Final state</i>



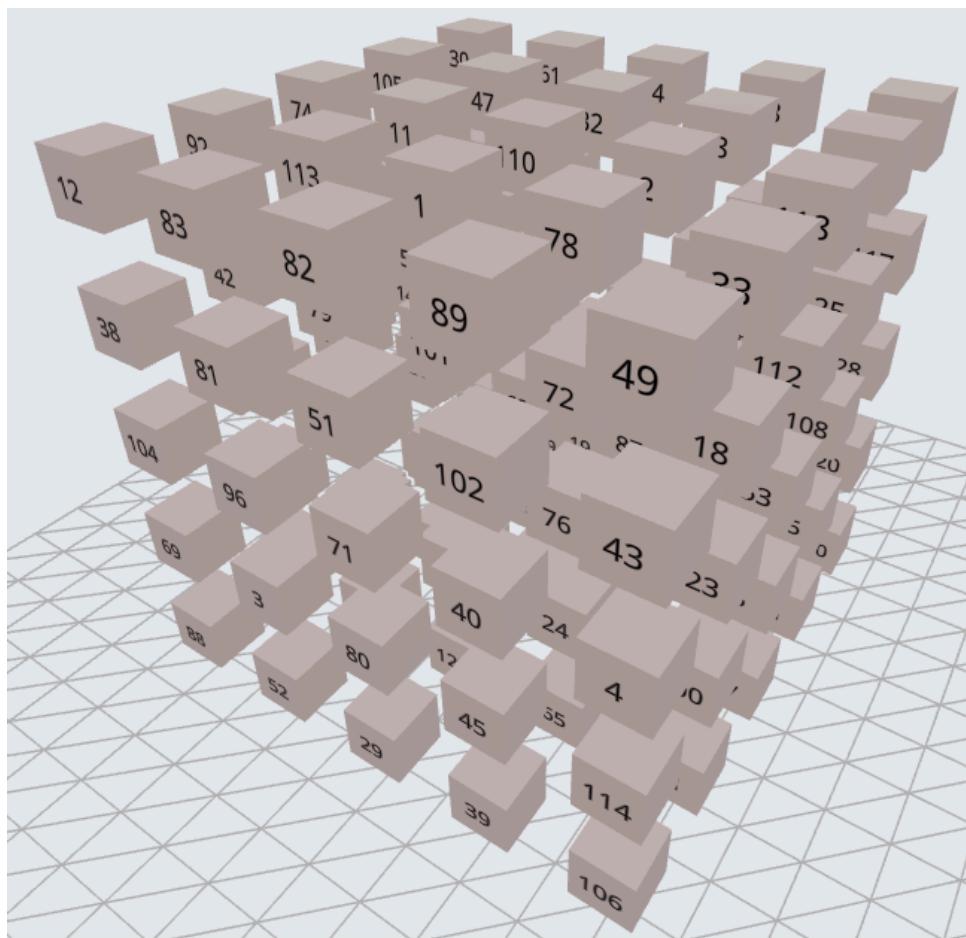
Skor fungsi objektif: -102

Grafik *objective function*:

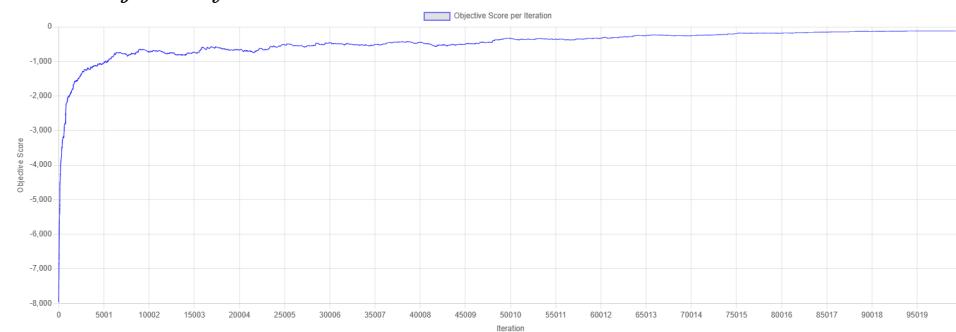


Grafik $\exp(\Delta E/T)$:

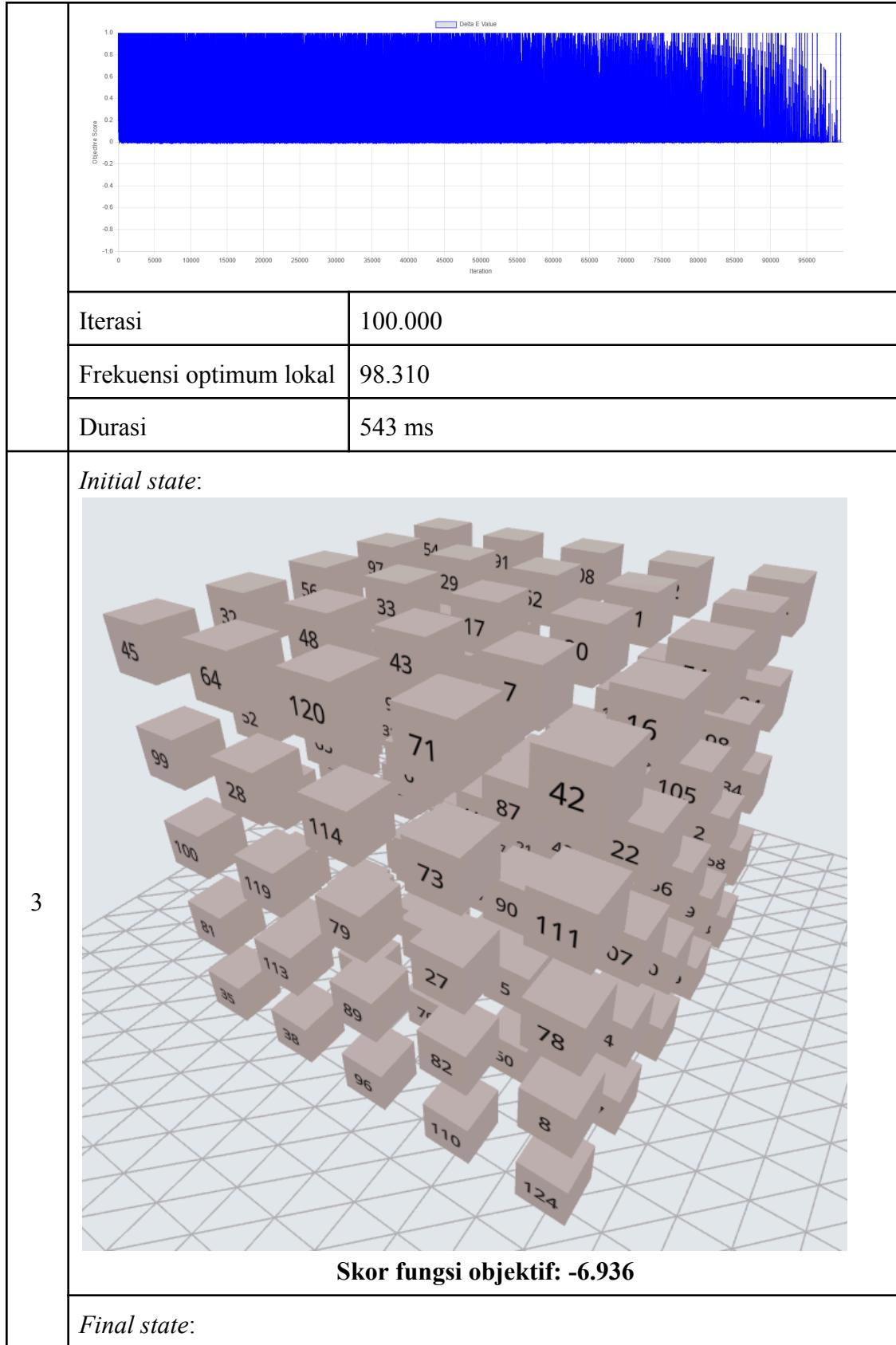
2	Iterasi 100.000
	Frekuensi optimum lokal 98.282
	Durasi 573 ms
2	<p><i>Initial state:</i></p> <p>Skor fungsi objektif: -7.971</p> <p><i>Final state:</i></p>

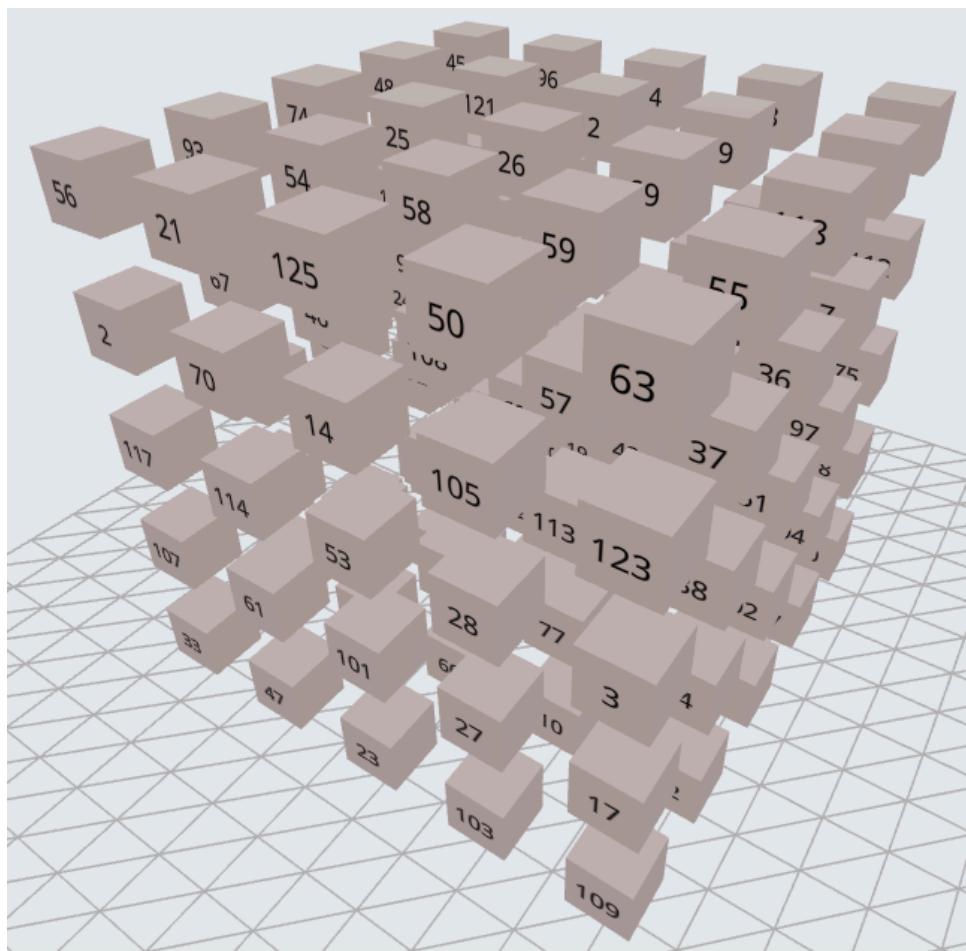


Grafik *objective function*:

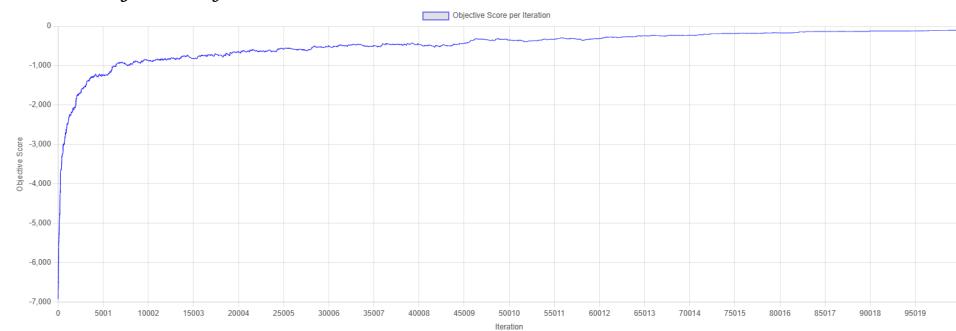


Grafik $\exp(\Delta E/T)$:

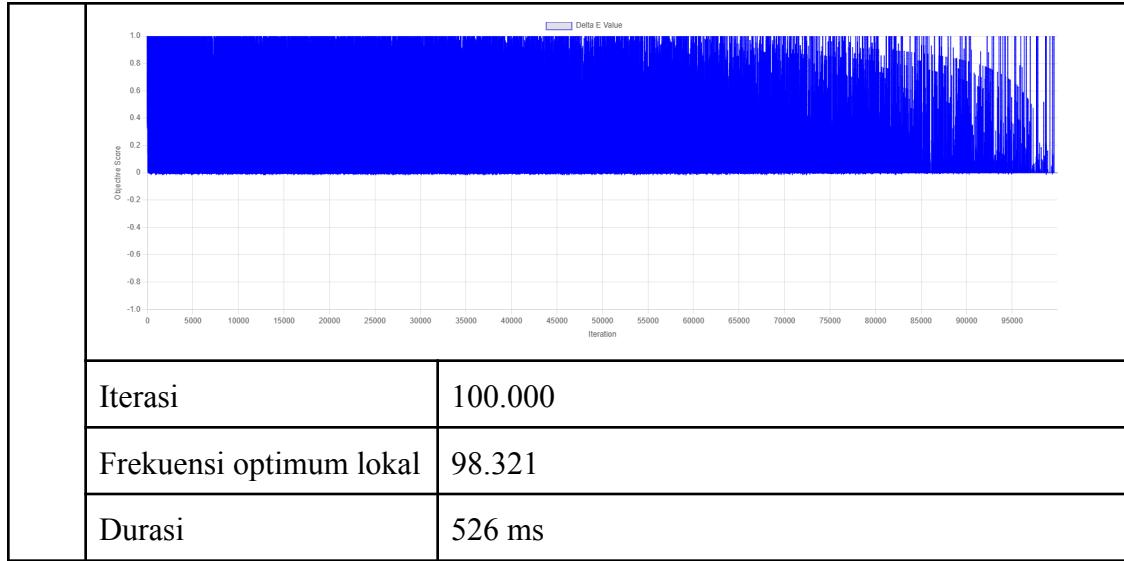




Grafik *objective function*:



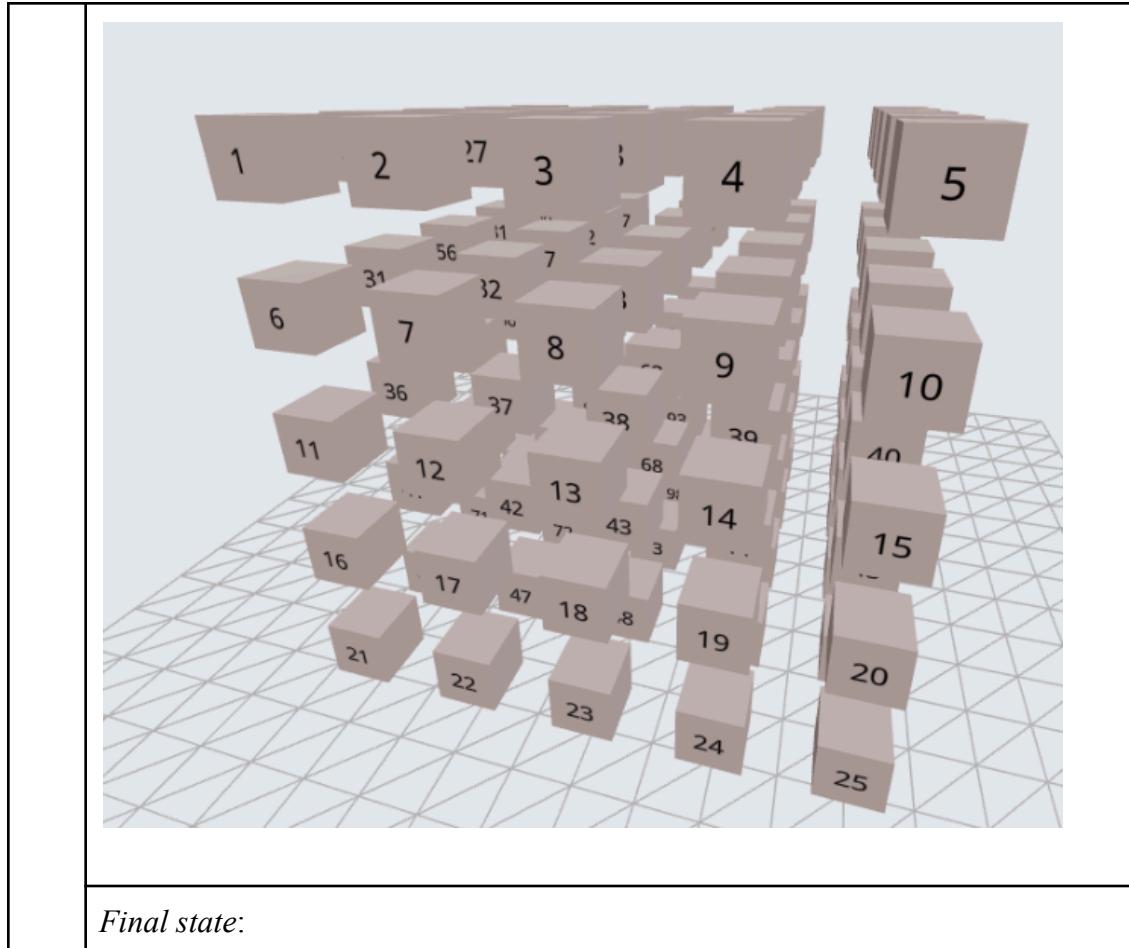
Grafik $\exp(\Delta E/T)$:

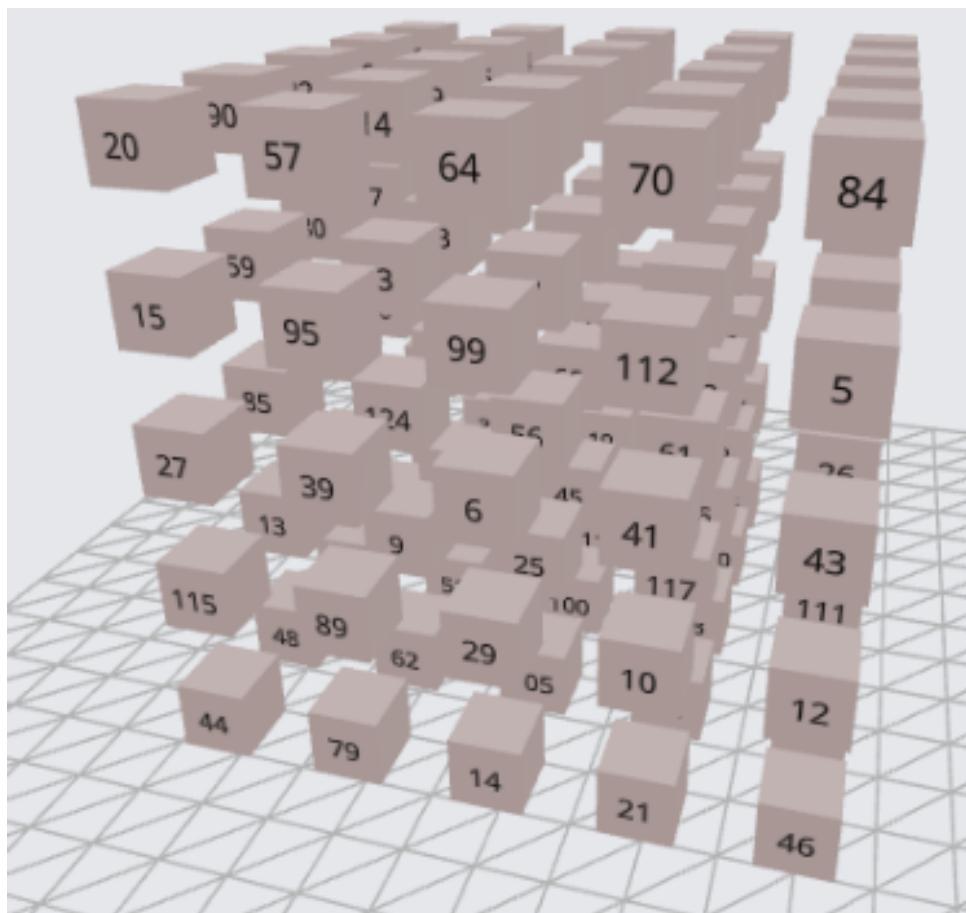


3.6. Genetic Algorithm

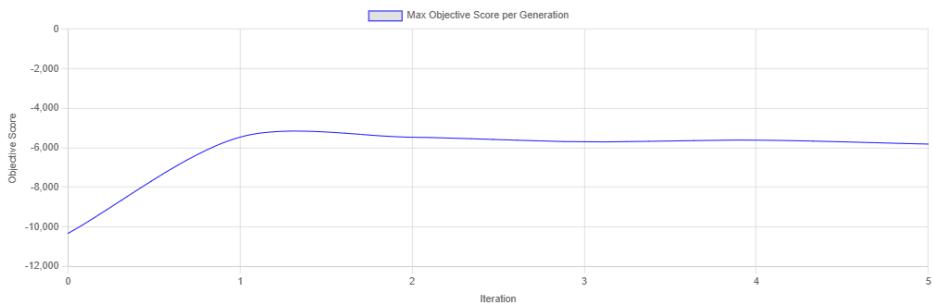
Tabel 6. Hasil Eksperimen Genetic Algorithm

No.	Hasil	
1	Jumlah populasi	10.000
	Banyak iterasi	5
	<i>Initial state:</i>	





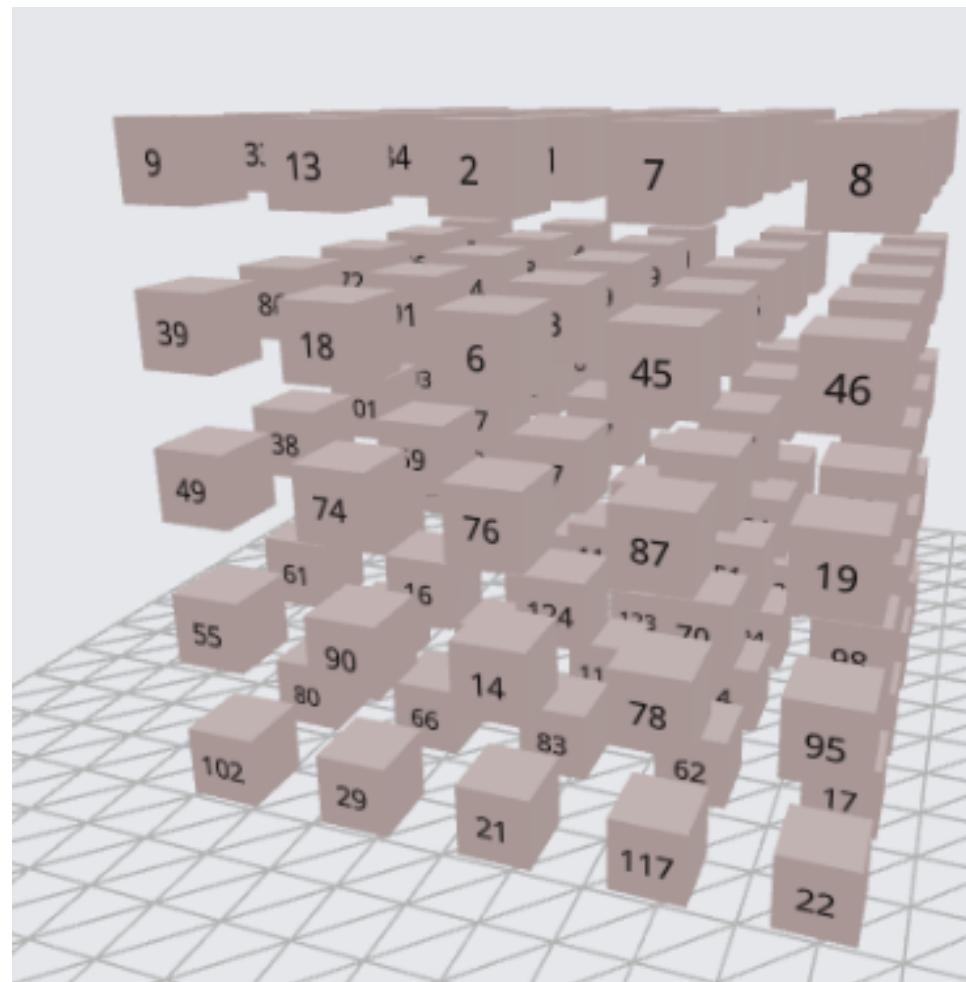
Grafik *objective function* maksimal per iterasi:



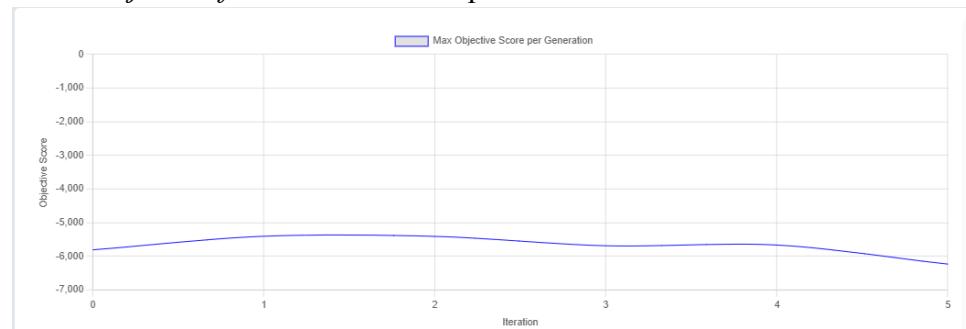
Grafik *objective function* rata-rata per iterasi:

	<p>Objective Score Mean per Generation</p> <p>Iteration</p>				
	<table border="1"> <tr> <td>Skor final</td><td>-5.804</td></tr> <tr> <td>Durasi</td><td>549 ms</td></tr> </table>	Skor final	-5.804	Durasi	549 ms
Skor final	-5.804				
Durasi	549 ms				
	<table border="1"> <tr> <td>Jumlah populasi</td><td>5.000</td></tr> <tr> <td>Banyak iterasi</td><td>5</td></tr> </table>	Jumlah populasi	5.000	Banyak iterasi	5
Jumlah populasi	5.000				
Banyak iterasi	5				
2	<p><i>Initial state:</i></p> <p>Score : -5804</p>				

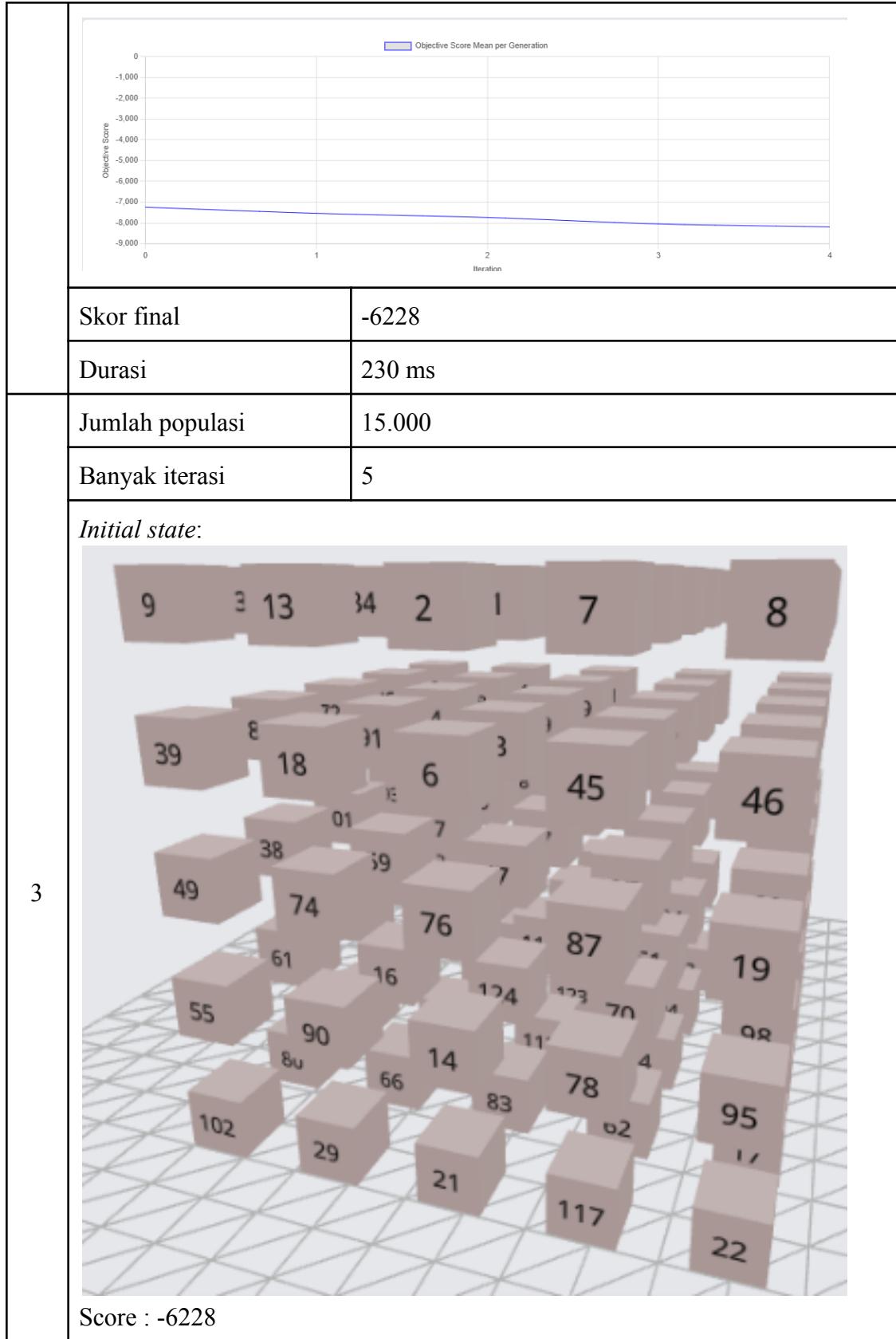
Final state:



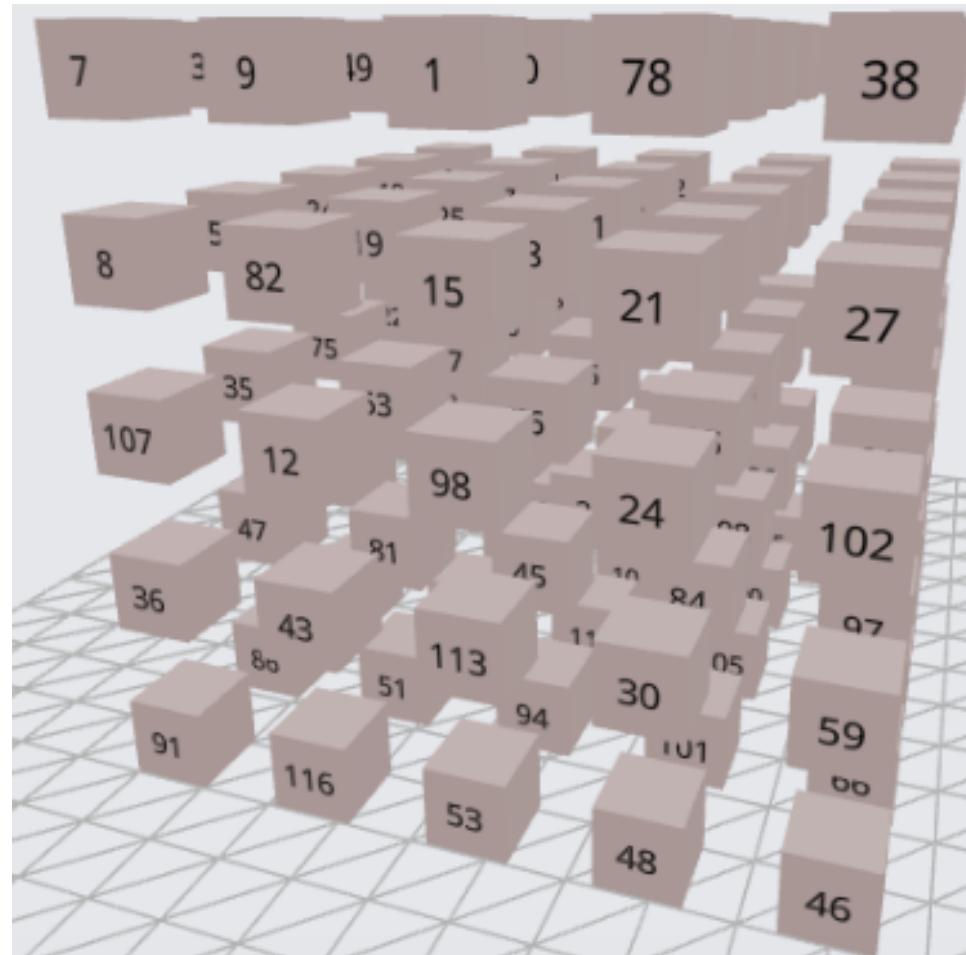
Grafik *objective function* maksimal per iterasi:



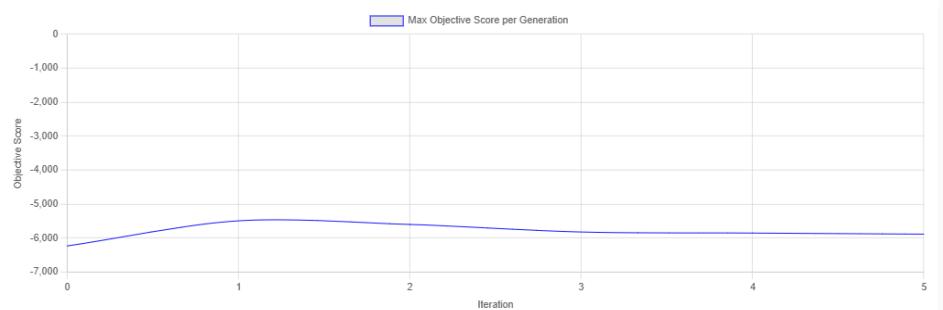
Grafik *objective function* rata-rata per iterasi:



Final state:

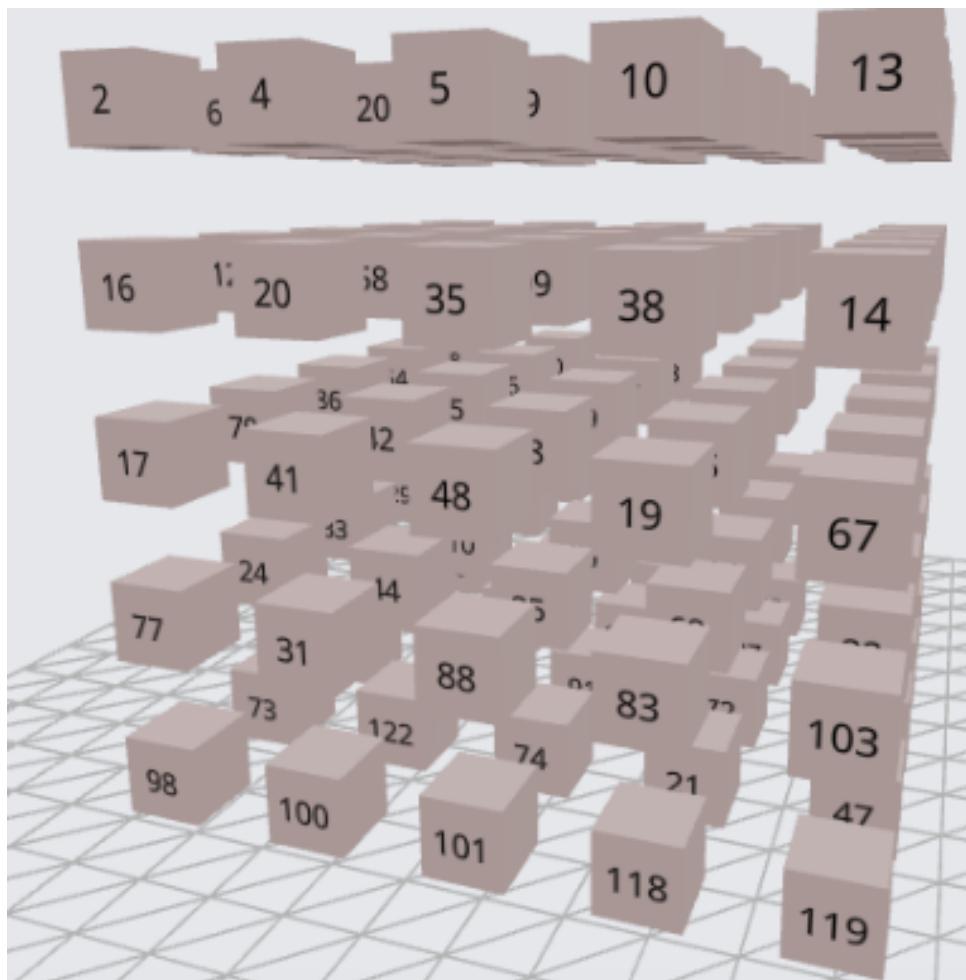


Grafik objective function maksimal per iterasi:



Grafik objective function rata-rata per iterasi:

	<p>Objective Score Mean per Generation</p> <p>Iteration</p>				
	<table border="1"> <tr> <td>Skor final</td><td>-5881</td></tr> <tr> <td>Durasi</td><td>973 ms</td></tr> </table>	Skor final	-5881	Durasi	973 ms
Skor final	-5881				
Durasi	973 ms				
	<table border="1"> <tr> <td>Jumlah populasi</td><td>10.000</td></tr> <tr> <td>Banyak iterasi</td><td>6</td></tr> </table>	Jumlah populasi	10.000	Banyak iterasi	6
Jumlah populasi	10.000				
Banyak iterasi	6				
4	<p><i>Initial state:</i></p> <p><i>Final state:</i></p>				

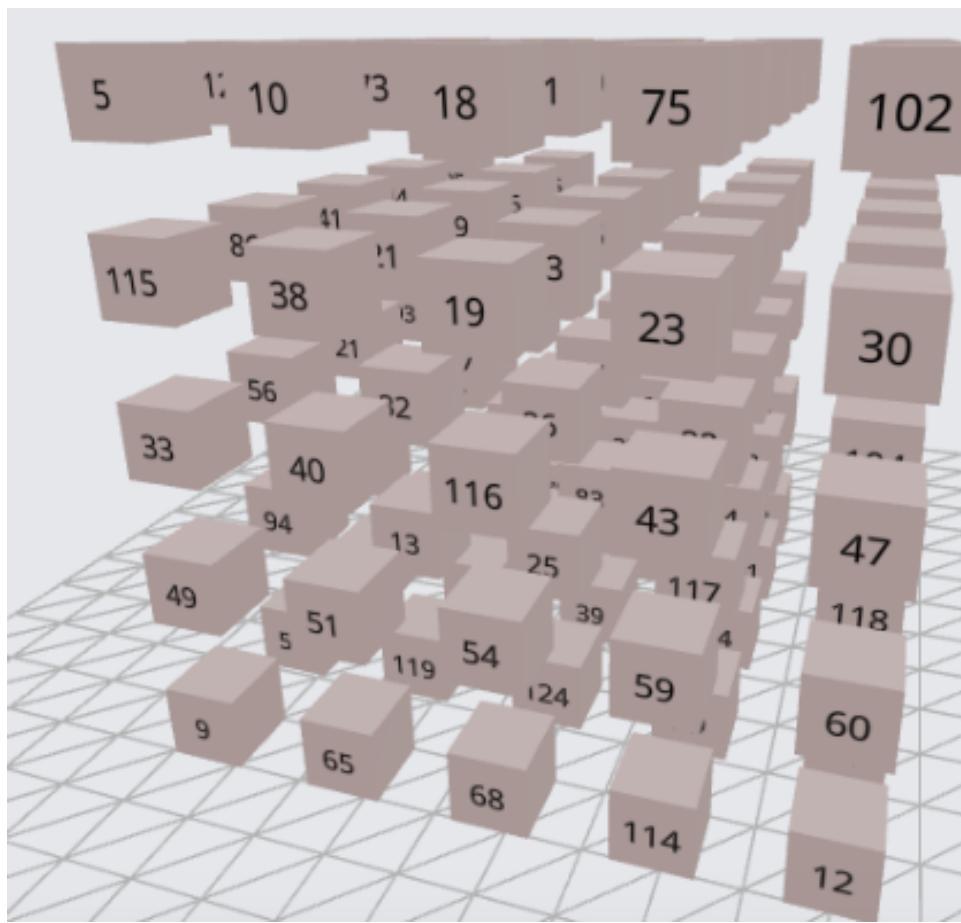


Grafik *objective function* maksimal per iterasi:

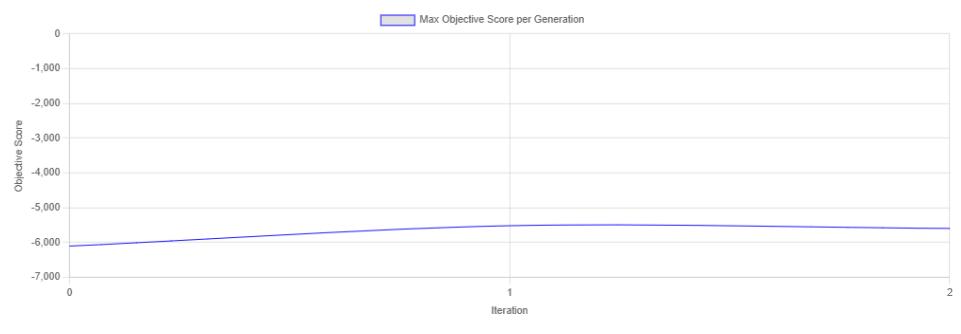


Grafik *objective function* rata-rata per iterasi:

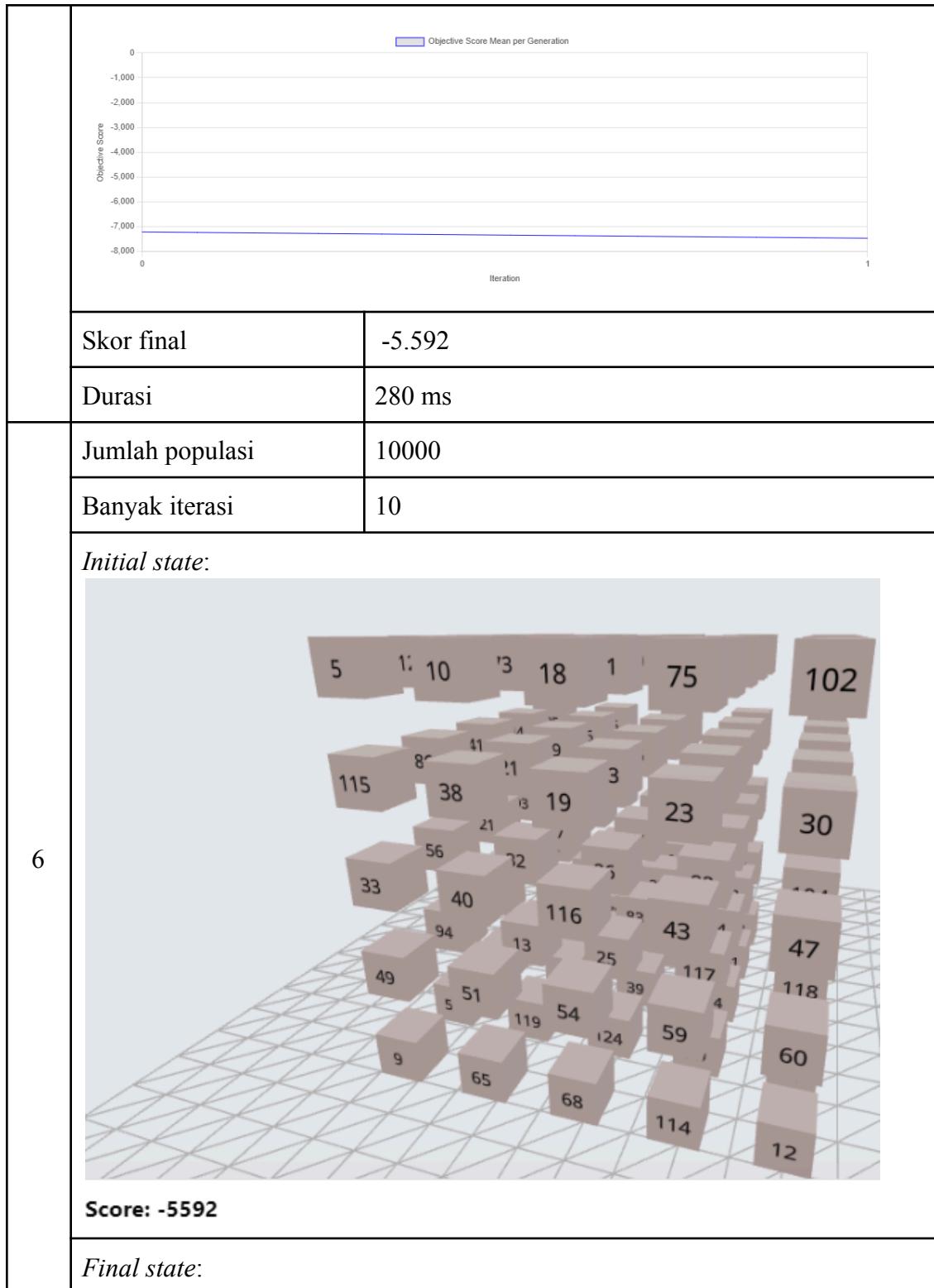
	<p>Objective Score Mean per Generation</p>				
	<table> <tr> <td>Skor final</td><td>-6075</td></tr> <tr> <td>Durasi</td><td>550 ms</td></tr> </table>	Skor final	-6075	Durasi	550 ms
Skor final	-6075				
Durasi	550 ms				
	<table> <tr> <td>Jumlah populasi</td><td>10.000</td></tr> <tr> <td>Banyak iterasi</td><td>2</td></tr> </table>	Jumlah populasi	10.000	Banyak iterasi	2
Jumlah populasi	10.000				
Banyak iterasi	2				
5	<p><i>Initial state:</i></p>				
	<p>Score: -6100</p>				
	<p><i>Final state:</i></p>				

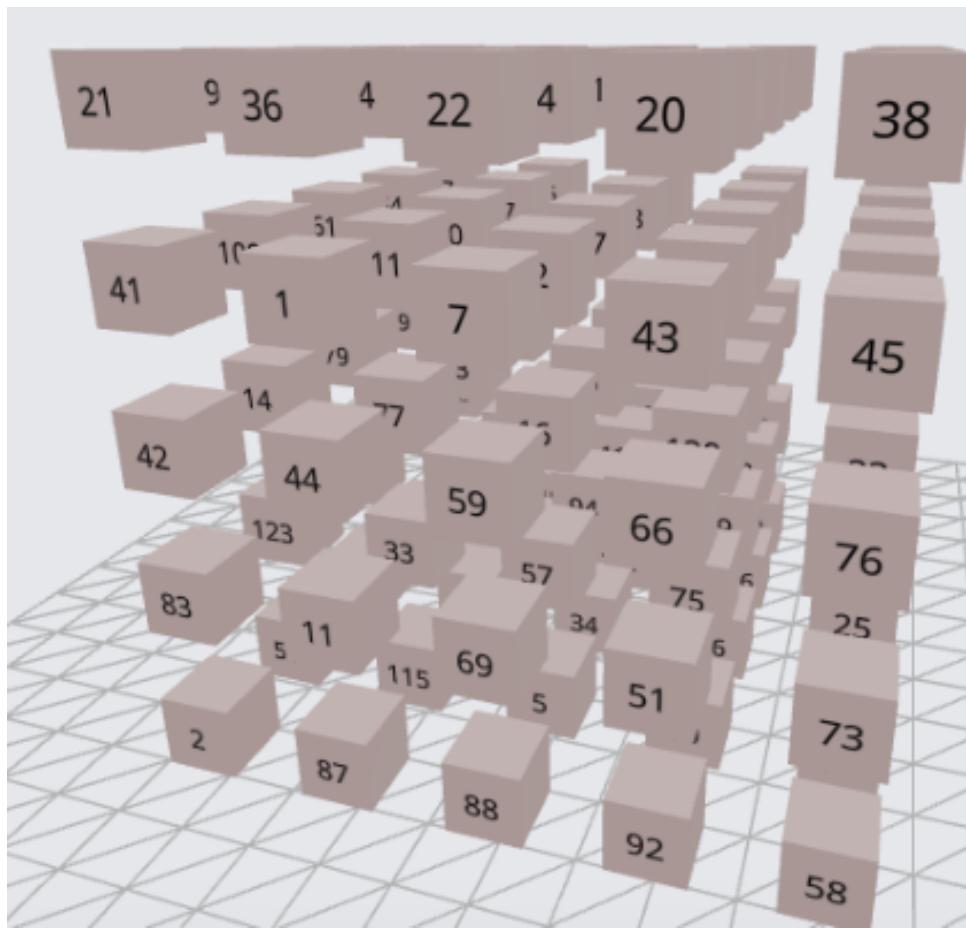


Grafik *objective function* maksimal per iterasi:

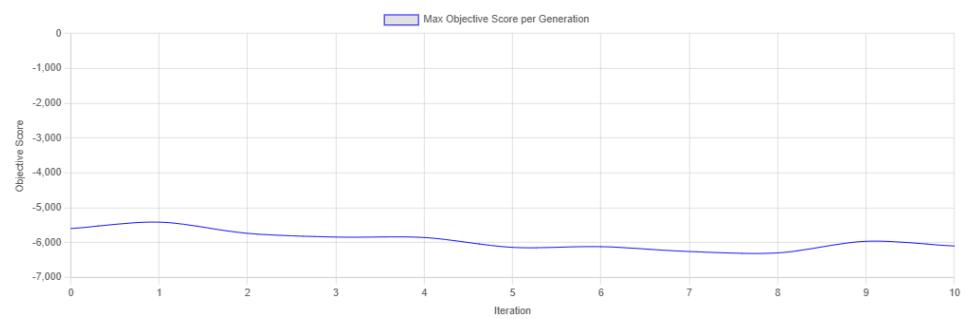


Grafik *objective function* rata-rata per iterasi:

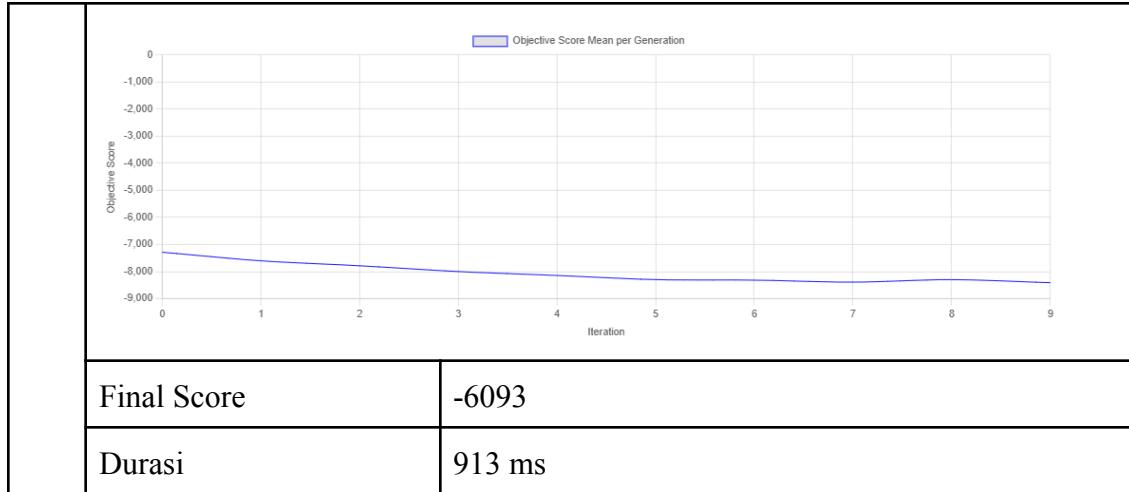




Grafik *objective function* maksimal per iterasi:



Grafik *objective function* rata-rata per iterasi:



Algoritma *Steepest Ascent Hill-Climbing* adalah approach yang paling *straightforward*. Hanya perlu mencari tetangga terbaik dan 'berpindah' ke *state* terssebut. Tetapi untuk mencari tetangga terbaik memerlukan harga komputasi yang cukup mahal karena *scope* dari masalah *diagonal magic cube* sangat luas, dengan jumlah tetangga tiap *state* adalah $125*124/2$ (7750) tetangga. Sehingga, walaupun mendapatkan hasil yang mendekati maksimum, tetap saja merupakan hal yang tidak *plausible*.

Algoritma *Sideways Move Hill-Climbing* adalah modifikasi dari *Steepest Ascent Hill-Climbing* yaitu memperbolehkan pindah ke tetangga yang nilai objektifnya sama dengan *state* saat ini. Dengan perubahan ini, algoritma dapat keluar dari *shoulder* dan naik lebih jauh, lebih konsisten dibandingkan dengan algoritma *Steepest Ascent Hill-Climbing*, tetapi hal ini tidak tanpa biaya, karena memerlukan waktu yang jauh lebih lama dibandingkan *Steepest Ascent Hill-Climbing*.

Algoritma *Random Restart Hill-Climbing*, memiliki beberapa kelebihan dan kekurangan. Kelebihannya adalah dapat memanggil algoritma *Steepest Ascent Hill-Climbing* secara berulang-ulang, tetapi karena Algoritma *Steepest Ascent Hill-Climbing* kurang efisien, Algoritma *Random Restart Hill-Climbing* menjadi sangat lama. Kelemahan yang lain adalah *Random Restart HC* tidak menyimpan state terbaik yang ditemui, hanya berhenti ketika menemui solusi. Konsekuensinya adalah algoritma ini perlu berjalan dengan waktu yang lama jika *scope* dari masalahnya sangat luas. Dapat dilihat dengan melakukan pengulangan 100 kali, algoritma masih tidak mendapatkan jawaban.

Algoritma *Hill-Climbing* yang terakhir adalah *Stochastic Hill-Climbing*. Berbeda dengan algoritma *Hill-Climbing* yang lain yang berbasis *Steepest Ascent HC*, *Stochastic HC* membangkitkan tetangganya secara random kemudian membandingkan nilai objektif tetangga tersebut dengan nilai objektif saat ini. Jika nilai tetangga lebih besar, maka algoritma akan pindah. Jika tidak, maka algoritma akan diam. Algoritma ini memiliki

keuntungan dibandingkan *Steepest Ascent based HC* yaitu perhitungan nilai objektif yang jauh lebih sedikit.

Tabel 7. Data Skor Fungsi Objektif Akhir

Algoritma	Eksperimen ke-			Rata-rata
	1	2	3	
<i>Steepest Ascent Hill-Climbing</i>	-593	-692	-800	-695
<i>Hill-Climbing with Sideways Move</i>	-351	-486	-508	-448
<i>Random Restart Hill-Climbing</i>	-907	-882	-1040	-943
<i>Stochastic Hill-Climbing</i>	-1147	-957	-1181	-1095
<i>Simulated Annealing</i>	-102	-120	-102	-108
<i>Genetic</i> (iterasi konstan)	-5804	-6228	-5881	-5971
<i>Genetic</i> (populasi konstan)	-6075	-5592	-6093	-5920

Berdasarkan data hasil skor fungsi objektif eksperimen yang telah dilakukan, algoritma yang paling efektif untuk menghampiri solusi permasalahan *magic cube* adalah *simulated annealing* karena algoritma tersebut menghasilkan *state* kubus dengan skor fungsi objektif yang paling tinggi. Urutan efektivitas setiap algoritma dalam menyelesaikan persoalan ini dari paling tidak efektif adalah *genetic algorithm* < *stochastic hill-climbing* < *random restart hill-climbing* < *steepest ascent hill-climbing* < *hill-climbing with sideways move* < *simulated annealing*.

Tabel 8. Data Waktu Eksekusi Algoritma dalam Milisekon

Algoritma	Eksperimen ke-			Rata-rata
	1	2	3	
<i>Steepest Ascent Hill-Climbing</i>	794	938	730	820
<i>Hill-Climbing with Sideways Move</i>	2445	2286	3058	2596
<i>Random Restart Hill-Climbing</i>	72319	4145	9928	88392 (sangat bergantung banyak <i>restart</i>)
<i>Stochastic Hill-Climbing</i>	382	284	275	313
<i>Simulated Annealing</i>	573	543	526	547

<i>Genetic</i> (iterasi konstan)	549	230	973	584
<i>Genetic</i> (populasi konstan)	550	280	913	581

Berdasarkan data waktu eksperimen yang telah dilakukan, algoritma yang paling cepat dalam menghampiri solusi permasalahan *magic cube* adalah *stochastic hill-climbing*. Urutan kecepatan eksekusi setiap algoritma dalam menyelesaikan persoalan ini dari paling lambat adalah *random restart hill-climbing* < *hill-climbing with sideways move* < *steepest ascent hill-climbing* < *genetic algorithm* < *simulated annealing* < *stochastic hill-climbing*.

Kesimpulan dan Saran

1. Kesimpulan

Untuk menghampiri solusi permasalahan *diagonal magic cube* dengan algoritma *local search*, *simulated annealing* adalah algoritma dengan performa terbaik, baik dalam harga komputasi (diukur dalam waktu) maupun kedekatan dengan nilai objektif maksimum.

2. Saran

- Untuk tugas/pekerjaan pemrograman berikutnya, sebaiknya kode dibuat lebih *reusable* dan memiliki *coupling* rendah.
- Untuk algoritma *random restart hill-climbing*, simpan *state* terbaik agar tetap mendapatkan hasil yang lebih baik dengan harga komputasi yang digunakan.
- Beli makanan dari Lokal Janari - Puri Indah.

Pembagian Tugas

NIM	Tugas
13522127	Implementasi algoritma <i>hill-climbing</i>
13522137	Implementasi <i>front-end</i> halaman web, termasuk visualisasi data
13522138	Implementasi algoritma <i>genetic</i>
13522144	Implementasi algoritma <i>simulated annealing</i>

Referensi

- Khodra, M. L. (2024). Modul 3: Beyond Classical Search [Slide Presentasi]. Institut Teknologi Bandung.
- Trump, W. (2005). *The Successful Search for The Smallest Perfect Magic Cube* dari <https://www.trump.de/magic-squares/magic-cubes/cubes-1.html> (diakses 31 Oktober 2024)